

# Interprocess Communications

Franco Maria Nardini

# Pipe

```
cat pippo.txt | less
```

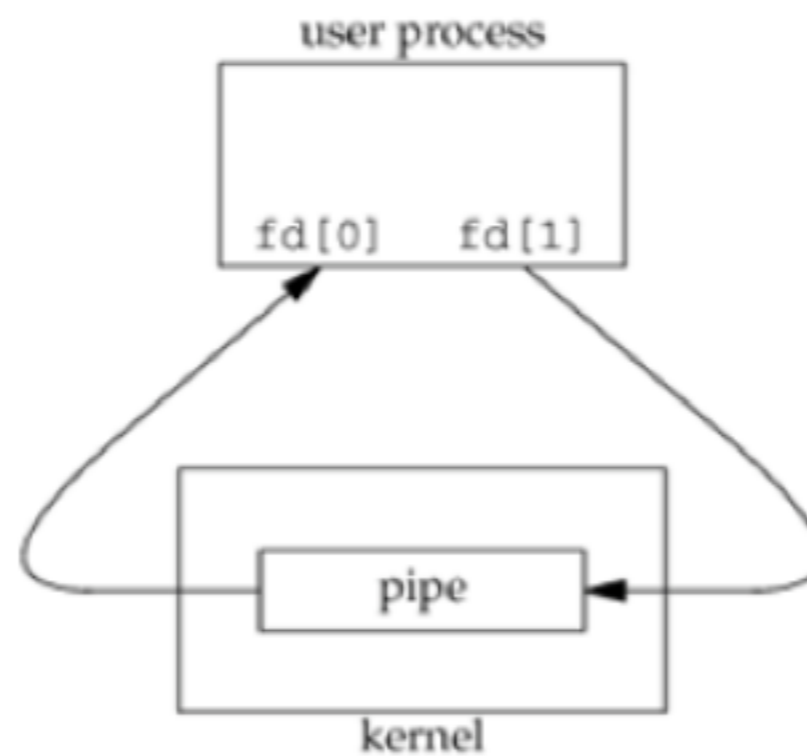
```
sort pluto.txt | uniq -c | less
```

- è il metodo più datato di IPC in UNIX
- half-duplex
  - POSIX.1 consente full-duplex
  - half-duplex sempre per massima portabilità
- può esser usata tra processi che hanno ancestor comune

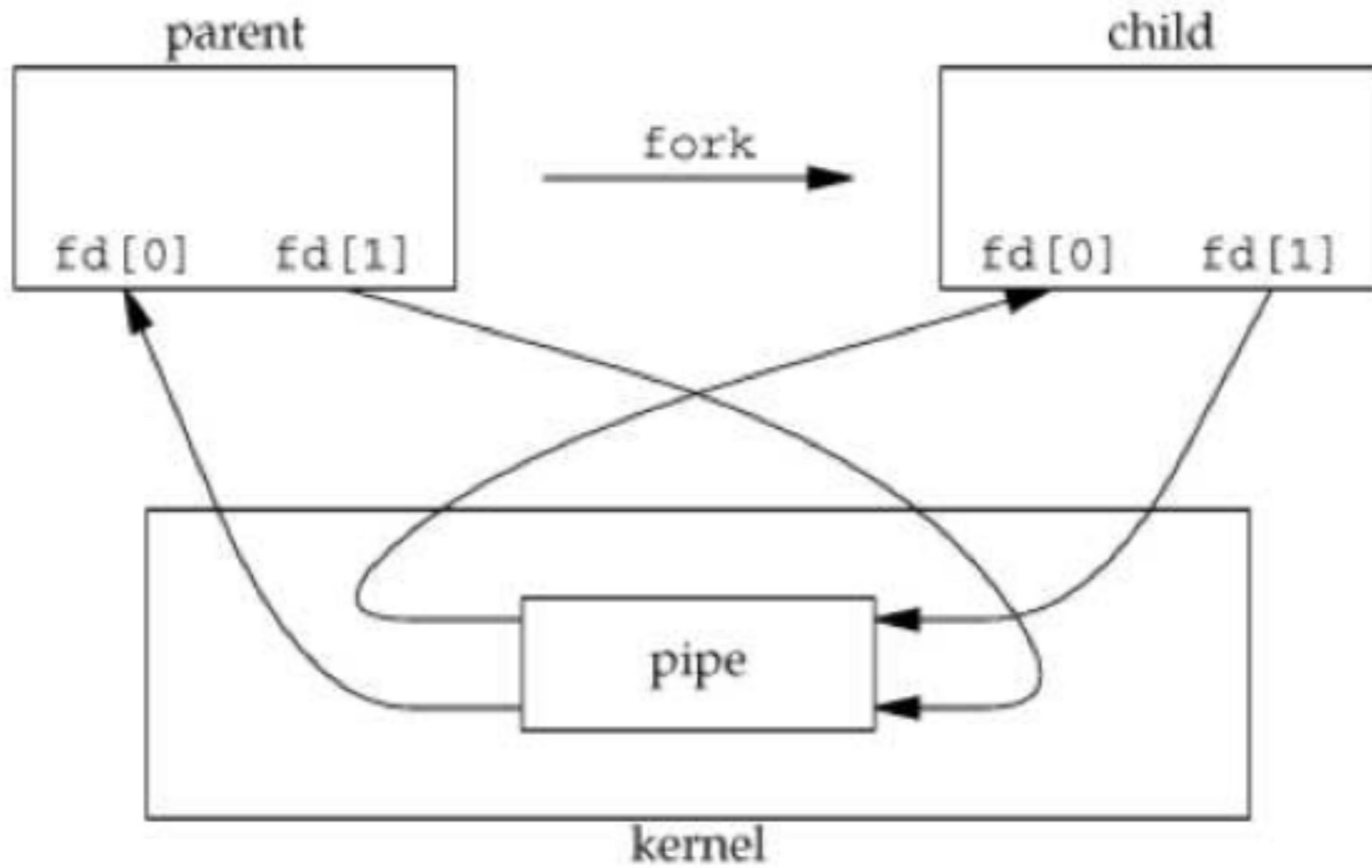
# pipe ( 2 )

```
#include <unistd.h>  
  
int pipe(int filedes[2]);
```

