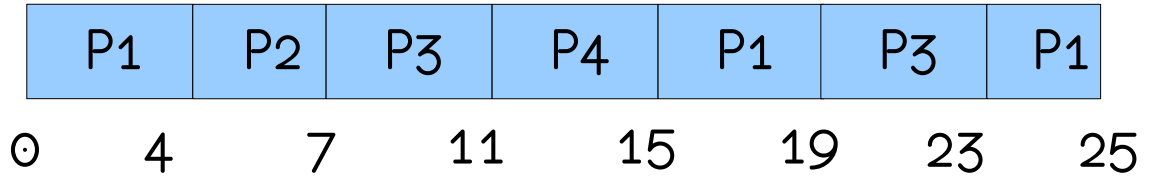
Scheduling a Priorità

- **Ogni processo ha associata una priorità che decide l'ordine di assegnazione della CPU. Può essere con o senza prelazione**
- La priorità può essere definita:
  - **Internamente**: sulla base di caratteristiche del processo. Es. SJF è un algoritmo di scheduling basato su priorità definita internamente, priorità = inverso della durata stimata del CPU burst
  - **esternamente**: non calcolabile, imposta in base a criteri esterni, dagli utenti per esempio

Round-robin

- Ideato espressamente per **sistemi time-sharing**, è preemptive. La coda ready è circolare, gestita FIFO. A ogni processo viene assegnata la CPU per un quanto di tempo, se non è sufficiente a concludere, il processo viene reinserito in coda ready e la CPU riassegnata al successivo
- **Esempio**, supponiamo di avere i soliti quattro processi e relativi CPU burst e che il quanto sia lungo 4:

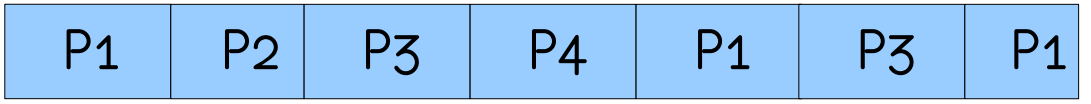
P	D
P1	10
P2	3
P3	8
P4	4



Round-robin

- Ideato espressamente per **sistemi time-sharing**, è preemptive. La coda ready è circolare, gestita FIFO. A ogni processo viene assegnata la CPU per un quanto di tempo, se non è sufficiente a concludere, il processo viene reinserito in coda ready e la CPU riassegnata al successivo
- **Esempio**, supponiamo di avere i soliti quattro processi e relativi CPU burst e che il quanto sia lungo 4:

P	D
P1	10
P2	3
P3	8
P4	4



0 4 7 11 15 19 23 25

tma = (0 + 4 + 7 + 11 + 11 + 8 + 4) / 4 = 45 / 4 = 11,25

Round-robin

- il tma è abbastanza alto ma **non si ha starvation**:

  ogni processo viene servito ogni **(n_proc – 1) * quanto** millisecondi
- All'utente sembra di avere a disposizione una CPU solo un po' più lenta (**1/n, n = num. processi**)
- La scelta del quanto è critica:
  - Se è molto lungo, allora RR tende a diventare un FCFS
  - Se è troppo corto, allora i tempi necessari ai cambi di contesto rischiano di appesantire e rallentare molto il processamento

Code a multilivello / con feedback

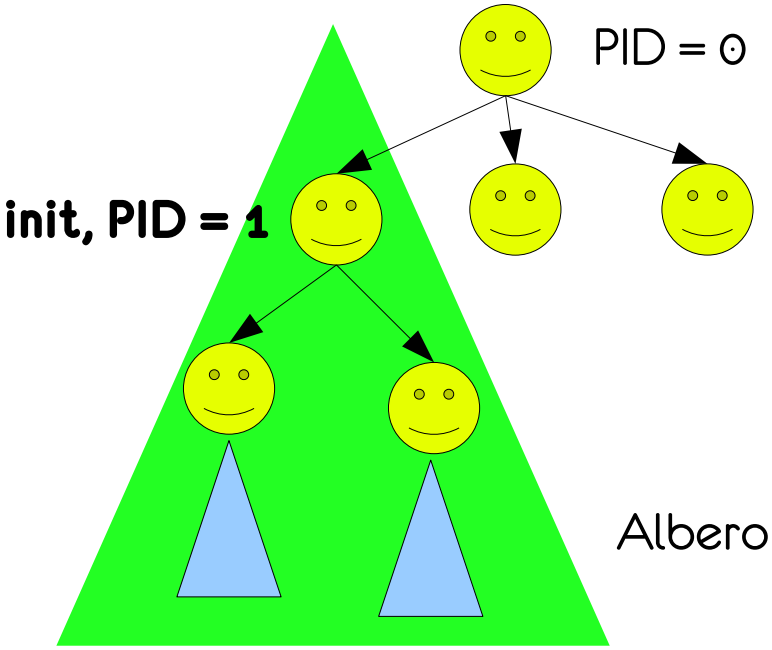
- Algoritmi utili quando è possibile distinguere i processi in categorie legate alla loro natura
- La ready queue è suddivisa in tante code quante sono le categorie
- Ogni coda può avere un algoritmo di scheduling diverso
- Esiste una priorità fra le code (eventualmente associata a meccanismi di invecchiamento per evitare starvation)
- Se ai processi è consentito cambiare coda allora si parla di multilivello con feedback

