AE2OSC

AE2OSC COURSEWORK DOCUMENTATION REPORT

19th May 2016

Name: Yichen PAN
Student ID: 6514897

University of Nottingham Ningbo China
School of Computer Science

Contents

1 INTRODUCTION

This is a documentation report for the AE2OSC Coursework which is to make use of operating system APIs (specifically, the POSIX API in Linux) and simple concurrency directives to implement a continuous file allocation algorithm.

Implementaion ideas and underlying principles are discussed below.

2 IMPLEMENTATION

2.1 Task 1

2.1.1 Interprocess communication

Interprocess communication (IPC) is the transfer of data among processes. One of the simplest interprocess communication methods is using shared memory. Shared memory allows two or more processes to access the same memory as if they all called malloc and were returned pointers to the same actual memory.

2.1.2 Implementation

The POSIX shared memory facility used here is a simple, formal interface to the use of mmap() to share segments.

At first, the shm_open() function is used to establish a connection between a shared memory object and a file descriptor. It creates an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor is used then by other functions (i.e. different process) to refer to that shared memory object.

Then, mmap() function is used to form an association between a file and a process's memory. Mapped memory permits different processes to communicate via a shared file. Mapped memory can be used for interprocess communication or as an easy way to access the contents of a file.

2.2 Task 2

2.2.1 Contiguous Allocation

In this task, contiguous memory allocation is implemented, where each block is one byte (the size of each character in the string/file). In practice, this kind of memory allocation has several benefits.

1. It is simple to implement as keeping track of the blocks allocated to a file is reduced to storing the first block that the file occupies and its length.
2. The performance of such an implementation is good as the file can be read as a contiguous file.

At the same time, it also has several drawback.

1. The operating system does not know, in advance, how much space the file can occupy. If the file is sized at 100K and later grows to 150K there may not be enough contiguous space to meet the new file size.
2. This method leads to fragmentation. As files are created and deleted holes will be left which may not be big enough to place a file.

2.2.2 Keep tracking of the memory usage

In order to know which memory is free and which memory is being used, we need to provide a memory usage tracking mechanism.

Two implementations are discusses and realized in the code.

2.2.3 Bit Maps

Under this scheme, the memory is divided into allocation units and each allocation unit has a corresponding bit in a bit map. If the bit is zero, the memory is free. If the bit in the bit map is one, then the memory is currently being used.

In the pratical implementation, the idea of bit maps is followed but rather than use one bit to represent the state of the memory block, one byte short integer is chosen. A fixed length of one-dimenssional array is used to store these bits.

2.2.4 Linked List

Free and allocated memory can be represented as a linked list. Each entry in the list contains the start, end location and a hole value which indicates the empty memory in this block. It also contains a pointer to the next memory entry.

2.2.5 Best fit algorithm under two

Best fit searches the entire list and uses the smallest hole that is large enough to accommodate the process. The idea is that it is better not to split up a larger hole that might be needed later.

For the linked list implmentation, when the producer inserts the files into the disk, the allocator traverses the free block list looking for a block large enough but with smallest size. If the hole part of the list entry has the same size as the requested memory, the entry is erased from the list. If the hole part size is bigger than needed, the entry is splitted in two entries, one of the requested size and the other with remaining size.

When the consumer deletes the files from the disk, the deallocator traverses the list to find the entry that the start address of the file to delete is in.

With regard to the bit map implementation, a nested loop is used to traverse the array and find the smallest continous memory that could fit the file.

2.3 Task 3

Due to the fact that in the file table, the address of the start location (i.e. of type char*) of the file (i.e. string) rather than a simple integer is stored, it is meaningless to reset it to -1. Instead, the index information in the file table is set to -1 which indicates the file has been deleted. It is sometimes necessary for two processes to communicate with one another. This can either be done via shared memory or via a file on disc. In this task, several threads are working on one shared memory buffer. To avoid race conditions, the idea of critical section, which ensures that one processes, whilst using the shared variable, does not allow another process to access that variable, is adopted here.

1. No two processes may be simultaneously inside their critical sections.
2. No assumptions may be made about the speed or the number of processors.
3. No process running outside its critical section may block other processes

4. No process should have to wait forever to enter its critical section.

Basically, in order to implement mutual exclusion, there are several common strategies, such as disabling Interrupts, lock variables, PetersonâĂŹs Solution, Test and Set Lock (TSL) and Sleep and Wakeup.

The problem scenario in task 5 is a classic problem called Producer-Consumer Problem. The logic behind my solution is that a variable, numOfFilesEver, that keeps track of the number of files the producer have generated. Particularly, the producer will check numOfFilesEver against NUM_OF_FILES (the predefined and specified number of files the producer needs to generate). If numOfFilesEver = NUM_OF_FILES then the producer returns. Otherwise it puts the new generated file into the disk and increments numOfFilesEver.

At the same time, when the consumer deletes a file from the disk, it first checks if another variable, numOfFilesLeft, is zero. If it is, it sends itself to sleep. Otherwise it removes a file from the disk and decrements numOfFilesLeft.

The calls to WAKEUP occur under the following conditions.

1. Once the producer has added a file into the disk, and incremented numOfFilesEver, it checks to see if numOfFilesEver is more than the specified number. If it is, the producer thread is killed; It also increments the variable numOfFilesLeft.
2. Once the consumer has removed a file from the disk, it decrements numOfFilesLeft. Then, if it smaller than 0, it sleeps the consumers.

Semaphores are used to realize Sleep and Wakeup mechanism.

2.3.1 Semaphores

In (Dijkstra, 1965) the suggestion was made that an integer variable be used that recorded how many wakeups had been saved. Dijkstra called this variable a semaphore.

Basically, if it was equal to zero it indicated that no wakeup's were saved. A positive value shows that one or more wakeup's are pending.