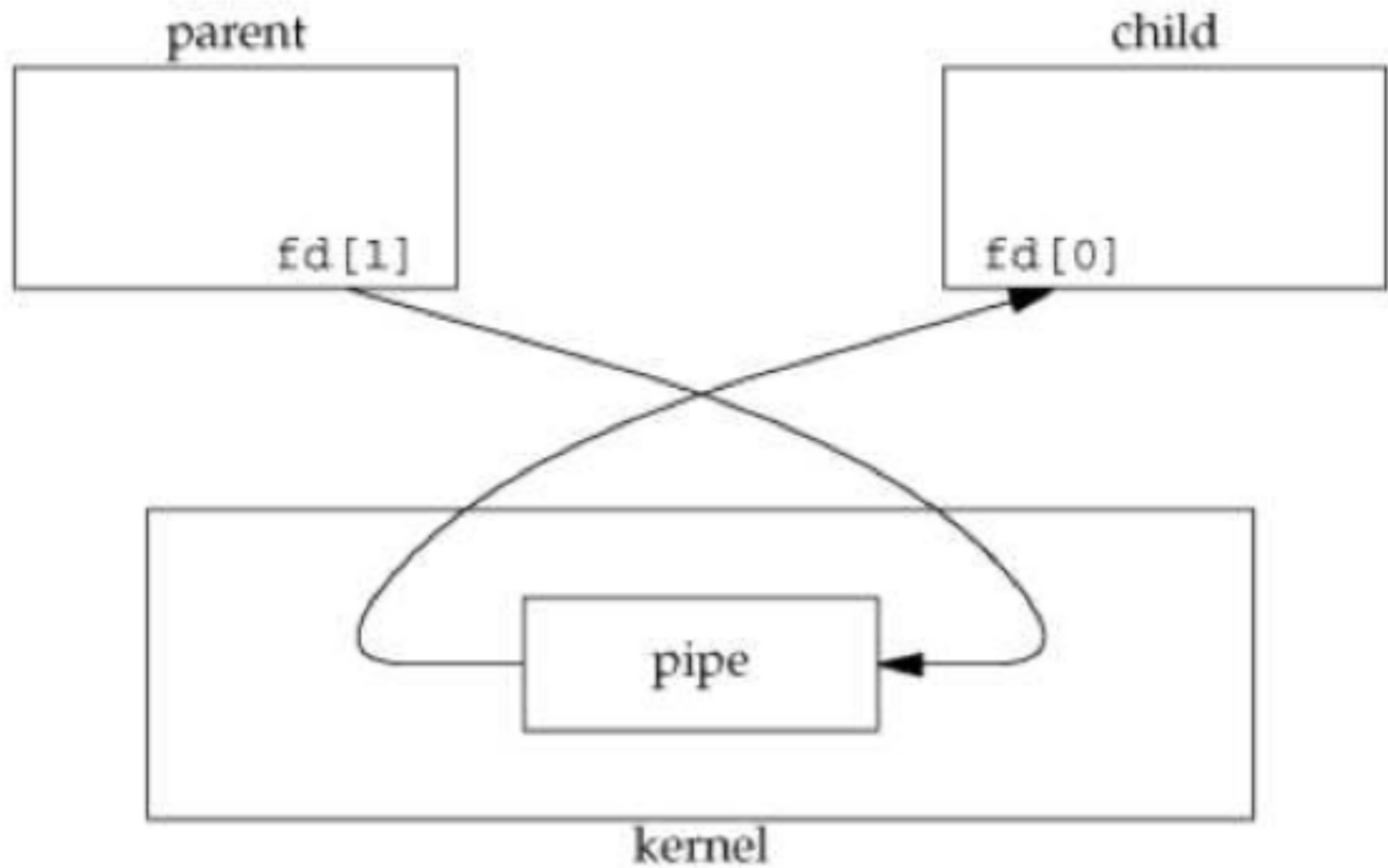
Returns: 0 if OK, -1 otherwise



# Pipe



# Pipe



# Pipe

- Dopo le chiusure:
  - `read(2)` da un pipe il cui `fd[1]` è stato chiuso: ritorna 0 dopo aver letto tutti i dati
  - `write(2)` su un pipe con `fd[0]` chiuso:
    - `write(2)` ritorna -1
- Quando si scrive su pipe, la costante `PIPE_BUF` specifica la dimensione del buffer del kernel associato al pipe

# Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/10/pipe1.c
```

```
$ cc -Wall ./pipe1.c
```

```
$ ./a.out
```

# popen ( 3 )

- La C standard library ci viene in aiuto! :)
- il lavoro “sporco” visto finora:
  - creare il pipe
  - `forkare` il processo
  - chiudere i terminali non usati del pipe
  - eseguire il comando a shell
  - attendere la terminazione



# popen ( 3 )

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

```
FILE *popen(const char *cmd, const char *type);
```

Returns: file pointer if OK, NULL otherwise

```
int pclose(FILE *fp);
```

Returns: termination status *cmd* or -1 on error

- storicamente implementata con pipe half-duplex, ora usa socket o full-duplex pipe (quando disponibile)
- *type*: 'r' oppure 'w' ('r+' per comunicazioni bidirezionali)
- *cmd* passato a `/bin/sh -c`

# `popen(3)`

- `popen(3)`: `fork`, `exec(cmdstring)` e ritorna uno standard I/O pointer



# `pclose(3)`

- `pclose(3)`: chiude lo standard I/O stream, attende la terminazione del comando e ritorna l'exit status della shell
  - se errore di esecuzione della shell: `exit(127)`
- esecuzione da Bourne shell (`/bin/sh -c cmd`)
- espansione di caratteri speciali

```
fp = popen("ls *.c", "r");
```

# Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/10/popen.c
```

```
$ cc -Wall ./popen.c
```

```
$ ./a.out
```

# Hands on! :)

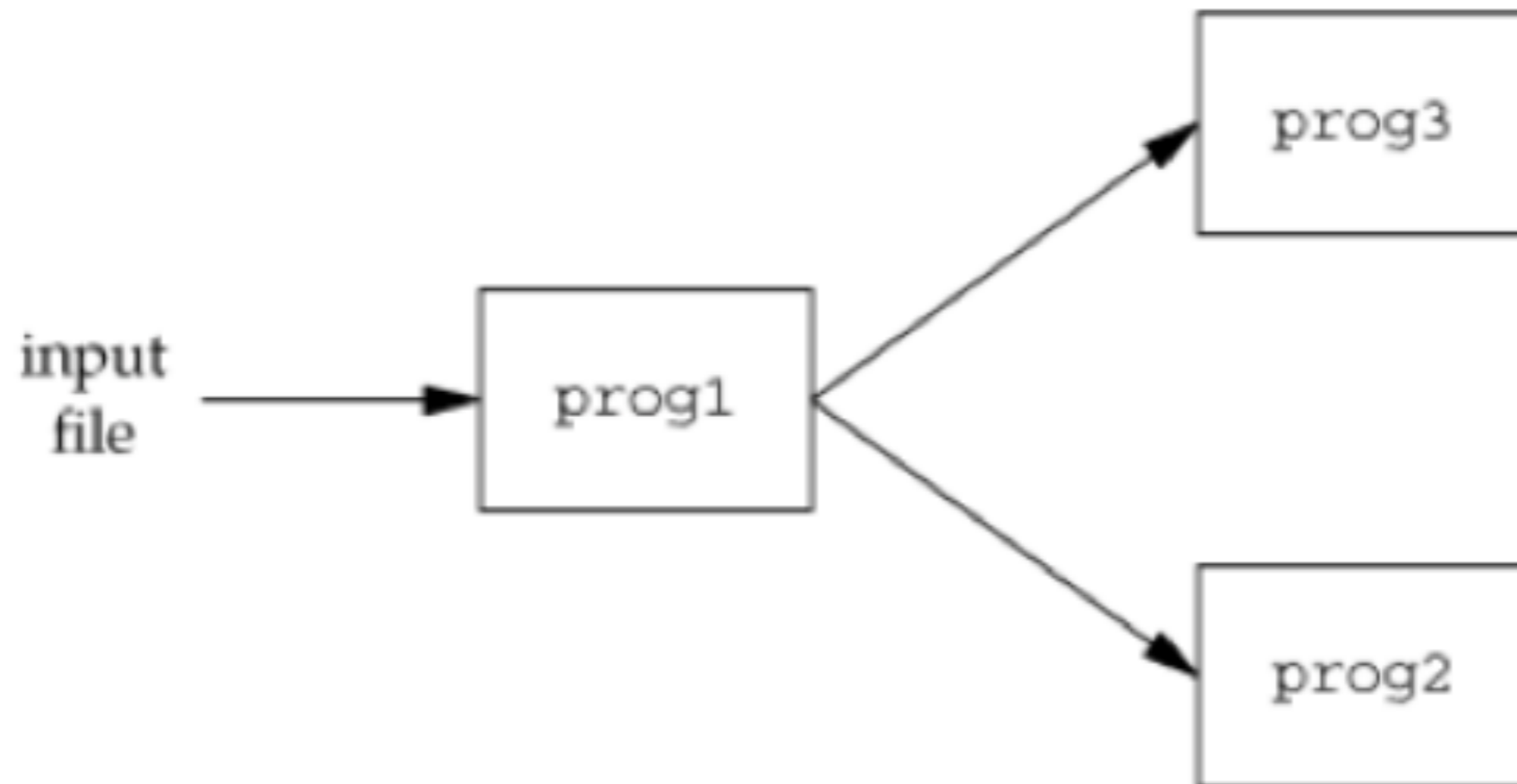
- Prendere il file `primi100numeri.txt` e dividerlo in due:
  - tutte le righe che matchano `"77392"` in `pluto.txt`
  - il resto in `paperino.txt`