creazione e terminazione

sezione 3.3 del libro (VII ed.)

Creazione

- come nasce un processo?
- un processo viene generato dall'unica entità attiva gestita dal SO: **un altro processo**
- con l'avvio del SO si genera un **albero di processi** che cresce e decresce dinamicamente con l'evoluzione dell'elaborazione



PID = Process Identifier

Albero di tutti i processi del sistema

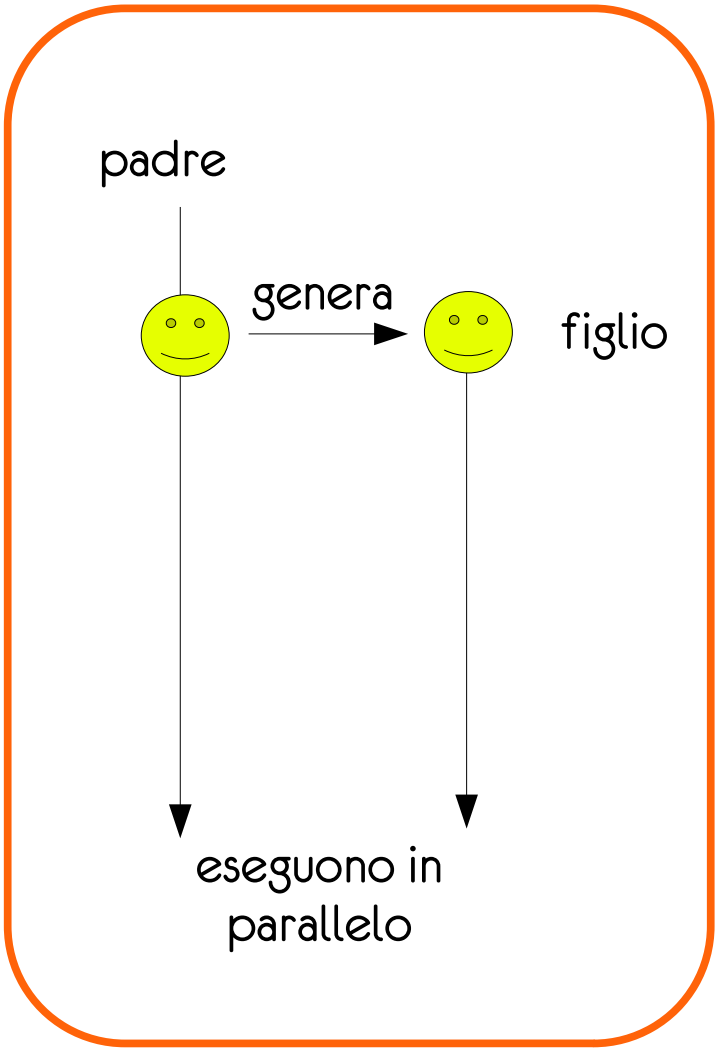
Creazione

- Ogni processo ha un identificatore univoco (numero intero), il **process identifier** o **PID**
- Ogni processo generatore è detto “**padre**”, ogni processo generato è detto “**figlio**”
- Un processo padre in generale **condivide delle risorse** (es. file aperti) con i propri processi figli, certe volte un figlio può usare solo un sottoinsieme delle risorse del padre
- Il processo padre può avere necessità di **passare dei dati** al processo figlio, che viene generato per eseguire una particolare elaborazione. Per esempio in Unix il figlio riceve **una copia di tutte le variabili** del padre

