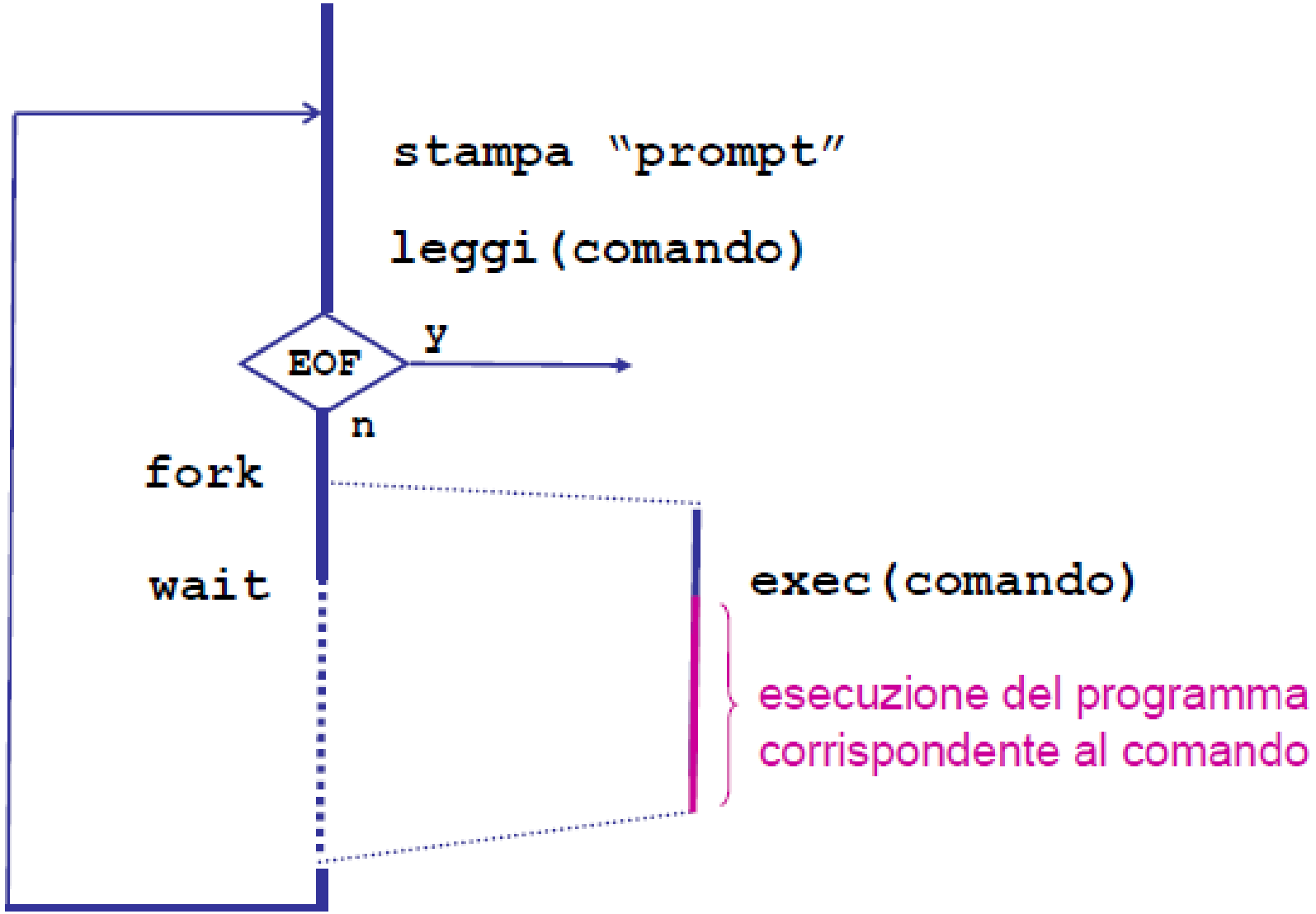
- Funzionamento di base (supponendo un solo comando per riga):



- Perché creare un nuovo processo? Al solito, varie risposte:
- 1. La più importante: il programma eseguito dall'interprete e quello del comando devono avere ben poco contesto in comune, ad es.:
  - codice e dati sono ben diversi
  - il comando non deve poter accedere a tutti i dati dell'interprete
- 2. Per come è l'**exec**, se la facesse direttamente il processo che legge il comando, una volta finito il programma passato a exec, il processo terminerebbe senza leggere i comandi successivi
- 3. L'interprete per default attende che il processo termini, ma non necessariamente; volendo (scrivendo "**comando &**"), il comando gira "in background" (nello sfondo) e nel frattempo si può usare l'interprete per lanciare altri comandi e controllare l'esecuzione del comando in corso

