Università degli Studi di Bologna
Scuola di Ingegneria

Corso di
Reti di Calcolatori T

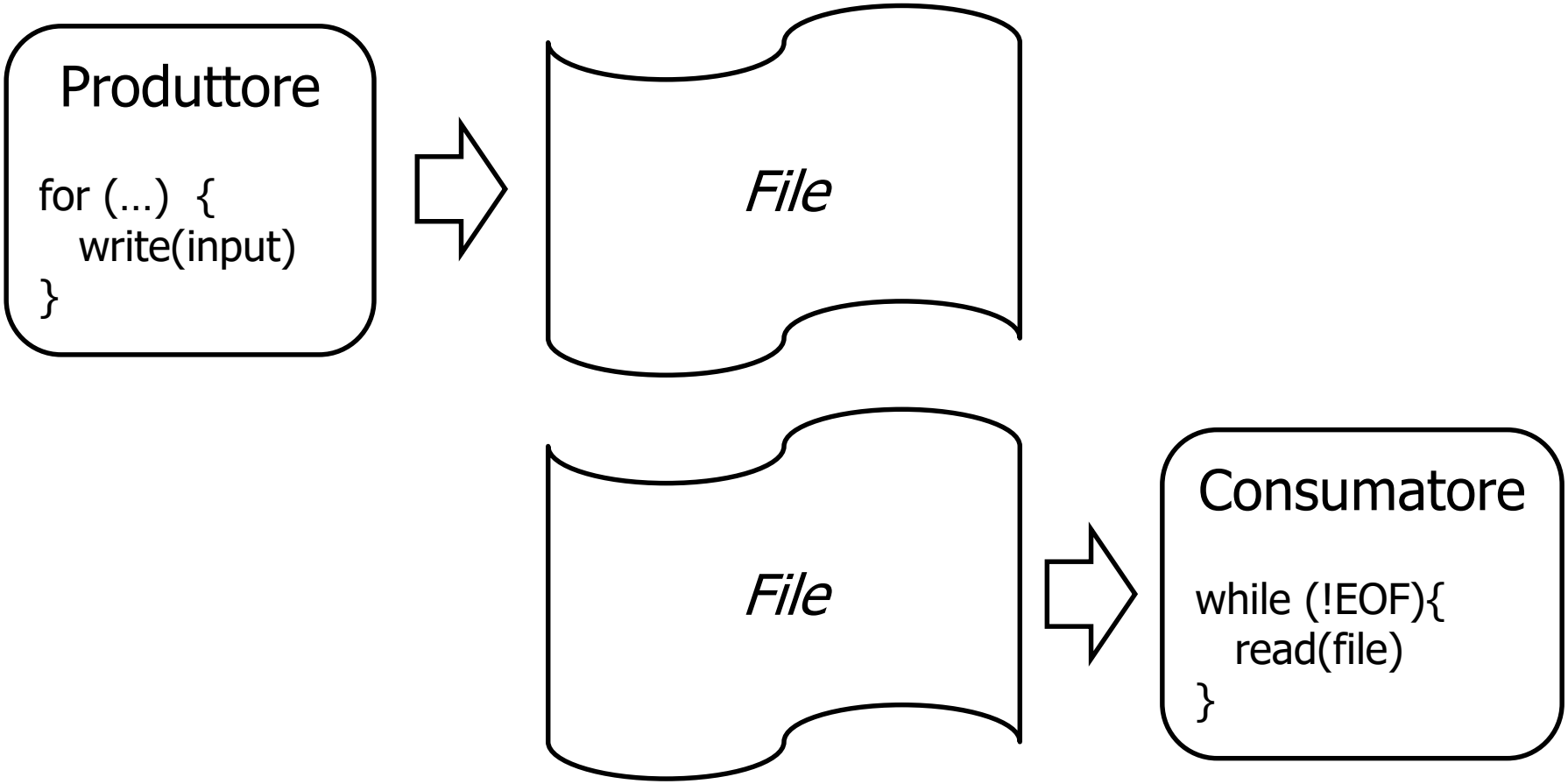
Esercitazione 0 (proposta)
Lettura e Scrittura File in Java e C

