Efficient struct storage

Data: 27/03/2023
Tags: #SO #C #SoftwareEngineering

Array

You can apply a data structure to a binary file of structs by creating an index or a lookup table. An index is a data structure that maps some value, such as a key, to the byte offset of the corresponding record in the file. A lookup table is a data structure that contains all the records in the file, indexed by some key or keys.

Here's an example of how you can create an index for a binary file of `Process_Info` structs:

```c
// Define a struct to hold an index entry
typedef struct index_entry {
    int pid;
    long offset;
} Index_Entry;

// Open the binary file for reading
FILE* fp = fopen("processes.bin", "rb");

// Create an index as an array of index entries
Index_Entry index[1000];  // assume there are at most 1000 records

// Read the file and populate the index
int i = 0;
while (fread(&info, sizeof(Process_Info), 1, fp) == 1) {
    if (info.active) {
        index[i].pid = info.pid;
        index[i].offset = ftell(fp) - sizeof(Process_Info);
        i++;
    }
}

// Close the file
fclose(fp);
```

In this example, the `Index_Entry` struct contains a PID and a byte offset. The `index` array is populated by reading each `Process_Info` struct from the file and checking whether it's active. If it's active, the PID and offset are added to the index. The `ftell` function returns the current byte offset of the file pointer, which is the end of the current record. We subtract the size of the `Process_Info` struct to get the offset of the beginning of the record.

With this index, you can efficiently search for a `Process_Info` struct by PID:

```c
int find_process_by_pid(int pid, Process_Info* info) {
    // Search for the PID in the index
    int i;
    for (i = 0; i < 1000; i++) {
        if (index[i].pid == pid) {
            break;
        }
    }
    if (i == 1000) {
        return 0;  // not found
    }
    // Seek to the offset of the corresponding record in the file
    fseek(fp, index[i].offset, SEEK_SET);

    // Read the record from the file
    if (fread(info, sizeof(Process_Info), 1, fp) == 1) {
        return 1;  // success
    } else {
        return 0;  // error
    }
}
```

This function searches for the PID in the index and returns the corresponding `Process_Info` struct, if found. It seeks to the offset of the record in the file and reads it using `fread`. Note that this function assumes that the file is sorted by PID, which may not be the case if the file is modified frequently. If the file is not sorted, you can use a different data structure, such as a hash table, to index the records.

Hash Table

A hash table is another data structure that you can use to apply a data structure to a binary file of structs. A hash table is a data structure that maps keys to values by computing a hash function on the key. The hash function maps the key to an index in the hash table, which points to a linked list of key-value pairs that have the same hash value.

Here's an example of how you can create a hash table for a binary file of `Process_Info` structs:

```
// Define a struct to hold a hash table entry
typedef struct hash_entry {
    int pid;
    long offset;
    struct hash_entry* next;
} Hash_Entry;

// Define the hash table size
#define HASH_TABLE_SIZE 1000

// Open the binary file for reading
FILE* fp = fopen("processes.bin", "rb");

// Create the hash table as an array of hash entries
Hash_Entry* hash_table[HASH_TABLE_SIZE];
for (int i = 0; i < HASH_TABLE_SIZE; i++) {
    hash_table[i] = NULL;
}

// Read the file and populate the hash table
Process_Info info;
while (fread(&info, sizeof(Process_Info), 1, fp) == 1) {
    if (info.active) {
        // Compute the hash value of the PID
        int hash_value = info.pid % HASH_TABLE_SIZE;

        // Create a new hash entry for the PID and offset
        Hash_Entry* new_entry = (Hash_Entry*) malloc(sizeof(Hash_Entry));
        new_entry->pid = info.pid;
        new_entry->offset = ftell(fp) - sizeof(Process_Info);
        new_entry->next = NULL;

        // Insert the new entry into the hash table
        if (hash_table[hash_value] == NULL) {
            hash_table[hash_value] = new_entry;
        } else {
            Hash_Entry* current_entry = hash_table[hash_value];
            while (current_entry->next != NULL) {
                current_entry = current_entry->next;
            }
            current_entry->next = new_entry;
        }
    }
}

// Close the file
fclose(fp);
```

In this example, the `Hash_Entry` struct contains a PID, a byte offset, and a pointer to the next entry in the linked list. The `hash_table` array is initialized to NULL pointers, and each entry is populated by reading each `Process_Info` struct from the file and checking whether it's active. If it's active, the PID and offset are hashed using a simple modulo function, and a new `Hash_Entry` is created and inserted into the linked list at the corresponding index.

With this hash table, you can efficiently search for a `Process_Info` struct by PID:

```
int find_process_by_pid(int pid, Process_Info* info) {
    // Compute the hash value of the PID
    int hash_value = pid % HASH_TABLE_SIZE;

    // Search for the PID in the linked list
    Hash_Entry* current_entry = hash_table[hash_value];
    while (current_entry != NULL) {
        if (current_entry->pid == pid) {
            break;
        }
        current_entry = current_entry->next;
    }
    if (current_entry == NULL) {
        return 0;  // not found
    }

    // Seek to the offset of the corresponding record in the file
    fseek(fp, current_entry->offset, SEEK_SET);

    // Read the record from the file
    if (fread(info, sizeof(Process_Info), 1, fp) == 1) {
        return 1;  // success
    } else {
        return 0;  // error
    }
}
```

This function searches for the PID in the index and returns the corresponding `Process_Info` struct, if found. It seeks to the offset of the record in the file and reads it using `fread`. Note that this function assumes that the file is sorted by PID, which may not be the case if the file is modified frequently. If the file is not sorted, you can use a different data structure, such as a hash table, to index the records.