La shell, l'interprete dei comandi

- Il codice di un semplice interprete, la "small shell" (da Haviland, Gray & Salama, "Unix System Programming", 2nd ed., Addison-Wesley 1998) realizza lo schema precedente
- Il sorgente è suddiviso(come es. di compilazione separata) in 2 file:
  - smallsh.c, che contiene le funzioni di maggior interesse per questo corso (effettuano le chiamate di sistema)
  - input.c, che contiene le funzioni per la lettura dell'input
  - da compilare digitando ***make*** da linea di comando nella directory dell'interprete
- Il main è del tipo

```
Loop
    leggi una riga
    processa una riga
end
```

Su una riga di solito c'è un comando solo, processa una riga ma non sempre, es.:
comando1 ; comando 2 // sequenza
comando1 | comando 2 // pipeline,
// non trattata nella versione base

- Una riga contiene una sequenza di "simboli":
  - simboli speciali di un solo carattere, nella shell base: fine riga, "**;**", "**&**"
  - sequenza di caratteri non speciali (e non spazi): nomi di comandi o argomenti

NB: gli spazi servono a separare i simboli

Ad esempio, la riga:

ls qui quo qua <return>

contiene 5 simboli e un comando, la riga:

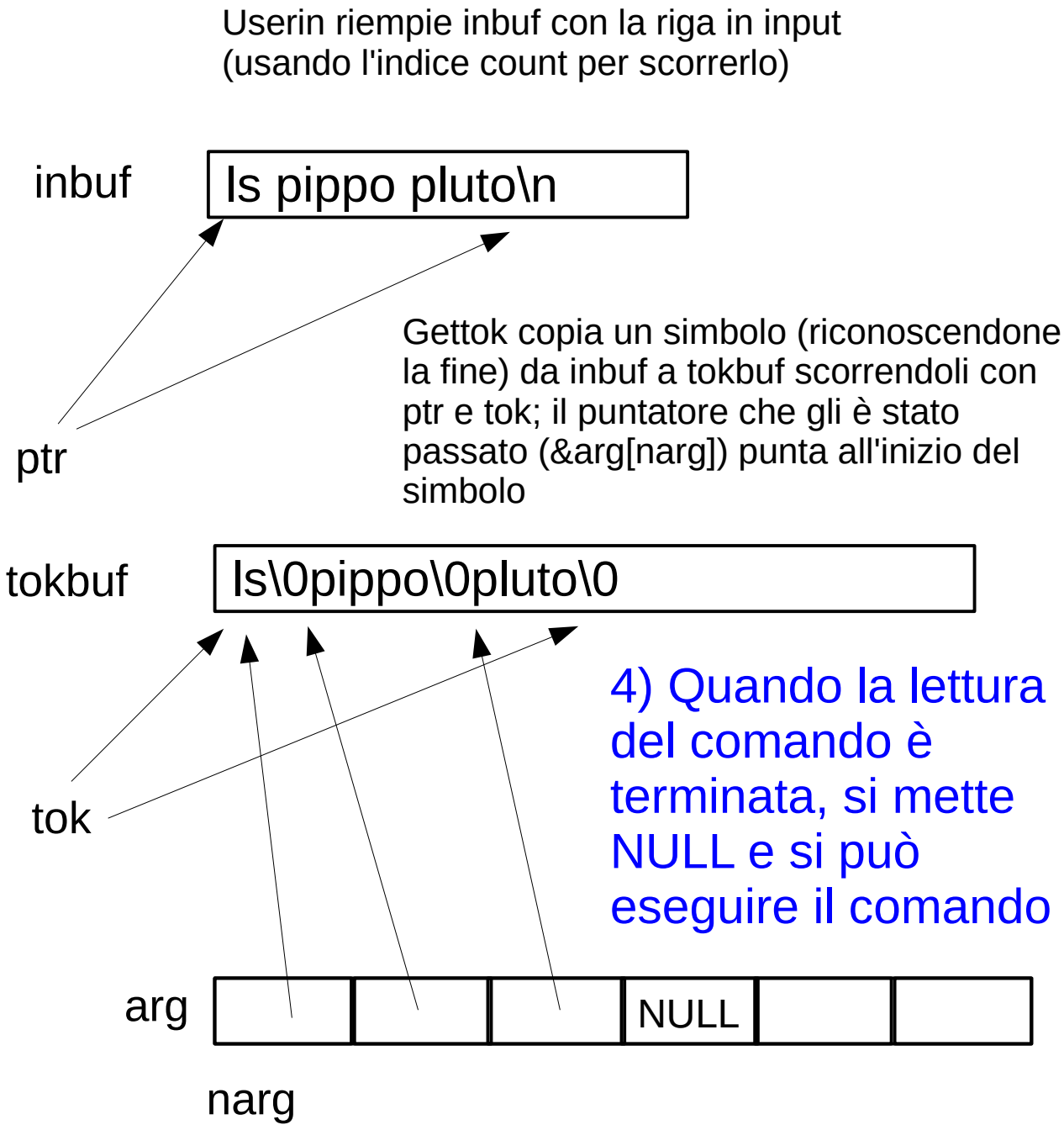
ls qui ; ls quo <return>

contiene 6 simboli e 2 comandi

1) La riga in input viene parcheggiata in un array di caratteri

2) ogni volta che serve conoscere il prossimo simbolo, si scorre l'array copiando il simbolo in un altro, terminando con "\0" (= fine stringa in C)...

3) ... riempiendo anche un elemento di un array di puntatori da passare a **execvp**



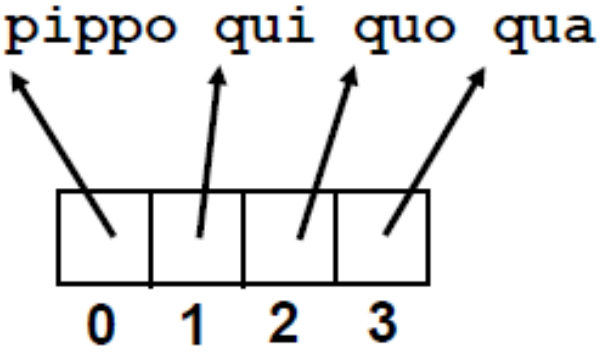
Argomenti sulla linea di comando

- Indipendentemente dalla versione di **exec** utilizzata, il programma (il cui nome di file viene passato come primo parametro a exec) accederà agli argomenti nel noto modo se

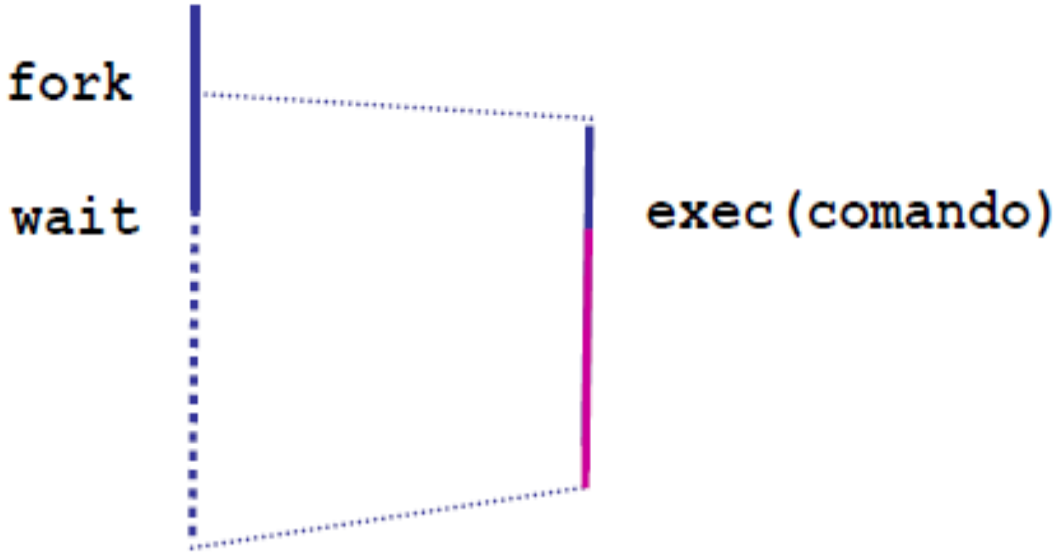
argc: contatore degli argomenti
argv: vettore degli argomenti

```
main(int argc, char *argv[])
{
   /* qui argv[1], ... , argv[argc-1] sono
      le stringhe passate come argomento */
}
```

