- Processo:
  - attività di eleborazione guidata da un programma
  - Istanza di un programma in esecuzione
- Il processo è indentificato da un PID (PID = Process IDentifier)
- Per ogni processo attivo il SO deve mantenere in memoria una immagine che possiamo vedere suddivisa in 3 parti:
  - Codice: in genere condiviso e read-only
  - Dati: globali, stack, heap...
  - Dati processo/SO: Interazione fra processo e s.o., es. tabella file aperti (implementazione: parte in tabella processi, parte in tabella globale)

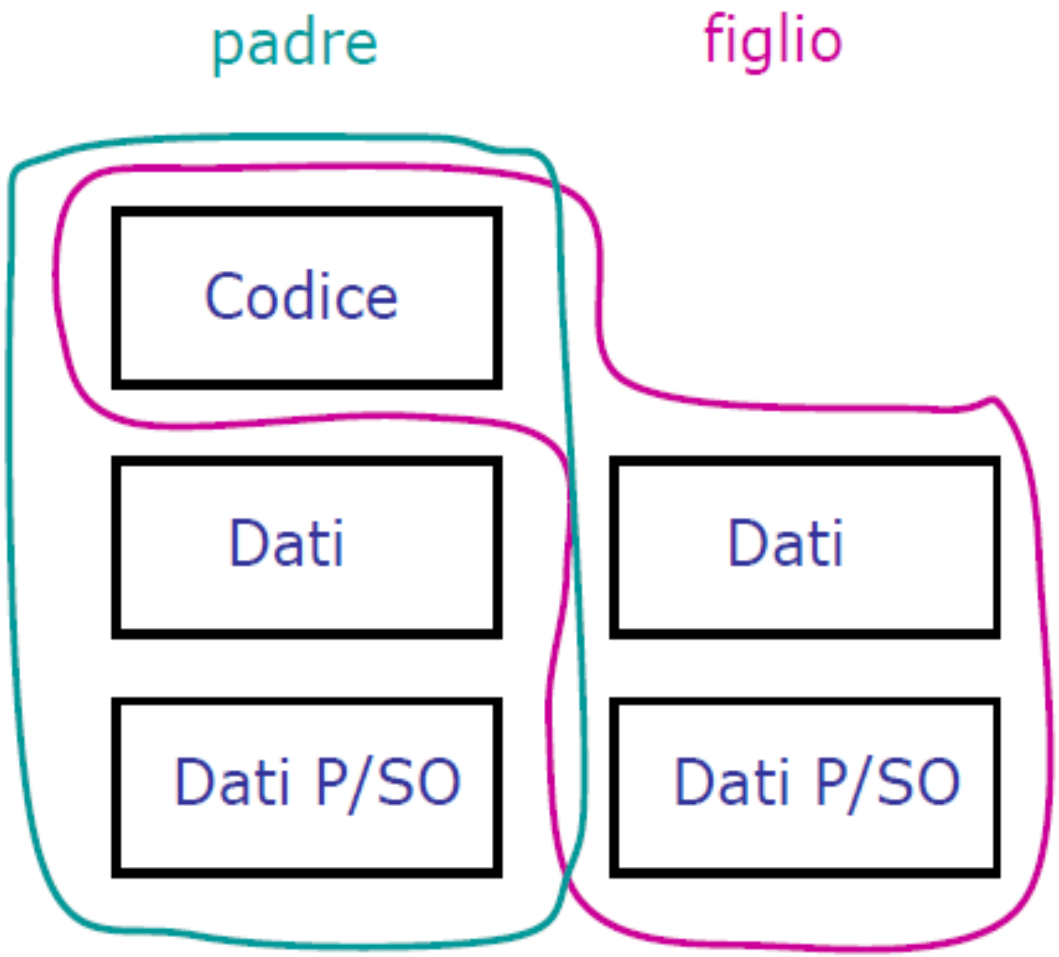
Processi: creazione

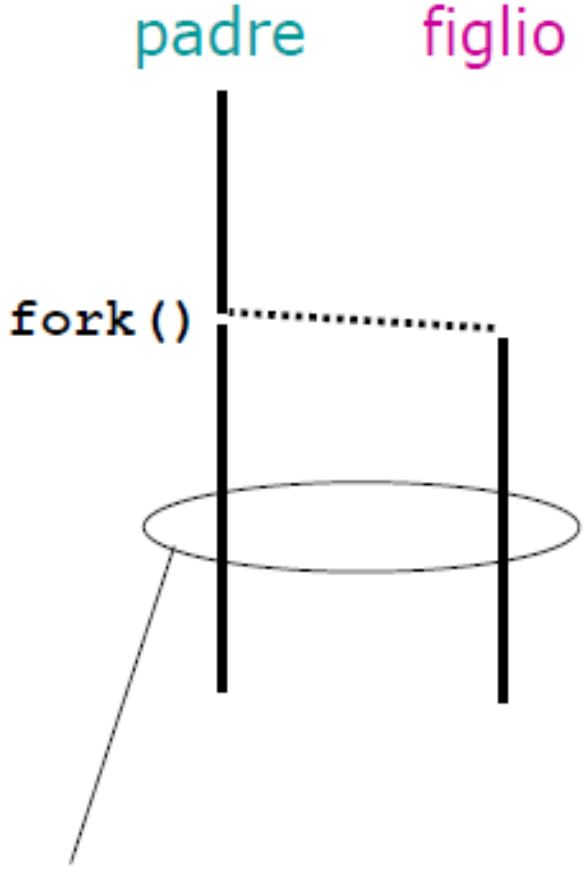
- In Unix un processo si crea con una chiamata di sistema

  fork();

- duplica il processo in esecuzione (“padre” – parent process), creandone una copia (“figlio” – child process) quasi identica:
- ha una copia dell’immagine (il codice può essere condiviso)
- ha un diverso PID
- la funzione fork restituisce al processo padre il PID del figlio (un valore sicuramente ≠ 0)
- il nuovo processo “nasce” uscendo anch’esso dalla chiamata di funzione *fork()* ottenendo però come risultato 0

