cammini.c

Il codice di inizia con i controlli sugli argomenti che gli vengono passati.
Viene invocata la funzione `atoi` per vedere quanti thread consumatori andranno creati:

```
// Converto il numero di thread consumatori da stringa ad intero
int numeroConsumer = atoi(argv[3]);
```

Successivamente faccio un controllo sul numero di consumatori.

Inizia la parte della dichiarazione del set di segnali che voglio gestire.
Definisco inizialmente tramite `sigset_t setSignal` il set di segnali, inizialmente vuoto, che vorrò gestire. Questo dovrà essere passato al thread che gestisce i segnali.
Ora pulisco la memoria associata a quel set (al momento in cui viene allocata la memoria per quel set può succedere che sia "sporca" cioè che ci siano dei bit spazzatura, per questo motivo prima di aggiungere i segnali che voglio gestire la pulisco) tramite la `sigemptyset` :

```
sigemptyset(&setSignal);
```

e successivamente inserisco in quel set i segnali che voglio gestire ovvero `SIGINT`:

```
sigaddset(&setSignal,SIGINT);
```

A questo punto creo la maschera per i segnali `SIGINT` tramite la funzione `ptherad_sigmask` :

```
int errSignalMask = pthread_sigmask(SIG_BLOCK,&setSignal,NULL);
```

Controllo il suo valore di ritorno (se != da zero allora c'è stato un errore).

A questo punto tutto è andato bene quindi procedo con la dichiarazione e la creazione del thread per la gestione dei senali di `SIGINT`.

Dichiaro due variabili atomiche di tipo `aromic_int` :

```
atomic_int stato_pipe = ATOMIC_VAR_INIT(0);
atomic_int stato_terminazione = ATOMIC_VAR_INIT(0);
```

queste serviranno per capire in quel stato mi trovo della pipe, cioè se è stata aperta o chiusa e anche in quale stato mi trovo, cioè se devo terminare oppure no.

Dichiaro la `struct` da passare al thread gestore dei segnali e gli passo i valori appena assegnati alle due variabili atomiche, il thread gestore dei segnali prende per argomento una `struct` di tipo `datiThreadSignal` che è definita come segue:

```
// Struttura da passare al thread di gestione di segnali
typedef struct{
    atomic_int* attiva_lettura_pipe;
    atomic_int* stato_terminazione;
} datiThreadSignal;
```

Dichiaro il thread `signalThread` tramite la funzione `pthread_t` ed infine creo il suddetto thread tramite la funzione `xpthread_create`:

```
xpthread_create(&signalThread,NULL,threadSignalBody,&datiTS,QUI);
```

Se è andato tutto bene fino a questo punto è ora della lettura del file `nomi.txt`.

Dichiaro un puntatore di tipo `FILE *`, al file aperto in sola lettura tramite la funzione `xfopen`:

```
// Apertura file nomi.txt per lettura
FILE *fileNomi = xfopen(argv[1],"r",QUI);
```

Dichiaro la capacità del mio array che conterrà le struct degli attori presenti nel grafo.
Le `struct attore` sono definite nella seguente maniera:

```
// Stuttura per la difinizione di ogni nodo del grafo
// (cioè per la definizione di ogni attore)
typedef struct {
    int codice;     // Codice univo identificativo
    char *nome;
    int anno;
    int numcop;     // Dimensione array dei coprotagonisti
    int *cop;       // Puntatore all'array di coprotagonisti
} attore;
```

Tale array di strutture di `attore` lo ottengo dalla chiamata alla funzione `inserimentoAttori`:

```
// Creazione array di attori
attore *arrayGrafo = inserimentoAttori(fileNomi, &capacity);
```

che prende per argomento il riferimento alla capacità dichiarata precedentemente e il puntatore al file aperto in lettura in precedenza (per avere maggiori informazioni su `inserimentoAttori` vedere la descrizione di `functionality.c`).