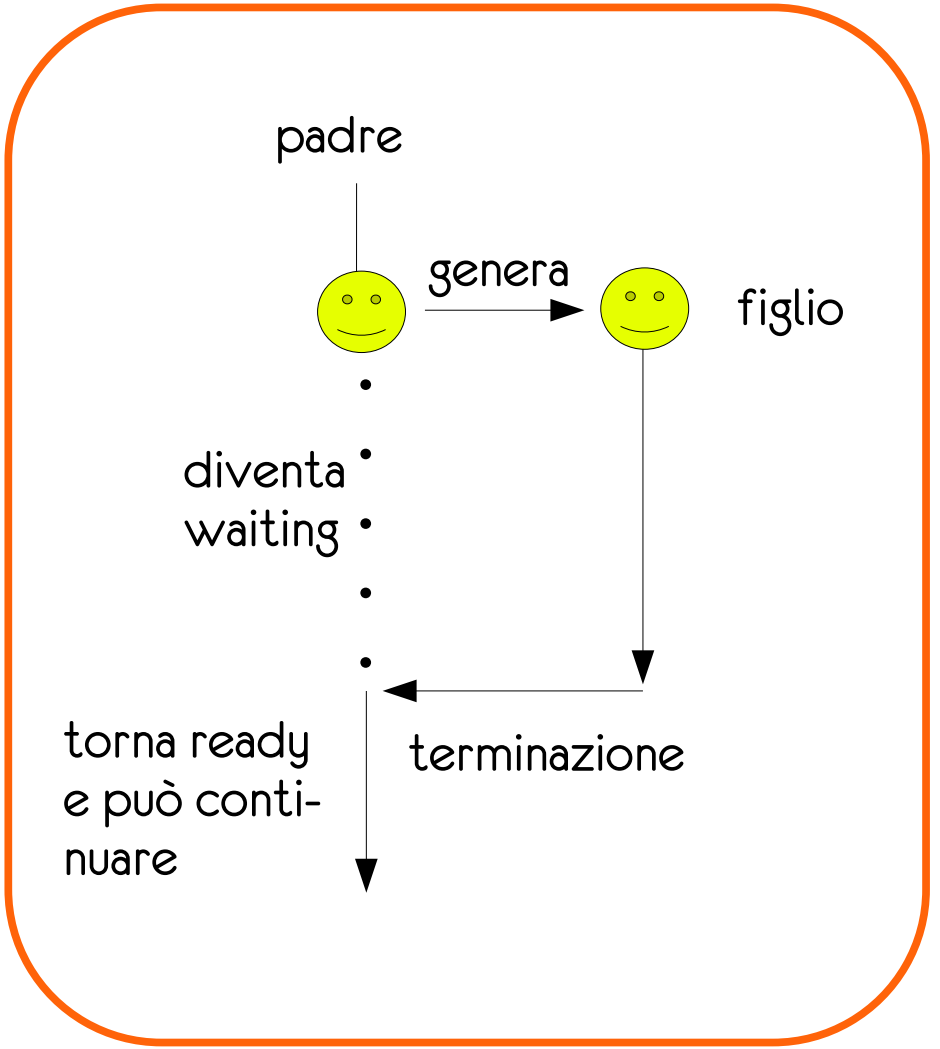
Creazione

- **Esecuzione**:
  - A) il padre continua la propria esecuzione in modo concorrente a quella dei figli
  - B) il padre si sospende in attesa della terminazione di tutti i figli
- **Programma eseguito**:
  - A) il figlio è una copia del processo padre
  - B) il figlio esegue un programma diverso
- in Unix un processo figlio inizialmente è una copia del processo padre e i due eseguono in parallelo. Un processo può cambiare il programma che esegue tramita una system call (**exec**)