Processi: effetto della creazione

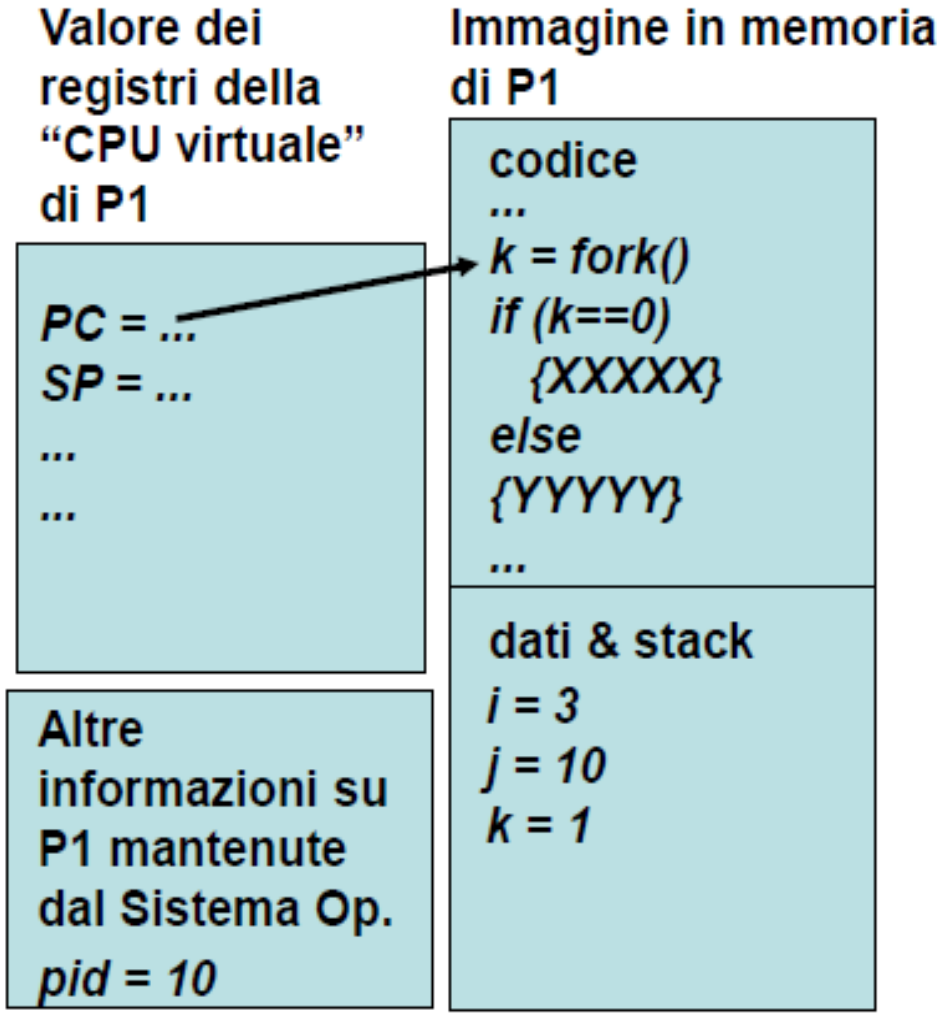




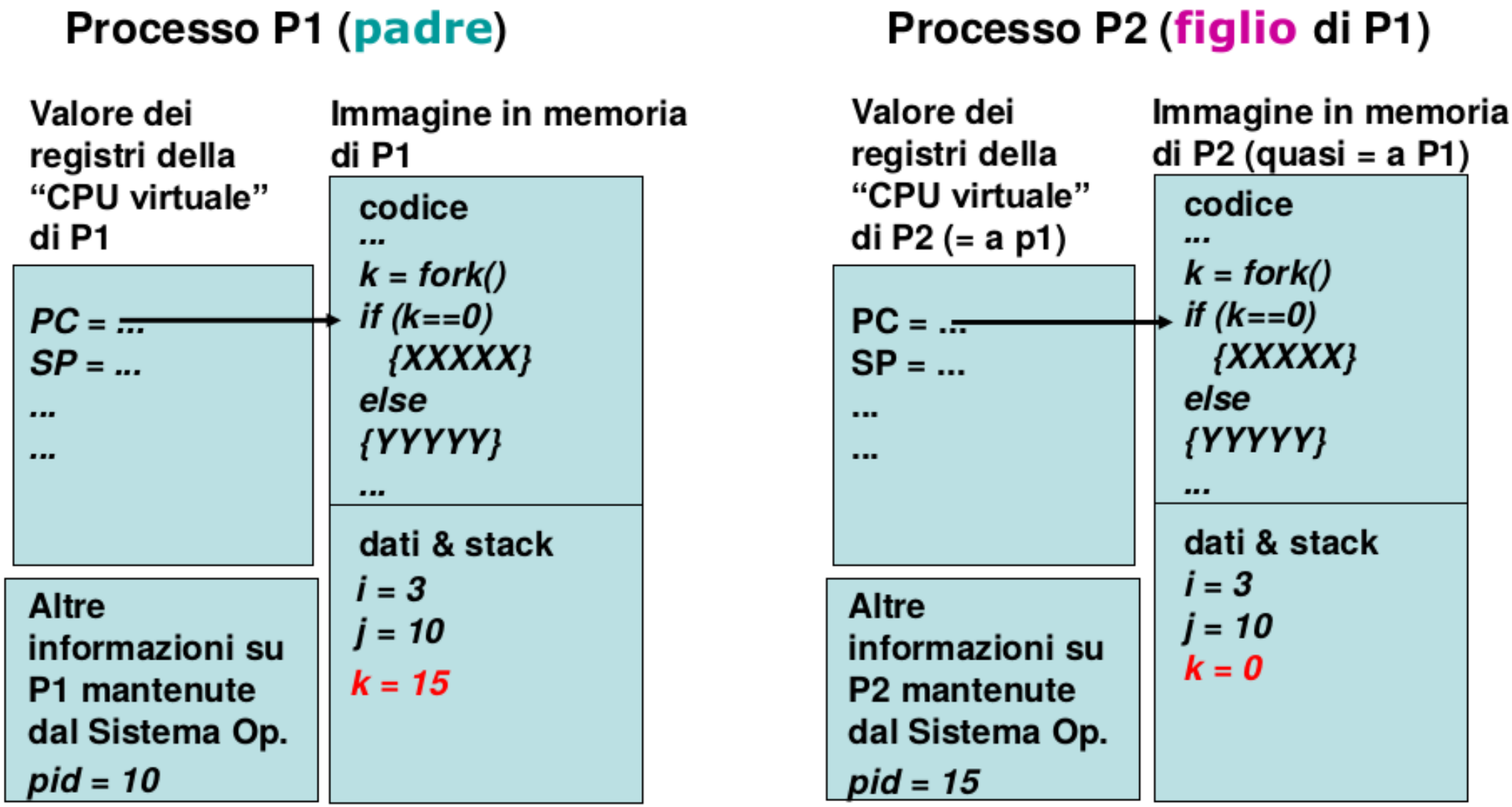
Pseudo-parallelismo (single core)
o reale parallelismo (multi-core)