Antonio Corradi, Luca Foschini
Michele Solimando, Giuseppe Martuscelli

Anno accademico 2019/2020

Architettura di riferimento

Si consideri lo schema in figura…



Proposta di modifica

Modificare **le soluzioni precedenti** facendo in modo che nel processo **produttore** **non venga chiesto all'utente quante righe scrivere**, ma si legga **fino a quando l'utente immette EOF (end of file)** per terminare l'inserimento (rendendolo cioè **un filtro**). Interfaccia d'uso produttore:

```
>java Produttore fileName.txt
>produttore fileName.txt
```

Modificare anche il processo **consumatore** in modo che si comporti come **un filtro vero e proprio** che esegue una elaborazione tra la lettura e la stampa a video (**N.B.: filtro a carattere!!**). In questo caso, stampa a video il contenuto del file tagliando il carattere (meglio la sequenza di caratteri) passati come argomento.

Interfaccia d'uso consumatore:

```
>java Consumatore  filterprefix  fileName.txt
>consumatore  filterprefix  fileName.txt
```

Proposta (dettagli)

Modificare quindi il processo **consumatore** in modo da farlo diventare un **filtro** che accetta anche la ridirezione in ingresso, e quindi possa prendere il contenuto **sia da input sia da un file passato come argomento** (dipendentemente dal numero di argomenti)

Come nel caso precedente, il consumatore passa tutto il contenuto del file, tagliando il carattere (**meglio la sequenza di caratteri**) passati come argomento. Interfaccia d'uso:

```
>java Consumatore prefixstring < fileName.txt
```

```
>consumatore prefixstring < fileName.txt
```

Si modifichi il programma **consumatore** in modo da consentire l'invocazione nei due modi, senza ridirezione (2 argomenti) e con ridirezione (1 argomento).

Documentazione in C: il man

Manuale: invocato con **`>man read`**

```
READ(2)                          Linux Programmer's Manual                          READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

       If  count is zero, read() returns zero and has no other results.  If count is greater than SSIZE_MAX,
       the result is unspecified.

RETURN VALUE
       On success, the number of bytes read is returned (zero indicates end of file), and the file  position
       is  advanced  by  this number.  It is not an error if this number is smaller than the number of bytes
       requested; this may happen for example because fewer bytes are actually available  right  now  (maybe
       because  we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or
       because read() was interrupted by a signal.  On error, -1 is returned, and  errno  is  set  appropri-
       ately.  In this case it is left unspecified whether the file position (if any) changes.

ERRORS
 Manual page read(2) line 1 (press h for help or q to quit)
```

Memento principali funzioni I/O

open	Apre il file specificato e restituisce il suo file descriptor (fd) Crea una nuova entry nella tabella dei file aperti di sistema (nuovo I/O pointer) Fd è l'indice dell'elemento che rappresenta il file aperto nella tabella dei file aperti del processo (contenuta nella user structure del processo) Possibili diversi flag di apertura, combinabili con OR bit a bit (operatore |)
close	Chiude il file aperto Libera il file descriptor nella tabella dei file aperti del processo Eventualmente elimina elementi dalle tabelle di sistema
read	read(fd, buff, n) legge al più n bytes a partire dalla posizione dell'I/O pointer e li memorizza in buff Restituisce il numero di byte effettivamente letti 0 per end-of-file -1 in caso di errore (perror e errno per sapere quale)
write	write(fd, buff, n) scrive al più n bytes dal buffer buff nel file a partire dalla posizione dell'I/O pointer Restituisce il numero di byte effettivamente scritti o -1 in caso di errore
lseek	lseek(fd, offset, origine) sposta l'I/O pointer di offset posizioni rispetto all'origine. Possibili valori per origine: 0 per inizio del file (SEEK_SET) 1 per posizione corrente (SEEK_CUR) 2 per fine del file (SEEK_END)

Memento principali comandi di sistema: gestione dei processi in UNIX

- Un processo utente in genere viene attivato a partire da un comando (da cui prende il nome). Ad es., dopo aver mandato in esecuzione il comando ‘hw’, verrà visualizzato un processo dal nome ‘hw’.