# Prima soluzione

```
$ cat primi100numeri.txt | grep "77392" > pluto.txt
```

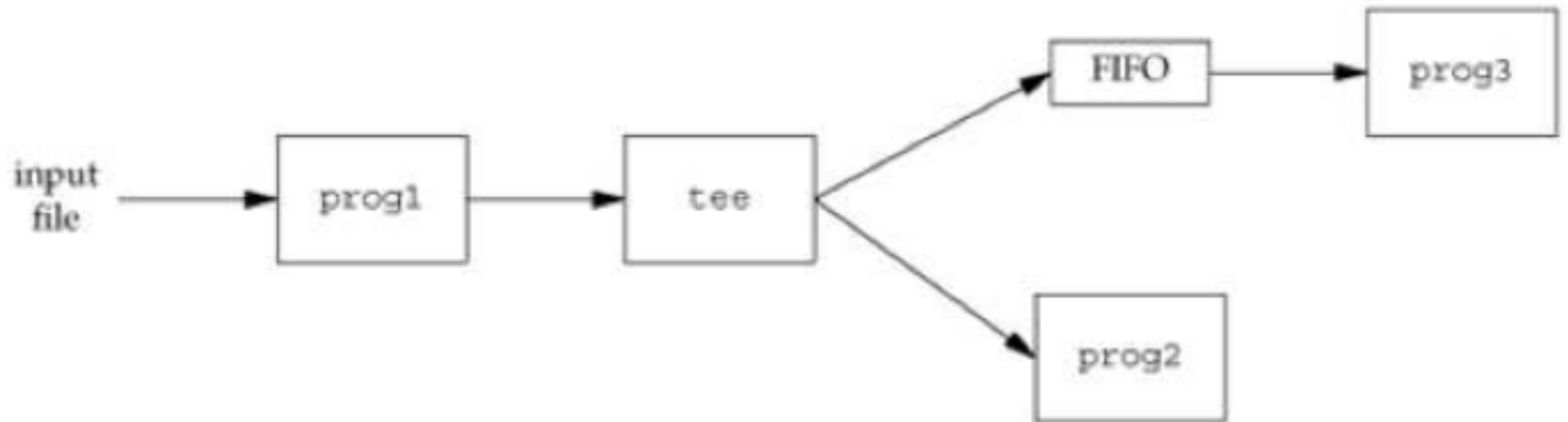
```
$ cat primi100numeri.txt | grep -v "77392" > paperino.txt
```

Si puòò far meglio? :)



# Si puòò far meglio? :)

- si, usando `tee(1)` e `mkfifo(1)`



```
$ mkfifo pippo
```

```
$ grep 77392 pippo > pluto.txt &
```

```
$ cat primi100numeri.txt | tee pippo | grep -v 77392 > paperino.txt
```



# FIFOs

- anche conosciute come “named pipes”
- consentono di far comunicare processi scorrelati tra loro
- è un tipo di file (?)
  - `st_mode` in `stat`, check con `S_ISFIFO`
- usate dalla shell per duplicare contenuti
  - connessioni non-lineari tra processi (?)
  - nessun file temporaneo!
- usate per comunicazioni client-server (rendezvous points)

# XSI IPC

- Tre tipi di IPC introdotti da System V:
  - semafori
  - shared memory
  - code di messaggi
- Comunicazioni tra processi su stesso host
  - Tutte consentono comunicazione *asincrona*

# XSI IPC

- Ogni struttura IPC è identificata mediante un *identificatore*
  - intero non negativo che il kernel usa per riferire la risorsa
  - diversamente dai *file descriptors* (?), non sono piccoli
  - crescono e tornano all'inizio (limite inferiore), quando raggiunto il MAX\_INT
- Serie di comandi e system calls per interagire
  - `msgget(2)`, `semop(2)`
  - `ipcs(1)`

# XSI IPC: Semafori

- Un semaforo è:
  - un contatore per consentire il corretto accesso ad una risorsa condivisa da parte di processi multipli
- Per ottenere la risorsa, un processo deve:
  - testare il semaforo che controlla la risorsa
  - se valore  $> 0$ , decremento il semaforo e uso la risorsa. al termine, incremento di nuovo il valore del semaforo.
  - se valore  $== 0$ , aspetto finché valore  $> 0$
- I semafori sono ottenuti con `semget(2)`, controllati con `semctl(2)`. Le operazioni sono fatte con `semop(2)`.