Processi: effetto della creazione

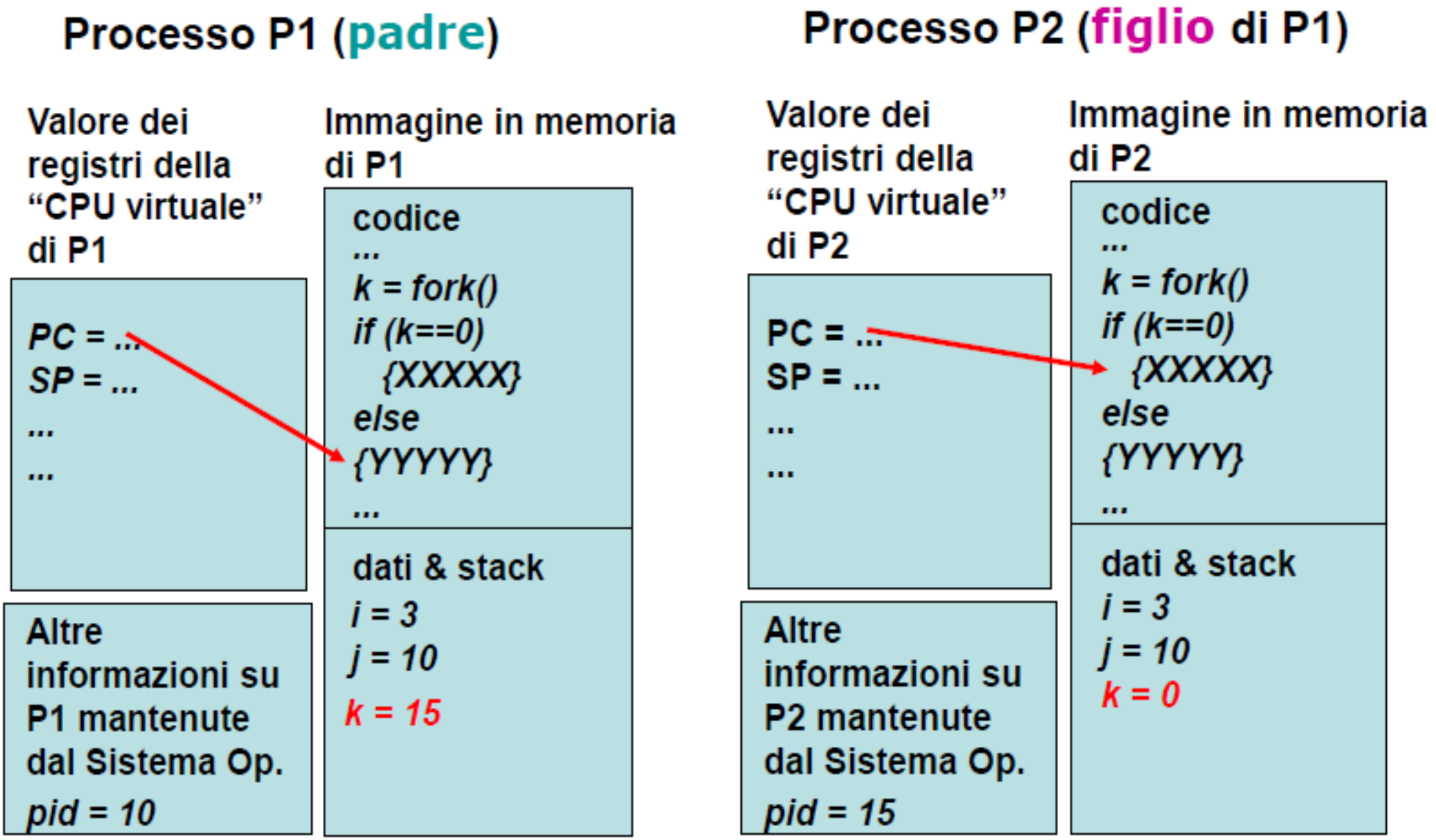
Processo P1 (padre)



Processi: effetto della creazione



Processi: effetto della creazione



Esercizi

- Esaminare ed eseguire il programma *clona.c* da appunti2
  - *Come si comporta?*

```
#include <stdio.h>
#include <unistd.h>

int main()
{

  fork();
  printf("Hello World\n");
}
```

Processi: schema di creazione

```
#include <unistd.h>
#include <sys/types.h>

pid_t pid;


if ((pid = fork()) < (pid_t) 0) {
      /* errore nella fork */
} else if (pid == (pid_t) 0) {
      /* codice eseguito solo dal p. figlio */
} else {
      /* codice eseguito solo dal p. padre */
}
```

- Esaminare ed eseguire *par.c*
  - Modificare il programma affinchè i numeri delle iterazioni di entrambi i cicli vengano passati come argomenti
  - … poi eseguire il programma ridirigendo l'output su file, es. chiamando "par arg1 arg2 > pippo" se "par" è il nome dell'eseguibile.
  - L'alternanza delle stringhe stampate su file dai due processi è diversa rispetto al caso di output su video? Perché?
  - Modificare il programma affinchè le stampe su file risultino come l’output su video