Creazione: fork



CASO A



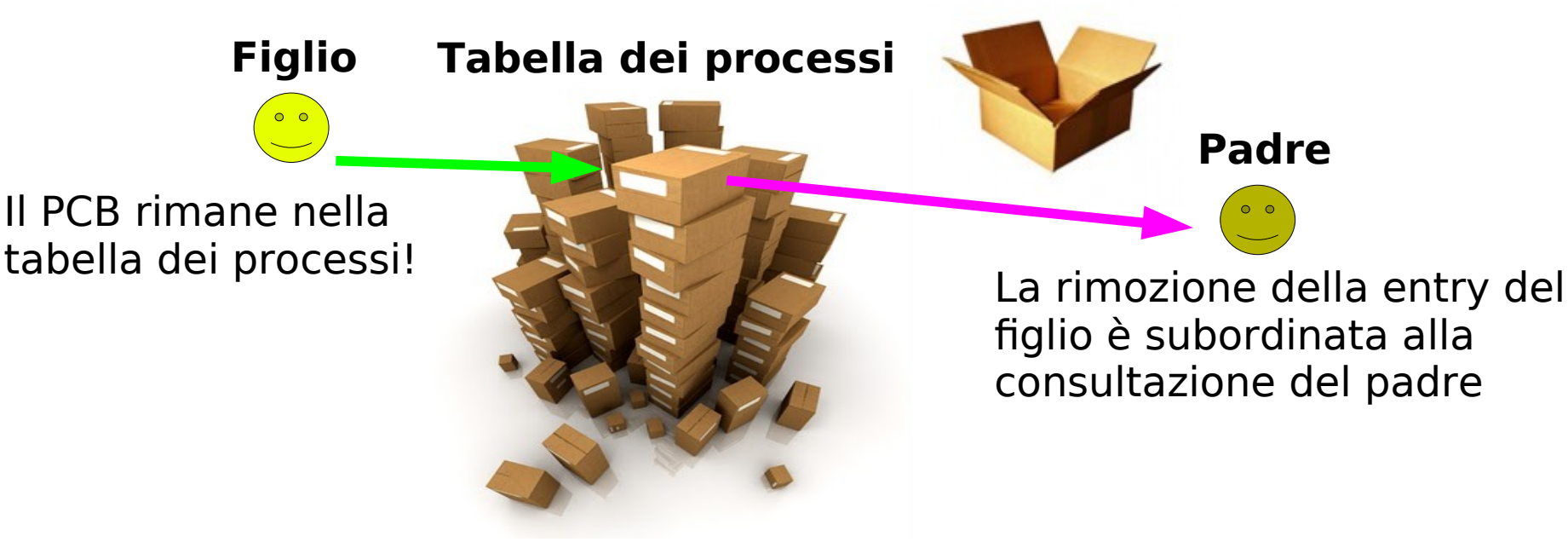
CASO B

Terminazione

- **Un processo giunto alla sua ultima istruzione notifica al SO che è pronto per terminare tramite la system call exit**
- Il SO libera le risorse allocate per il processo. Se il padre del processo terminato attendeva la sua terminazione, **questa gli viene notificata,** insieme ad alcuni dati inerenti la terminazione del figlio

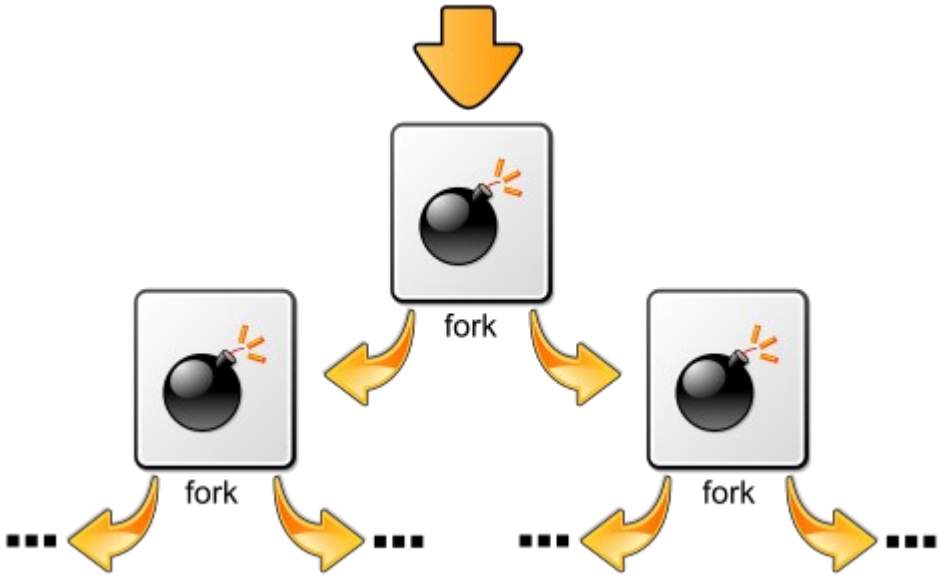
Problemi

- **... system call exit**
- Il SO libera le risorse allocate per il processo. Se il padre del processo terminato attendeva la sua terminazione, **questa gli viene notificata**, insieme ad alcuni dati inerenti la terminazione del figlio
- **Attenzione**: in certi SO i dati sulla terminazione del figlio sono conservati fino a quando non vengono ispezionati dal padre!