- ***Tramite ps si può vedere la lista dei processi attivi***

```
msolimando@lab3-linux:~$ ps
PID TTY STAT TIME COMMAND
4837 p2 S 0:00 -bash
6945 p2 S 0:00 hw
6948 p2 R 0:00 ps
```

Memento principali comandi di sistema: terminazione forzata dei processi

- È possibile 'terminare forzatamente' un processo tramite il comando **kill c**he invia un segnale ad un processo.

- Ad esempio:
- **kill –9 <PID>** provoca la terminazione del processo
  - Esempio: **kill –9 6945** → termina il processo **'hw'**

- per conoscere il PID di un determinato processo, si può utilizzare il comando **ps**

Memento principali comandi di sistema: temporizzazione di un programma C

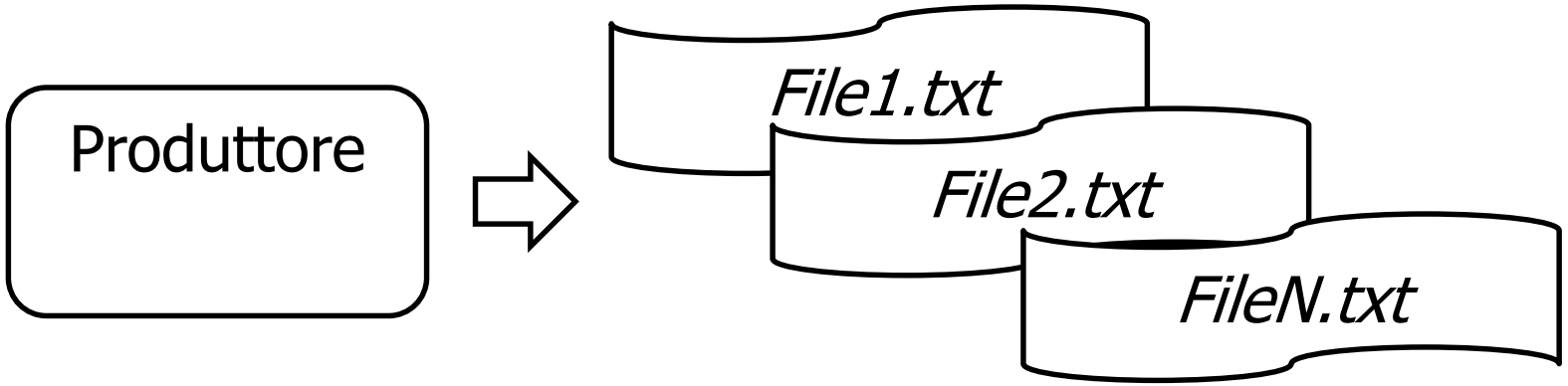
- Tramite la funzione **sleep()** è possibile sospendere temporaneamente un processo.
- **#include <unistd.h>**
  unsigned int sleep(unsigned int seconds);
- Valore di ritorno INT → Zero se è trascorso il tempo richiesto o il numero di secondi rimasti se la chiamata è stata interrotta da un handler in risposta ad un segnale.