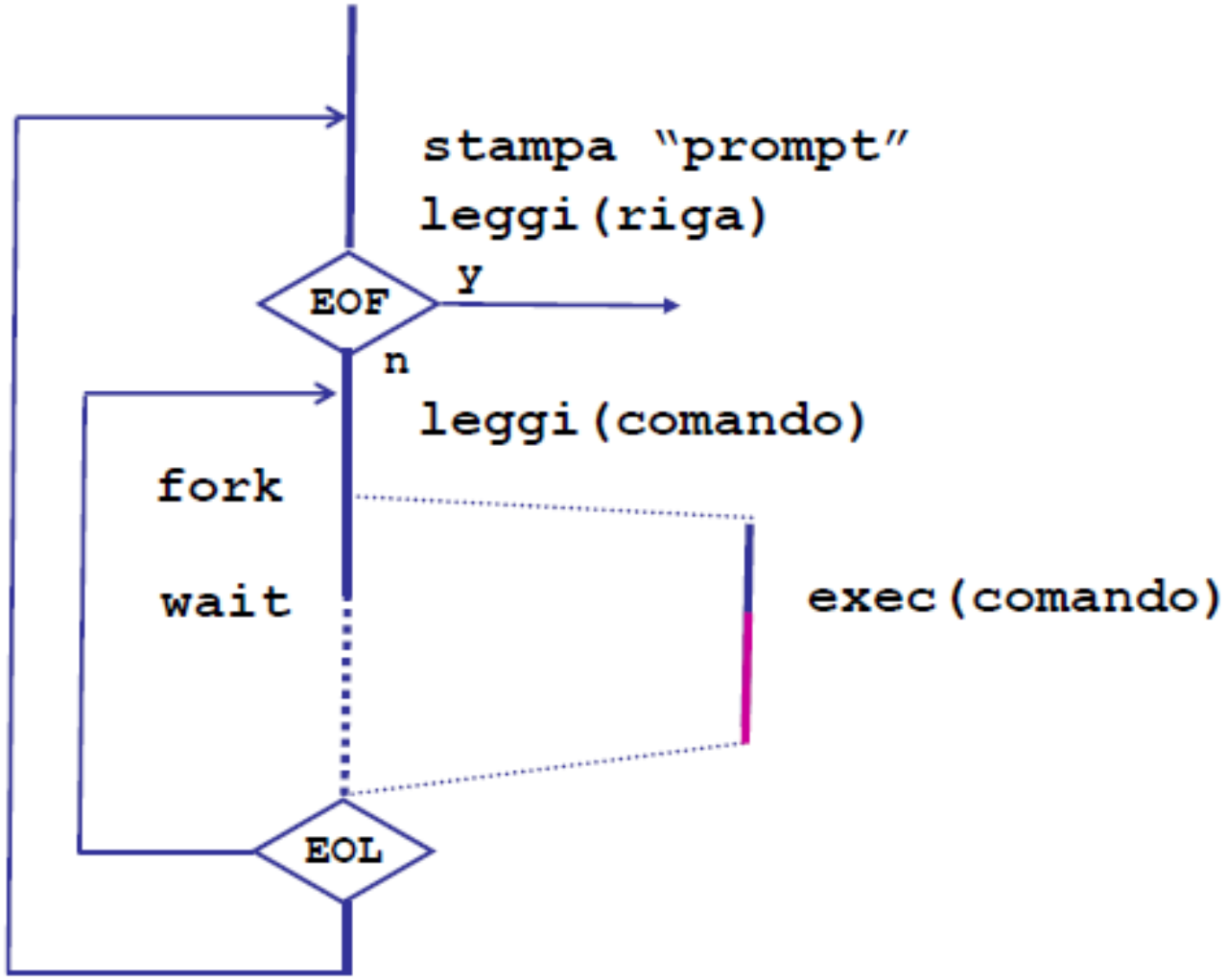
- Ad esempio per il comando: **pippo qui quo qua** si può costruire un array come segue, passarlo a **execv**(«v» per «vettore degli argomenti), e il programma se lo ritrova in **argv**, con **argc==4**



- L’esecuzione del comando è la combinazione fork/exec/wait già vista:



- Ma la struttura complessiva è leggermente più complicata di quella vista inizialmente:



La bash (Bourne Again Shell) e altre shell esistenti, fanno molto di più, ad esempio:

- Comando «in background», cioè non si aspetta la terminazione del processo che lo esegue (ma poi bisogna dare notizia della sua terminazione):

```
nomecomando arg1 ... argN &
```

- Ridirezione standard I/O con la notazione :

```
nomecomando arg1 ... argN < filein
nomecomando arg1 ... argN > fileout
nomecomando arg1 ... argN 2> errors
```

- Pipeline:

```
comando1 | ... | comandoN
```

- ...

- Modificate il codice shell affinché lanci comandi «in background» cioè non aspetti la terminazione del processo che lo esegue
  - Inizialmente fate in modo che tutti i comandi siano lanciati in background
  - In seguito fate in modo che solo in caso di presenza del simbolo &, venga lanciato il comando in background
    - Fate attenzione che venga trattata correttamente l'esecuzione di un comando in background seguito da uno in foreground