Ottenuto l'array di attori, significa che ho terminato di leggere il file nomi e quindi chiudo il file tramite la chiamata alla `fclose(..)`, e controllo il suo valore di ritorno (se equivale ad `EOF` significa che si è verificato un errore).

A questo punto ho terminato la lettura del file `nomi.txt`, allora passo alla lettura del file `grafo.txt`.

Dichiaro un puntatore di tipo `FILE *`, al file aperto in sola lettura tramite la funzione `xfopen`:

```
// Apertura file grafo.txt in lettura
FILE *fileGrafo = xfopen(argv[2],"r",QUI);
```

Successivamente invoco la funzione `completamentoInserimentoAttori`:

```
completamentoInserimentoAttori(fileGrafo,arrayGrafo,&capacity,numeroConsumer);
```

questa funzione ha il compito di implementare il paradigma produttore-consumatore in modo tale da inserire, per ogni attore, tutti i codici identificativi degli attori che hanno lavorato almeno una volta con esso, nell'array `numcop` presente nella `struct` (per avere maggiori informazioni su `completamentoInserimentoAttori` vedere la descrizione di `functionality.c`).

Finita l'esecuzione di questa funzione significa che il file `grafo.txt` è stato letto del tutto, quindi chiudo il file tramite la chiamata alla `fclose(..)`, e controllo il suo valore di ritorno (se equivale ad `EOF` significa che si è verificato un errore).

A questo punto è il momento della dichiarazione e creazione della pipe.
Tramite la funzione `mkfifo` creo la una pipe FIFO (anche detta pipe nominativa):

```
// Creo la pipe FIFO (speciale tipo di pipe che permette la comunicazione
// tra processo non correlati)
int errPipe = mkfifo("./cammini.pipe",0666);
```

prende per argomento, il nome richiesto dal testo, ed i permessi (in base ottale) che la pipe avrà (`0666` indica che i tre gruppo che costituiscono i permessi ovvero "User", "Group" e "Others" abbiano tutti i permessi sia in lettura che in scrittura).

Controllo dunque il valore di ritorno della creazione della pipe (se è 0 allora la pipe è stata creata con successo, se è `EEXIST` allora la pipe era già stata creata, qualsiasi altro valore ritorni termina l'esecuzione).

Apro a questo punto la pipe e mi salvo il file descriptor:

```
// Apro la pipe in sola lettura (fd sta per file descriptor)
int fd = open("./cammini.pipe",O_RDONLY);
```

la apro in sola lettura perché il compito di scrivere i dati nella pipe è assegnato al programma `cammini.py`, dove mi invia tramite la pipe i valori dei codici degli attori di cui calcolare il cammino minimo.

Aperta la pipe avverto il gestore dei segnali che, da questo momento in poi se arriva un segnale di `SIGINT` deve avviare la procedura di terminazione. Lo faccio settando la variabile atomica `attiva_lettura_pipe` ad 1:

```
atomic_store(datiTS.attiva_lettura_pipe,1);
```

Fatto questo viene chiamata la funzione `camminiMinimiThreadCreate` che si occupa della creazione degli `ABR`, del calcolo dei cammini minimi e della creazione dei file `souce.dest` in cui ci sono tutti gli attori che collegano `souce` a `dest`.

```
camminiMinimiThreadCreate(fd,&stato_terminazione,arrayGrafo,capacity,&signalThread
);
```

(ARGOMENTI ??)

A questo punto se siamo arrivati fino a questo punto significa che è terminato in modo "naturale" cioè senza ricevere segnali.
Quindi si procede a "killare" il thread gestore dei segnali tramite l a `pthread_kill`:

```
// Informiamo il thread gestore dei segnali che il programma è terminato e quindi
// può terminare pure lui
pthread_kill(signalThread, SIGINT);
```

ed infine chiamo la funzione `signalDeallocation`

```
signalDeallocation(arrayGrafo,capacity,&signalThread);
```

che provvede alla de-allocazione di tutto ciò che è rimasto in memoria (per avere maggiori informazioni su `signalDeallocation` vedere la descrizione di `functionality.c`).