# Example

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/11/  
sendemo.c
```

```
$ cc -Wall ./sendemo.c
```

```
$ ./a.out (prima shell)
```

```
$ ./a.out (seconda shell)
```

```
$ ipcs -s
```

```
$ ipcrm -s semID (rimuove il semaforo)
```

# XSI IPC: Shared Memory

- Metodo più veloce di IPC
- Consiste nell'accedere ad una memoria condivisa a volte controllata con semafori
- ottenimento di un identificatore di memoria condivisa con `shmget(2)`
- collegamento del segmento condiviso allo spazio degli indirizzi di un processo con `shmat(2)`
- scollegamento con `shmdt(2)`

# Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/11/  
shmdemo.c
```

```
$ cc -Wall ./shmdemo.c
```

```
$ ./a.out Ciao!
```

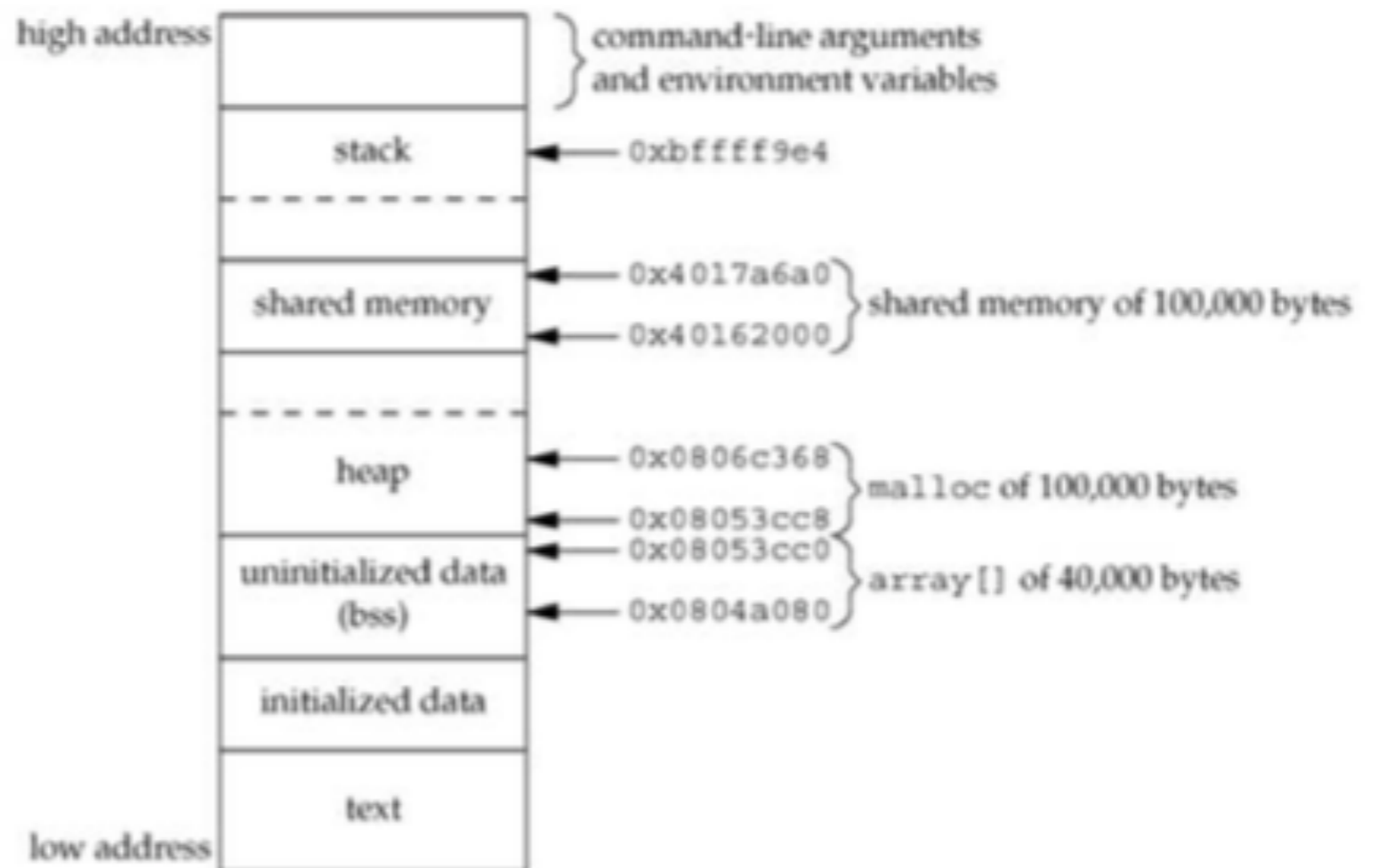
```
$ ipcs
```

# Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/11/memory-layout.c
```

```
$ cc -Wall ./memory-layout.c
```

```
$ ./a.out
```