Esercizi – par.c

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

int main()
{

  int i,j;
  pid_t n;

  n=fork();
  if (n==(pid_t)-1)
    {perror("fork fallita");
     exit(1);
    }
  if (n==(pid_t)0) {
      for (j=0;j<50;j++) {
        for (i=0; i< 100000000; i++);
        printf("     Figlio %d \n",j);
      }
  }
  else {
      for (j=0;j<50;j++) {
        for (i=0; i< 100000000; i++);
        printf("Padre %d\n",j);
      }
  }
}
```

Processi: terminazione

- La terminazione normale di un processo avviene:
  - Ritornando dal `main()`
  - Chiamando la funzione `exit()`
    - Vengono chiusi tutti gli stream aperti e svuotati i buffer
  - Chiamando la funzione `_exit()` o `_Exit()`
  - Ritornando dall'ultimo thread di un processo
  - Chiamando `pthread_exit()` dall'ultimo thread di un processo
- La terminazione anormale avviene:
  - Chiamando `abort()`
  - Ricevendo un segnale
- Tutte le funzioni exit si aspettano un argomento intero chiamato *exit_status*
- *Nel main* `return(0)` *è equivalente a* `exit(0)`

Processi: attesa e terminazione

- Un processo p1 può attendere la terminazione di un processo figlio con:

y=wait(&x);

- (e una variante *waitpid* più generale). In realtà non sempre c'è una attesa – ad es. se quando viene chiamata c'è già un figlio terminato, ma in entrambi questi casi wait, all'uscita della funzione, informa p1 che un suo processo figlio p2 è terminato:
- il PID del processo terminato (p2) viene restituito dalla funzione
- nell'intero il cui puntatore è stato passato a wait vengono messe informazioni su come p2 è terminato (di sua iniziativa, o è stato interrotto tramite i "segnali" più avanti nel corso)
- Queste informazioni vengono (per default) mantenute nella tabella dei processi dopo che un processo termina, in attesa di essere lette con wait dal processo che lo ha generato; una volta che sono state lette, per default l'elemento della tabella relativo al processo viene etichettato libero, in modo da non occupare un posto

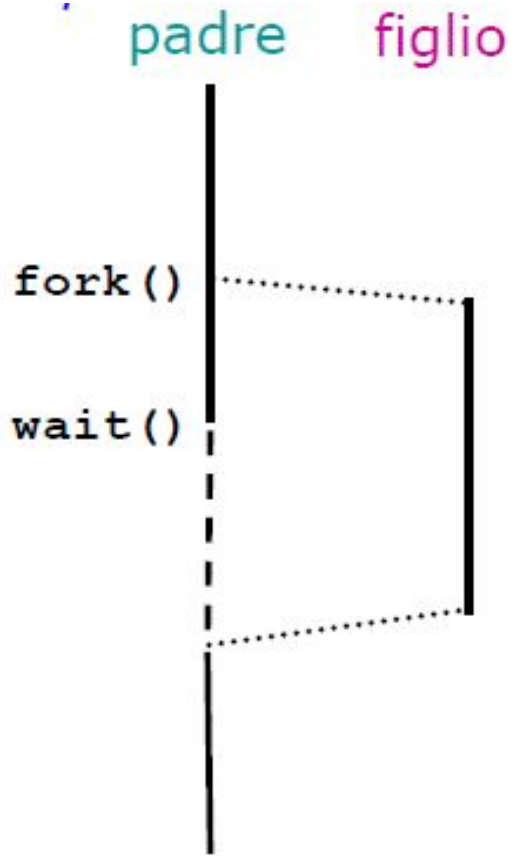
Se p1 ha almeno un processo figlio p2 terminato che non è ancora stato “aspettato” (nessuna wait ha dato a p1 l’informazione che p2 è terminato, o comunque quell’informazione è ancora nella tabella), allora p1 esce subito dalla wait, con l’identificatore di p2

Altrimenti, se p1 ha processi figli non terminati, viene sospeso fino a quando uno di questi termina

Altrimenti? (tutti i p. figli, se ce ne sono stati, sono terminati e sono stati “aspettati”)

R: dalla chiamata si esce subito con errore.

Se p1 venisse sospeso, rimarrebbe sospeso “per sempre”



Esercizi – generazione più processi

- Nel programma vengono generati 5 processi figli che vanno ad eseguire una stessa funzione con valori diversi del parametro:

