Sistemes Operatius II - Pràctica IV

Gabriel Lluís i Pau Soler

11 de Desembre de 2019

1 Introducció

En aquesta pràctica hem d'implementar les següents funcionalitats:

1. Crear un arbre a partir de dos fitxers, un *diccionari* i una *llista de documents de Gutenberg* mijançant un *mmap* amb les funcions dels fitxers donats: *tree-to-map.c* i *dbfnames-mmap.c*.
2. Crear un arbre tal com en el punt anterior, però amb un procés pare que mapi l'arbre buit a memòria i un fill que l'ompli.
3. Crear un arbre tal com en el punt anterior, però amb mútilples fills.

No sabíem si havíem d'entregar només l'útlim apartat havent fet els dos primers prèviament, així que hem modificat el menú amb dues opcions més per poder tenir accés a les tres implementacions diferents. Dit això, els següents apartats corresponen a les opcions 1, 2 i 3 del menú respectivament. La resta d'opcions no es tractaran en aquesta memòria ja que no han sofert canvis respecte l'anterior pràctica.

2 Mmaping

Aquesta secció correspon a la opció 1 del menú i el mètode *practica4()*.

Important *tree-to-map.h* i *dbfnames-mmap.h* tenim accés a les funcions *serialize_node_data_to_mmap(tree)*, que retorna el char d'on comença el mmap a memòria, i *dbfnames_to_mmap(data)*, que mapa els fitxers de llista a memòria compartida. Un cop tot mapat, accedim a cada fitxer amb *get_dbfname_from_mmap(mmap, index)* amb l'index corresponent.

Per trobar aquest índex, prèviament obrim el fitxer i llegim la primera línia amb un *fgets* extraient el primer nombre d'aquest. Però perque el mètode *dbfnames_to_mmap (data)* funcioni correctament, hem de recol·locar el punter del fitxer a la primera posició d'aquest (*fseek(data, 0, SEEK_SET)*). Llavors, amb un iterador anem accedint i processant tots els fitxers.

És important no oblidar-se d'alliberar els mmap's amb els mètodes pertinents també proporcionats (*dbfnames_munmap(mmap)* i *deserialize_node_data_from_mmap*

(tree, mmap)).

3 Un Pare i un Fill

Aquesta secció correspon a la opció 2 del menú i el mètode *crear_arbre_fill()*.

Per implementar aquesta funcionalitat ens hem inspirat fortament en la pràctica 3 de Sistemes Operatius 1.

Cada procés té les següents funcionalitats:

- **Pare:** No té mètode propi (està dins de *crear_arbre_fill()*). La única funció que té és serialitzar els nodes, esperar i deserialitzar-los.
- **Fill:** Dins del mètode *child()*. Les funcions que ha de fer són: trobar el nombre de fitxers a recórrer, mapar-los a memòria, accedir a ells i escriure les paraules a l'arbre.

El funcionament és com l'apartat anterior: es mapa el diccionari a memòria i es crea el procés fill. Mentres el pare espera amb *pause()* el fill va recorrent tots els fitxers i introduint-los a l'arbre. Quan el fill acaba envia un senyal a pare conforme ja ha acabat i pare deserialitza l'arbre.

Som conscients que també es podria haver implementat amb un *wait(NULL)* i que el funcionament hauria estat el mateix, però ens sembla més segur usar senyals.

4 Un Pare i Múltiples Fills

Aquesta secció correspon a la opció 3 del menú i el mètode *crear_arbre_fills()*.

Cada procés té les següents funcionalitats:

- **Pare:** No té mètode propi (està dins de *crear_arbre_fill()*). Les funcions que té són: serialitzar els nodes, mapar els fitxers, esperar a tenir els resultats, desmapar els fitxers i deserialitzar l'arbre.
- **Fills:** Han de processar tots els fitxers de text i introduir-los a l'arbre. Ens hem d'assegurar que no es produeixin *race conditions*, ja que si passa no tindrem resultats correctes a l'arbre. Per evitar-les farem servir semàfors.