# XSI IPC: Message Queues

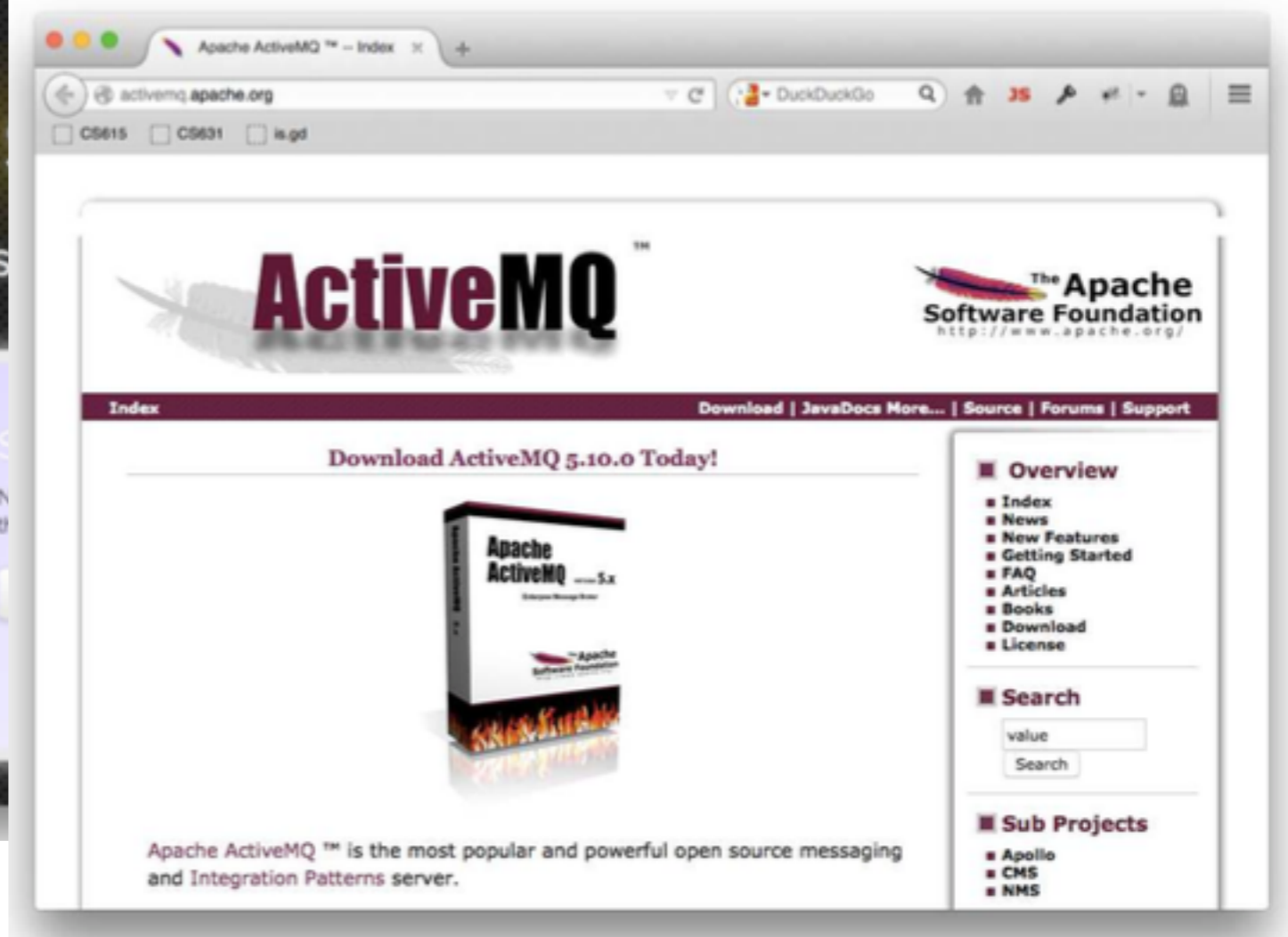
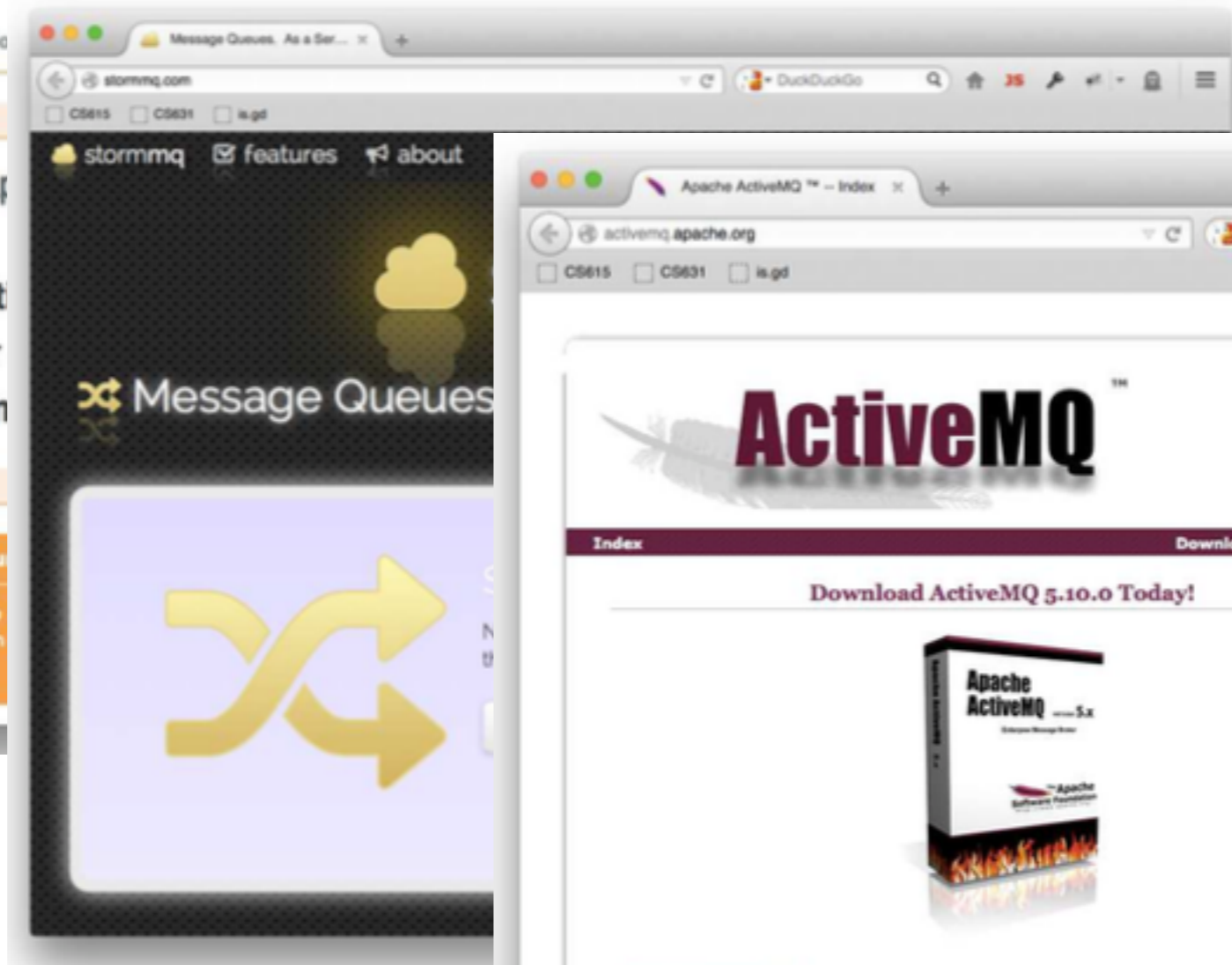
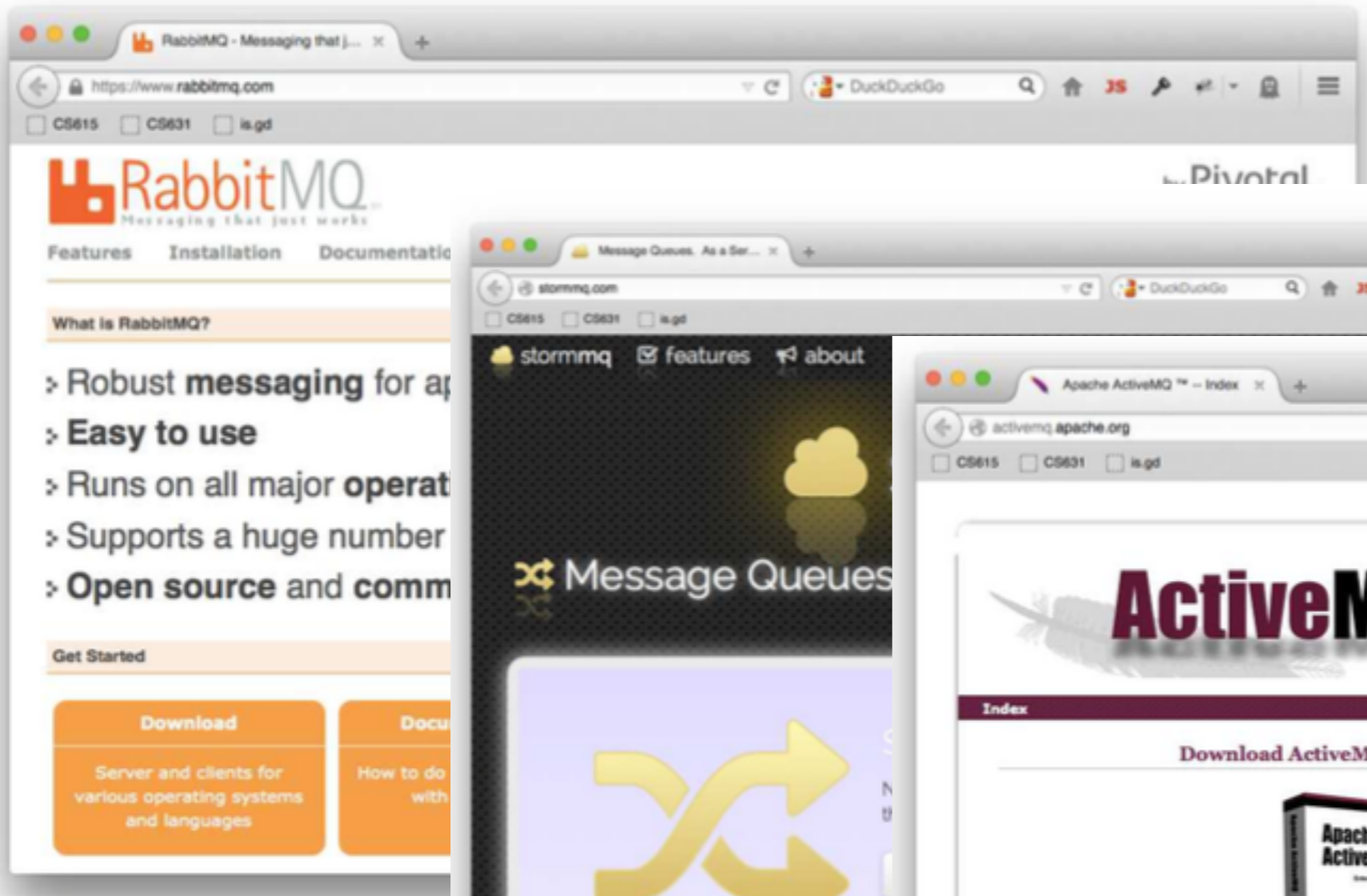
- Liste linkate (?) di messaggi memorizzate dal kernel
- Creare e aprire una coda con `msgget(2)`
- Aggiungere un messaggio a fine coda con `msgsnd(2)`
- Controllo della coda con `msgctl(2)`
- Ricezione messaggi dalla coda `msgrcv(2)`

# XSI IPC: Message Queues

- Il messaggio ha una struttura definita dall'utente