Fork bomb / wabbit

- Dicesi wabbit (o fork bomb) un programma che crea un numero smisurato di figli, che a loro volta creano un numero smisurato di figlio che a loro volta ... ecc. ecc.
- Effetto: il numero di processi coesistenti, gestibili da un SO è limitato. Uno wabbit lo riempie rapidamente bloccando, di fatto, il sistema.



http://en.wikipedia.org/wiki/File:Fork_bomb.svg

Terminazione

- **Un processo può causare la terminazione di un altro in modo esplicito, tramite system call, a patto di conoscerne il PID**
- Alcuni possibili motivi:
  - il processo sta usando troppe risorse
  - la sua elaborazione non occorre più
  - il padre è terminato e il SO forza i suoi figli a fare altrettanto (es. SO VMS)

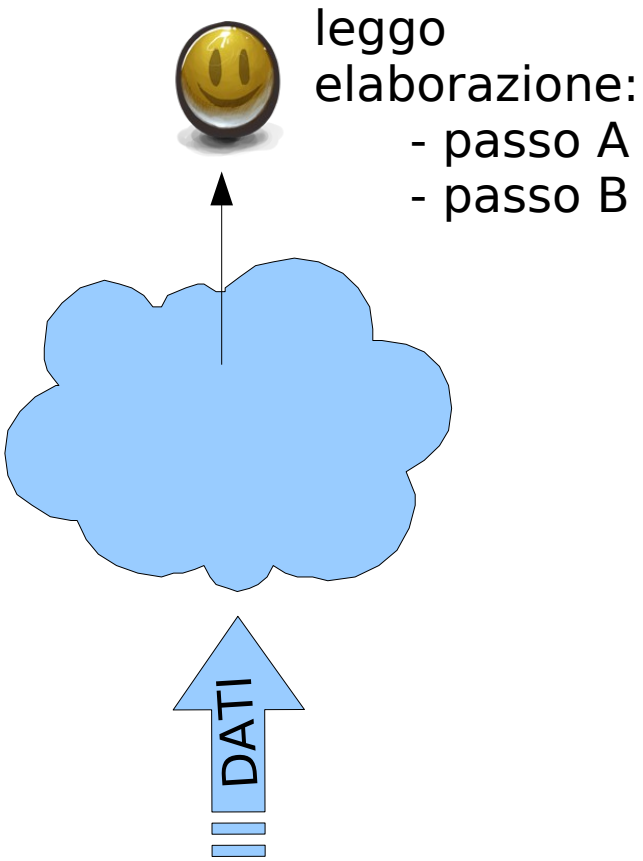
modelli di comunicazione

sezioni 3.4, 3.6.1 e 3.6.2 del libro (VII ed.)