```
::::::::::::::
5figli.c
::::::::::::::
#include <stdio.h>
...
void proc(int i)
{ int n;

  printf("Processo %d con pid %d\n",i,getpid());
  for (n=0;n<500000000;n++);
}

main()
{
int i;
pid_t pid;

for(i=0;i<5;i++)
     if (fork()==0)
          { proc(i); exit(0);}
for(i=0;i<5;i++)
     { pid=wait(NULL);
       printf("Terminato processo %d\n",pid);
     }
}
```

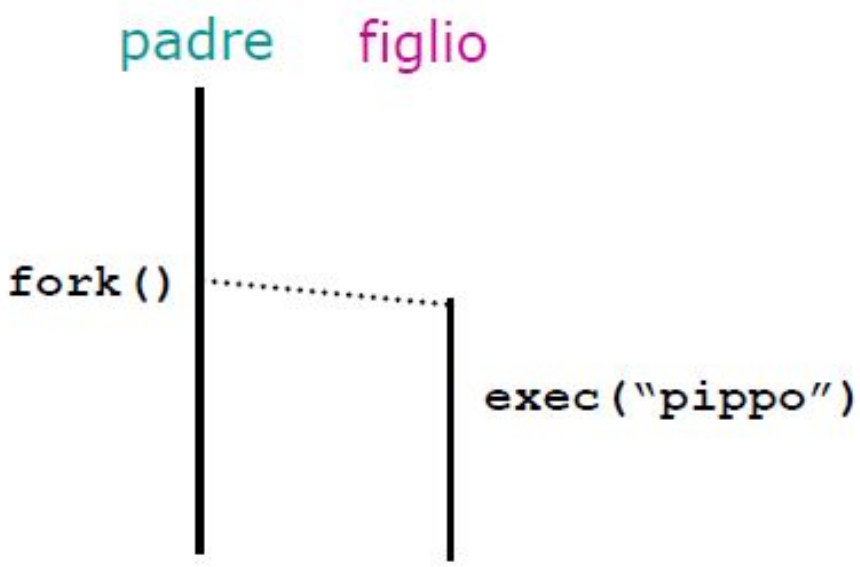
Cosa succede rimuovendo la exit(0)?

Scrivere un programma analogo usando `waitpid` al posto di `wait` – vedi manuale

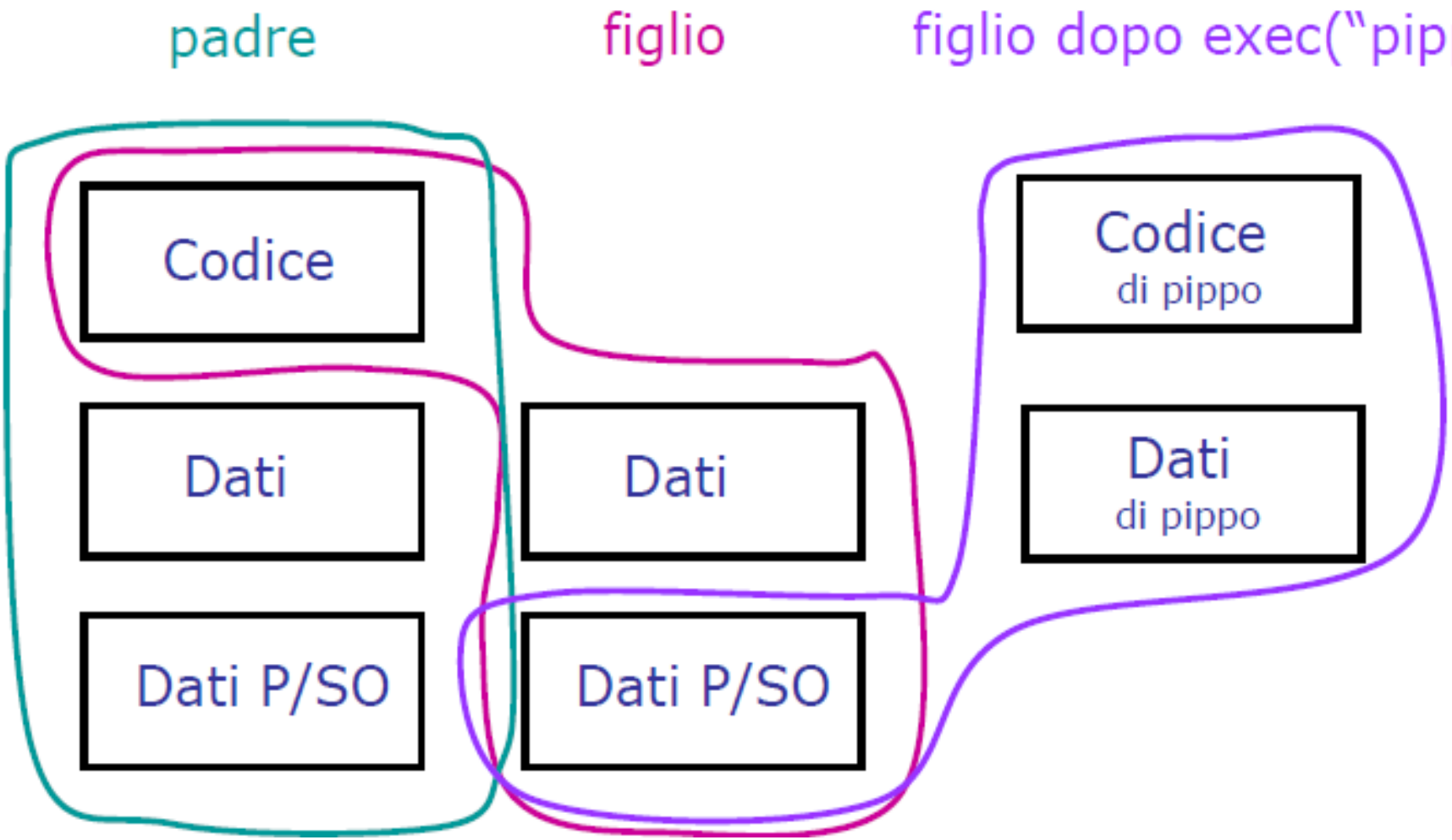
- Modificare uno degli esempi precedenti, aggiungendo una variabile che permetta di verificare che processo padre e figlio hanno due copie delle variabili: il figlio eredita i valori precedenti alla fork, ma le modifiche successive (visualizzate con delle printf) nei due processi sono indipendenti.

- Clonare un processo identico non pare molto utile...