```
struct mymsg {  
    long mtype;          /* message type */  
    char mtext[512];    /* body of message */  
};
```

# Altre Code



# Sockets

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

int socket(int domain, int type, int protocol);
```

Domain	Description
PF_LOCAL	local (previously UNIX) domain protocols
PF_INET	ARPA Internet protocols
PF_INET6	ARPA IPv6 (Internet Protocol version 6) protocols
PF_ARP	RFC 826 Ethernet Address Resolution Protocol
...	...

Type	Description
SOCK_STREAM	sequenced, reliable, two-way connection based byte streams
SOCK_DGRAM	connectionless, unreliable messages of a fixed (typically small) maximum length
SOCK_RAW	access to internal network protocols and interfaces
...	...

# UNIX/LOCAL domain

- si crea un socket usando `socket(2)`
- ci si “attacca” al socket usando `bind(2)`
- `bindare` un nome in dominio UNIX, corrisponde a creare un file sul file system
- entrambe i processi devono conoscere e concordare il nome del socket
- i file servono per il rendezvous, non per il message delivery quando la connessione è stabilita
- il socket deve esser rimosso con `unlink(2)`

# Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/11/  
udgramsend.c
```

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/11/  
udgramread.c
```

```
$ cc -Wall ./udgramsend.c -o send
```

```
$ cc -Wall ./udgramread.c -o read
```

# Esempio

```
1$ ./read
```

```
2$ ls -l
```

```
2$ ./send socket
```

```
1$ (?)
```

# Internet Domain

- utilizzano Internet! :)
- diversamente dai socket visti prima, i nomi di socket Internet non passano dal file system
  - non c'e' bisogno di unlink dopo la chiusura
- indirizzo può essere un qualsiasi indirizzo di rete valido
- porte riservate a root: 1-1023
- richiesta di una specifica porta con `bind(2)`
- determinare la porta usata con `getsocketname()`



# Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/11/  
dgramsend.c
```

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/11/  
dgramread.c
```

```
$ cc -Wall ./dgramsend.c -o send
```

```
$ cc -Wall ./dgramread.c -o read
```

# Esempio

```
1$ ./read
```

```
2$ ./send localhost #porta
```

```
1$ (?)
```