Sistemes Operatius 2 - Pràctica 4

Taras Yarema

11 de desembre del 2019

Processos fills

a) Si tot està en la memòria compartida, l'accés a les dades no serà mai un problema per als processos fills. Tot i així, s'haurà d'implementar, per exemple, un semàfor per a cada estructura que estigui compartida i pugui ser escrita/llegida per els processos.

b) En aquest cas, entenc que l'arbre està compartit entre els processos a la secció crítica. Com en el cas anterior, si no hi ha una un semàfor en l'estructura dels nodes de l'arbre, no es pot assegurar la correcte escriptura/lectura de l'arbre per part dels fills.

c) És viable, però *extremadament* poc òptima. Quan es processen moltes paraules, el fet de bloquejar tot l'arbre pot implicar que el procés tardi més temps. Per altra banda, es veritat que no tindríem *race conditions*.

d) És la opció que he desenvolupat. En aquest cas és idoni, ja que podem control·lar l'accés d'escriptura de tots els nodes individualment. D'aquesta manera, evitem les *race conditions*.

Solució plantejada

Fitxer `red-black-tree.h`

Per a poder utilitzar semàfors a C, he hagut d'afegir un camp més a l'estructura dels nodes.

```
typedef struct node_data_ {
    char *key;
    int num_times;
    sem_t sem;
} node_data;
```

Fitxer `red-black-tree.c`

Funció `void free_node_data(node_data *)`

S'ha modificat afegint `sem_close(&data->sem)` per a tancar el semàfor del node.

Fitxer `red-black-tree.h`

Anàlogament al cas dels nodes, per a poder processar els arxius entre diferents fills, s'ha implementat la següent estructura:

```c
typedef struct file_name_{
    char key[MAX_CHARS];
    sem_t sem;
    int done;
} s_file_name;
```

Amb aquesta, els processos saben si poden escriure en l'arxiu amb nom `key`, i si aquest ja ha estat processat.

Fitxer `tree-to-mmap.c`

S'han modificat les funcions per serialitzar i deserialitzar l'arbre per tenir en compte el semàfor de l'estructura del node.

Fitxer `dbfnames-mmap.c`

Modificacions de les funcions originals (sense la `_` inicial) per a soportar l'estructura de `s_file_name`.

Fitxer `process-mmap.c`

Funció `void process_list_files_sem(char *, char *, int, int)`

Funció base per a que cada fill pugui efectua el procés del arxiu. Es passen els punters a la memòria compartida tant dels arxius de la bases de dades com de l'arbre. També rep l'`id` del procés fill i un `offset`.

Inicialment, havia plantejat anar modificant les dades de l'arbre directament a la memòria compartida. No era viable, ja que havies de cercar linealment [1]. Per aquesta raó, cada cop que s'entra es transformen les dades de la memòria compartida en un `rb_tree`. D'aquesta manera, totes les operacions que es realitzen són les mateixes que en el cas de la pràctica 3.

Cal notar que aquesta transformació només es realitza un cop per cada procés fill. Un cop l'ha feta, recorre la memòria que fa referència als fitxers de la base de dades i va actualitzant els que estan disponibles i no s'han processat anteriorment.

Això es realitza utilitzant les funcions `sem_wait`, per a bloquejar/esperar, i `sem_post` per a desbloquejar un semàfor. En el cas del processament dels fitxers, també utilitzo `sem_getvalue` per a saber si un fitxer està sent processat. D'aquesta manera es pot anar saltant al següent fins que es trobi un de disponible, o bé s'acabin.

Fitxer `main.c`

S'ha modificat el cas en que l'usuari escull la primera opció. El procés pare crea `MAX_FORKS` processos fills, els deixa processant els arxius de la base de dades i espera a que tots acabin.

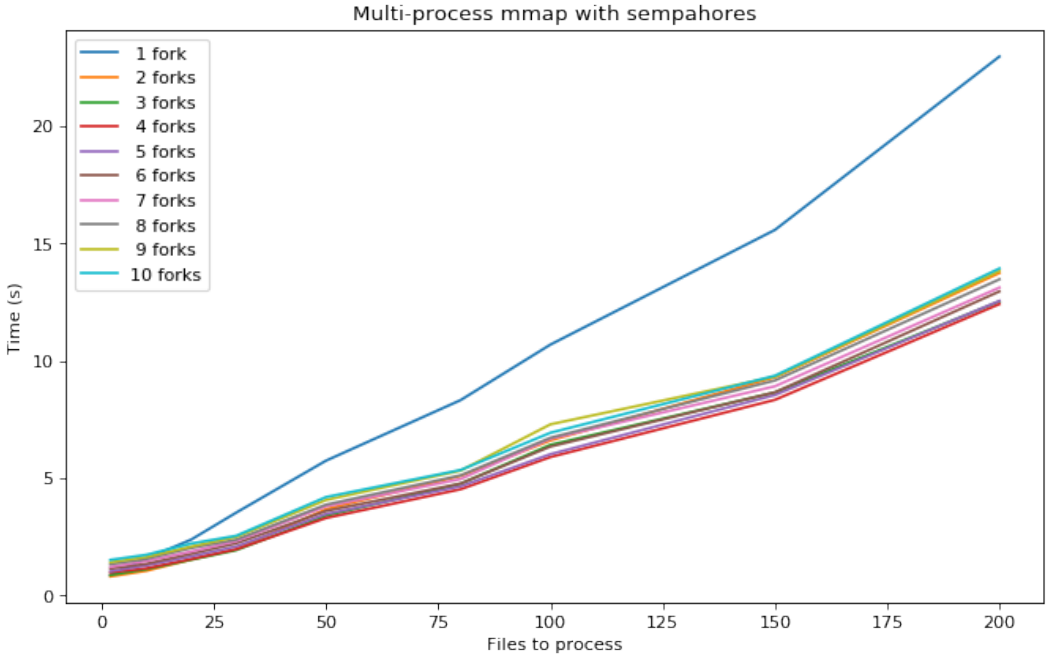
[1] No es pot assegurar que la posició dels nodes fos en un ordre concret. L'algoritme recursiu de cerce, a `rb_tree`, té una complexitat de $O(\log n)$. Per al diccionari donat, al voltant de 300000 paraules, aquesta està acotada a 15 iteracions. Comparat aquesta complexitat amb la de l'algoritme lineal sobre la memòria compartida ($O(n)$), era inviable d'escalar.

Benchmarks

Si s'executa la comanda `make test` a la terminal, es generarà un binàri `test`. Aquest accepta els següents paràmetres `./test <number_forks> <db_list_file>`.

És una simplificació del codi modificat en la funció `main` d'aquesta pràctica, però només amb la part de multi.processament. Els tests han estat realitzant amb l'ajut de *Jupyter Notebook*, es pot trobar l'arxiu en el directori de la documentació.

Com podem observar, l'ús de diversos processos ajuda a que la inicialització de l'arbre sigui fins el doble de ràpida. Concretament, un $50,04\%$ més ràpida en el cas de tenir una base de dades amb 200 fitxers [2] [3].



He vist que, en general, augmentar el número de *forks* no sempre és millor. Per a bases de dades petites, amb 2 fills n'hi ha prou. Un cop treballes amb més, per exemple més de 50, 4 processos és la configuració òptima [4].

[2] Comparant l'execució amb 4 processos *vs* 1 procés fills.
[3] Especificació del ordinador: Intel i7-4510U (4) @ 3.100GHz amb 12GB de RAM.
[4] En el cas del meu processador