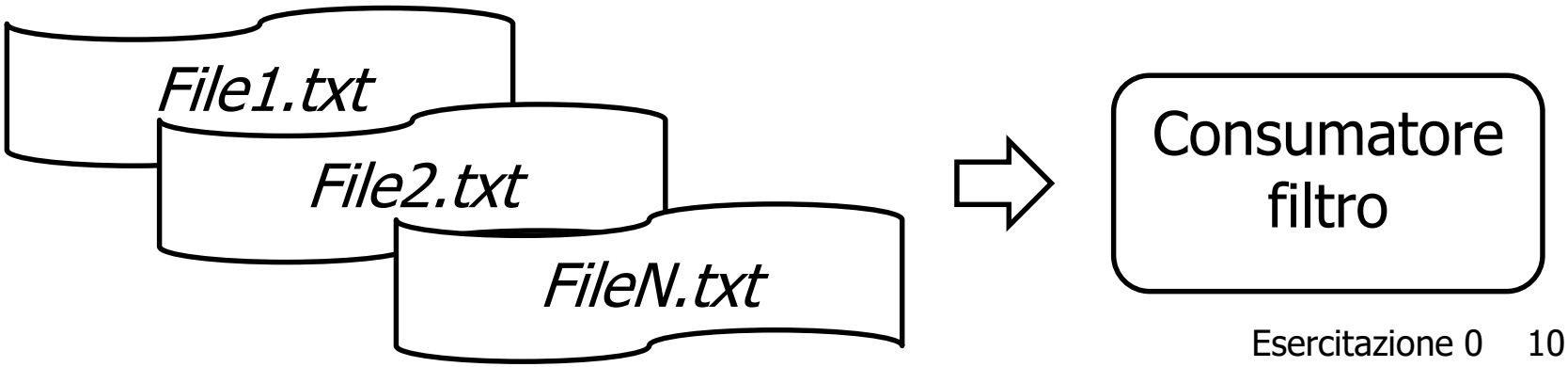
- Per ottenere il tempo di sistema in C si può utilizzare la funzione **gettimeofday** (definita in **sys/time.h**), che sfrutta le strutture:
  - //timeval salva il tempo, sempre dalle 00:00, dell'1/1/1970; ha due campi, uno per i secondi (tv_sec) e uno per i microsecondi (tv_usec)
  - **struct timeval tv;**
  - **struct timezone tz;**
  - **gettimeofday(&tv, &tz);**
  - **tv.tv_sec; tv.tv_usec;** // secondi e microsecondi
  - **tv.tv_usec/1000;** // millisecondi

E in Java? Uso di **System.currentTimeMillis()**

Proposta di estensione: programmi che lavorano su più file

Si vogliono modificare i programmi **produttore** e **consumatore** per renderli programmi che lavorano su più file contemporaneamente. Architettura che si vuole ottenere:





Proposta di estensione: interfacce d'uso

Modificare **le soluzioni precedenti** facendo in modo che il processo **produttore** **possa accettare come argomenti più nomi di file testo**, eliminando la possibilità di ridirezione dell'input.
Interfaccia d'uso produttore:

```
>java Produttore fileName1.txt fileName2.txt … fileNameN.txt
>produttore fileName1.txt fileName2.txt … fileNameN.txt
```

Modificare anche il processo **consumatore** in modo che **possa accettare come argomenti più nomi di file di testo** (senza ridirezione).

Interfaccia d'uso consumatore:

```
>java Consumatore  prefix  fileName1.txt … fileNameN.txt
>consumatore  prefix  fileName1.txt … fileNameN.txt
```

Proposta di estensione: dettaglio produttore

Il **produttore** **non chiede all'utente quante righe scrivere**, ma è un programma sequenziale che legge **fino a quando l'utente immette EOF (end of file)** per terminare l'inserimento (è **un filtro**).

Ciascuna riga letta comincia con un prefisso composto da un intero seguito dal separatore ':'; tale intero indica all'interno di quale file andrà scritto (in append) il contenuto della riga letta ad esempio:

```
>1:Questa riga andra' in fileName1.txt
// in fileName1.txt, che assumiamo essere il primo argomento
>3:Questa riga andra' in fileName3.txt
// in fileName3.txt, che assumiamo essere il terzo argomento
>1:Questa riga andra' in fileName1.txt
```

…

Una volta terminato il ciclo di letture il **produttore** **chiude tutti i file aperti e libera tutte le risorse occupate**.

Proposta di estensione: dettaglio consumatore

Il **consumatore** **agisce in modo concorrente, lanciando diversi processi figli che lavorano in modo indipendente, ciascuno su uno dei file passati come argomenti** **in ordine.**

Il **processo padre** **controlla gli argomenti** e **genera i figli per il processing degli argomenti correttamente ricevuti** (cioè file esistenti e presenti nel direttorio locale), quindi termina.

Ciascun **processo figlio** **è un filtro** che **legge il file fino a EOF (end of file)**. Si noti che gli output generati non devono essere scritti su standard output, ma rispettivamente scritti sui singoli file di testo passati come argomenti all'invocazione (ovviamente cambiandone il contenuto).

A tale scopo, si suggerisce che ciascun figlio crei un **file di appoggio** che sarà popolato col contenuto filtrato e poi copiato sul file di input, (sostituendolo al file esistente ed eliminandolo alla terminazione).