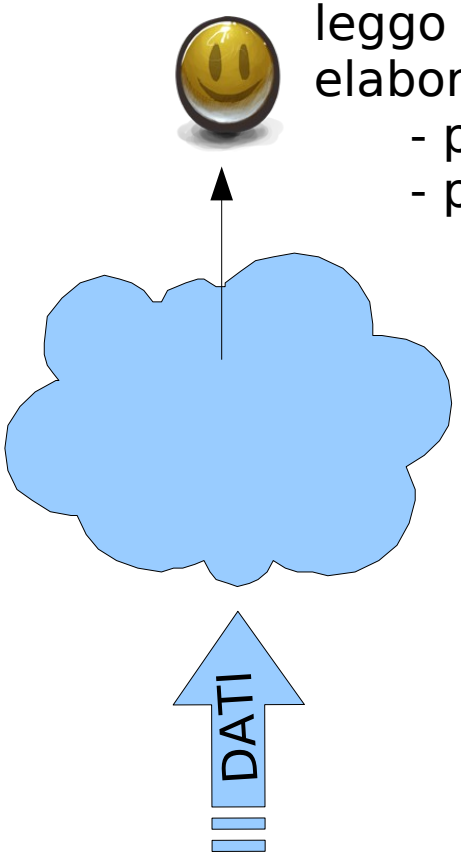
Processi cooperanti

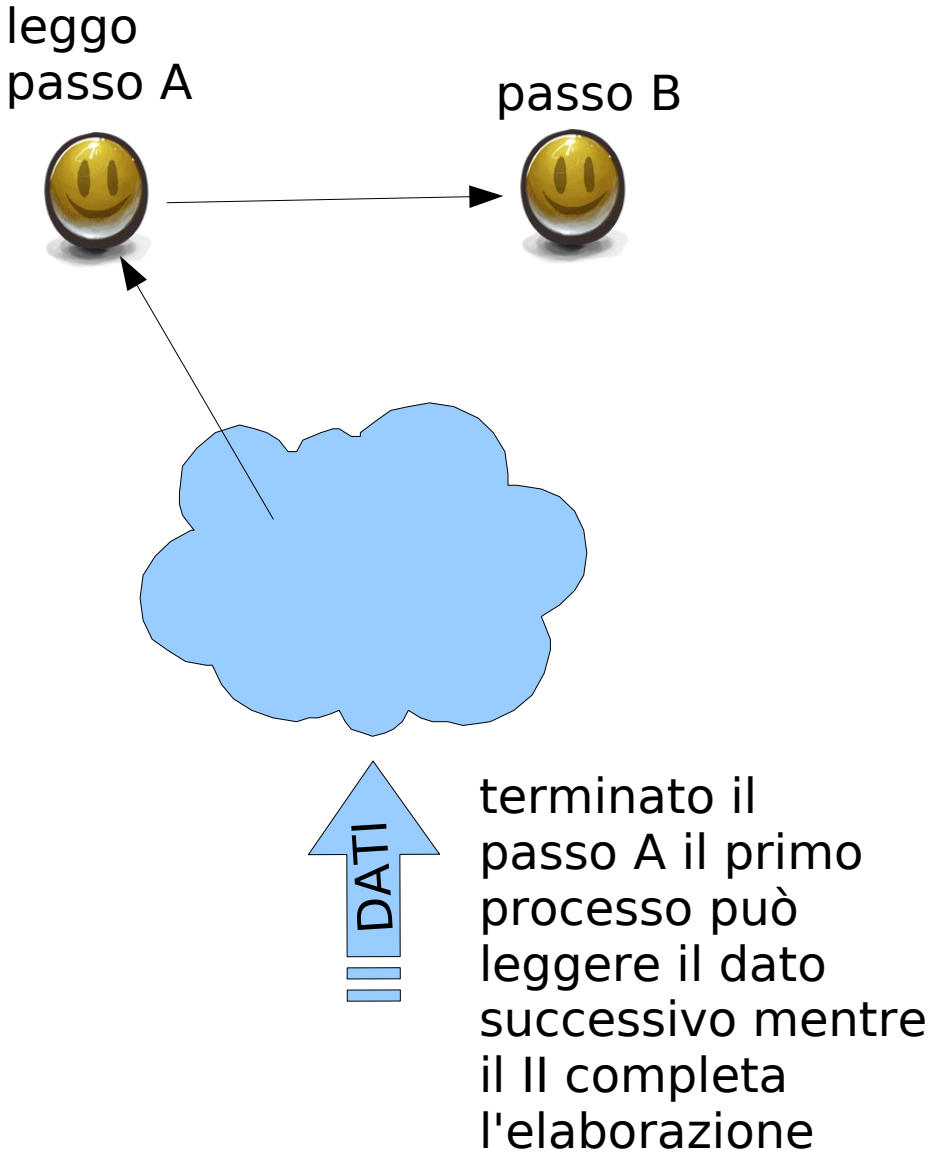
- L'esigenza di far comunicare dei processi nasce quando tali processi cooperano allo svolgimento di un compito
- Un approccio a processi cooperanti è vantaggioso rispetto a un sistema a processi monolitici perché:
  - maggiore efficienza: in molte circostanze è più veloce un'esecuzione a processi paralleli, si sfruttano meglio i tempi morti
  - accesso concorrente a dati condivisi: più utenti/applicativi possono dover usare gli stessi dati, un accesso concorrente migliora i tempi di risposta
- NECESSITÀ: far comunicare / sincronizzare i processi che interagisono tramite meccanismi di "inter-process communication"
- due modelli: **a memoria condivisa** e **a scambio di messaggi**

Processi cooperanti



Processi cooperanti





Produttore / consumatore

- Caso molto frequente: un processo produce dei dati che vengono consumati da un altro processo
- Es. un web server produce pagine HTML consumate da un web browser, un compilatore produce codice assembly, consumato da un assembler

pipeline di processi

come fanno I vari processi a passarsi i dati semi-lavorati?