4.1 Memòria Compartida

Independentment de com haguem implementat l'escriptura a l'arbre, necessitem una zona de memòria on puguin accedir tots els processos per a controlar els semàfors i quins fitxers s'estan processant ja. Per tant, hem creat una estructura *shared_mem* que conté dos semàfors:

- **clau_tree**: Controla que més d'un procés no estigui escrivint a l'arbre.
- **clau_counter:** Controla que dos processos no augmentin a la vegada counts, variable que diu quin fitxer toca llegir.

Per poder-hi accedir en tot moment el que fem és un mmap d'aquesta memòria compartida, així tots els fills hi tenen accés i no variables noves. Creiem que és millor això que anar fent variables globals, ja que aquesta opció és bastant descontrol.

Hem hagut de modificar el Makefile per compliar els semàfors. Hem hagut d'afegir *-lpthread*, si no no compilaria.

4.2 Fills

Les preguntes més importants d'aquest punt són: **quants processos creem** i **com ho fem**.

El nombre de processos fills no pot ser arbitrari, depen de l'ordinador en què s'estigui executant el codi. Si el teu ordinador té 8 cores i poses 16 fills no faràs que el teu programa vagi més ràpid, sinó que faràs un coll d'ampolla. Aquesta explicació descarta que es facin tants fills com fitxers a processar.

Una decisió seria posar per defecte fer 2 processos, ja que avui en dia trobar un ordinador amb menys de dos nuclis al processador és força rar, però hem preferit usar la funció *getn_procs()* de *sys/sysinfo.h*. Aquesta funció retorna un int amb el nombre de processadors que té la teva màquina. Sabent aquest nombre, hem de posar un *fork()* dins d'un for amb el nombre que hem trobat i allà implementem les funcionalitats que necessitem.

Com a nota final, ens hem d'assegurar que pare s'espera a que tots els fills acabin, ja que com està feta la nostra primera implementació, el pare (amb *wait(NULL)*) no esperava a que tots els fills haguessin acabat. Mitjançant la línia *while((wait(status)) > 0)* ens assegurem que el pare no seguirà fins que tots els fills hagin acabat de processar el que estiguin fent.

4.3 Optimització

La primera opció que se'ns proposa, fer una secció crítica en què estigui inclòs agafar el fitxer a processar, llegir i extreure les paraules, i actualitzar les dades dels nodes, és correcta, de fet, és la primera opció que vam implementar en començar la pràctica. Tot i això, que sigui correcta no vol dir que sigui una bona solució, ja que és extremadament lenta.

En quant a les opcions b i c proposades, també són funcionals, però en ambdues tenim el problema de que bloquejem tot l'arbre quan actualitzem el contador, motiu pel qual, igual que la opció a, també resulten en una execució molt lenta, equiparable a crear només un procés fill.

La última opció que ens proposen és, amb molta diferència, la més ràpida de les quatre, ja que quan ha d'actualitzar el comptador no bloqueja tot l'arbre, sino només el/s nodes implicats en el procés. Ho expliquem amb més detall a continuació.

Per implementar només la paralització dels nodes hem hagut de modificar (a part del main obviament) *red_black_tree.h*, *tree_to_map.c*, així que entreguem dues pràctiques, una amb les tres opcions principals i l'altre amb la millora.

Per a fer-ho hem implementat un semàfor per node, és a dir, **cada node es bloqueja si un procés hi està escrivint**. Per a fer-ho hem afegit la variable *sem_t clau_node* al mètode *diccionari_arbre()*. Ho hem fet així per evitar-nos haver de recórrer tots els nodes inicialitzant els semàfors i, com que maparem l'arbre a memòria, ja estaran compartits per tots els processos en els que s'hi pot accedir. Afegint *sem_wait(clau_node)* i *sem_post(clau_node)* on hem de sumar la vegada al node ja hem acabat.

Perquè aquest muntatge funcioni, hem de modificar el mètode *serialize_node _data_to_mmap_recursive()*, de manera que no només refresqui les keys i el nombre de cops que apareix al 'updatejar', sinó que també ho faci amb els semàfors.

Així doncs, entreguem dues implementacions per a aquesta pràctica, una amb dos semàfors, més lenta, que bloqueja tot l'arbre en actualitzar el comptador, i una amb un semàfor per cada node, molt més ràpida.