- Con *if (fork()==0)* ...
  - bisogna scrivere nello stesso programma l'intero codice eseguito dal p. figlio
- In effetti, spesso non si fa così, nel ramo del p. figlio si può usare una chiamata di sistema della "famiglia" exec

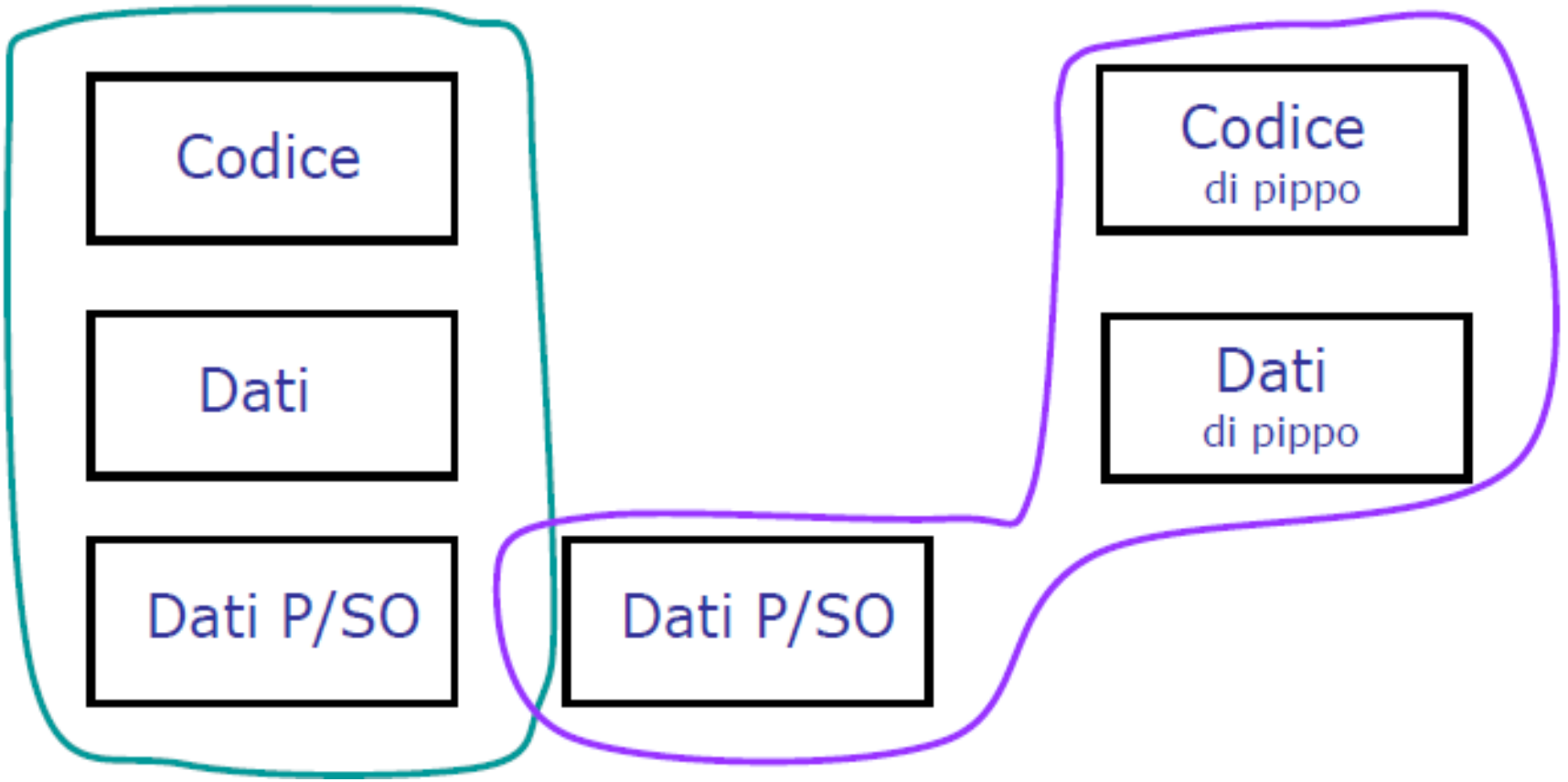
Per la precisione sono diverse funzioni C che permettono di accedere alla stessa chiamata di sistema, qui semplificate in: *exec(nome_file_eseguibile)*



Processi: esecuzione di un programma



Processi: esecuzione di un programma



NB arrivati a questo punto, fra il p. figlio e il "Codice" iniziale non c'è più alcuna relazione

Processi: *exec* in dettaglio

- *int execl(const char *pathname, const char *arg0, ... /* (char *) NULL */);*
- *int execv(const char *pathname, char *const argv[]);*
  - Sostituiscono il codice (e i dati) attuali del processo chiamante con quelli dell'eseguibile *pathname*, restituisce -1 in caso d'errore
- Cambia la modalità con cui si passano gli argomenti

- In execl si specifica una lista di argomenti da passare all'eseguibile terminati da NULL. Es:
  - *execl("/bin/ls", "ls", "-l", (char *) NULL);*
- In execv si passa un vettore di stringhe contenente gli argomenti da passare all'eseguibile terminati da NULL:
  - *arg={"ls", "-l", NULL};*
  - *execv("/bin/ls", arg);*

Processi: *exec* in dettaglio

- *int execlp(const char *filename, const char *arg0, ... /* (char *) NULL */);*
- *int execvp(const char *filename, char *const argv[]);*
  - Sostituiscono il codice (e i dati) attuali del processo chiamante con quelli dell'eseguibile *filename*, restituisce -1 in caso d'errore
- In execlp/execvp si passa il nome dell'eseguibile cercandolo nelle directory specificate in PATH. La variabile d'ambiente viene ereditata dal processo chiamante.

Esercizi – exec

- Esaminare il programma provaexec.c ed hello.c in appunti2 e provate ad eseguirlo:

```
::::::::::::::
provaexec.c
::::::::::::::
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
main()
{
  pid_t n;
  int s;
  if((n=fork())== (pid_t)-1)
    {perror("fork fallita");
     exit(1);
    }
  else if (n==(pid_t)0)
         {/* processo figlio */
         execl("hello","hello",NULL);
         perror("exec fallita");
         }
       else
         {/* processo padre */
         wait(&s);
         }
}
```

```
::::::::::::::
hello.c
::::::::::::::
main()
{
  printf("Hello \n");
}
::::::::::::::
```

Esercizi – exec

- Modificare il precedente esercizio usando la funzione execv al posto di execl
- Verificare i comportamenti indesiderati che si possono ottenere se si fa una stampa con printf che NON causa lo svuotamento del buffer e poi si chiama:
  - Exec
  - Fork

  per testarlo potete modificare in modo opportuno provaexec.c

Esercizio

- Scrivere un programma che, preso come argomento da linea di comando un percorso ad un file (eventualmente da creare), generi 3 processi nell'ordine P1, P2 e P3. Ciascuno di essi scrive sul file la stringa:
  - Sono il processo P<i> con pid: <pid>\n
- I processi devo scrivere sul file in ordine inverso alla loro creazione,

  ossia si deve ottenere nel file la sequenza:

  Sono il processo P3 con pid: <pid3>\n

  Sono il processo P2 con pid: <pid2>\n

  Sono il processo P1 con pid: <pid1>\n
- Verificare che l'ordine sia preservato indipendentemente dallo scheduling della CPU, ad esempio ritardando con una `sleep` la scrittura da parte del processo P3