Operating Systems

Assignment 1 Report

Georgios - Alexandros Vasilakopoulos
A.M. 1115202000018

Technical Details

- To compile & run:

```
$ make
$ ./parent <file_name> <lines_per_segment> <requests_per_child>
```

- The number of child processes is defined internally in `parent.c` as `NUMBER_OF_CHILDREN`.
- Upon execution, for every process, a text file is created. For the parent process, the file is named `parentReport.txt` and it contains the load and unload times of every requested segment. For any child process, the file is named `childReport<ID>.txt` [1], and it contains the request and response times, for every segment request made by that process. In order to remove the produced files easily, use `$ make clean`.
- The processes were timed according to the wall time [2], in order to obtain real, objective time measurements. CPU clock times would provide no information about the waiting times of each process. In the beginning, the parent process calls `time_0 = getTime(0)`, which returns a time counter. When a child process needs to know the current time, it will call `getTime(time_0)`
- The creation and destruction of the shared memory structure, defined in `shared_memory.h`, is managed by the parent process, through the corresponding static functions. The encapsulated memory within the structure is allocated dynamically, according to parameters such as `CHARS_PER_LINE` [3], and `number_of_segments`
- POSIX semaphores were used in order to synchronize the interactions between the processes. The static functions `initializeSemaphores(...)` and `unlinkSemaphores(...)` handle the creation and destruction of a semaphore "array". Each semaphore of the array is named arbitrarily as "`seg<i>`", where `<i>` is the index of the semaphore within the array. In order to avoid code repetition in error checking, macro functions were used (defined within `macros.h`)
- The child processes are generated through `fork()` and, upon creation, they call the `child(...)` function, which takes as arguments: a pointer to the shared memory, the semaphores, a time reference point `time_0`, and an `ID`, used to produce the report file.
- The children, initially, generate a random request. When a request is fulfilled, the new request will have a 0.7 probability to be on the same segment and an 0.3 probability to be completely random, once again.

Process Synchronization: Parent's Perspective

- Each segment of the file has a corresponding semaphore and a readers counter. All of the segment semaphores have an initial value of 1.
- As soon as the parent process finishes creating the children processes, it signals the semaphore `availabletoServe` and waits for the semaphore `writeRequest`

[1] `ID` is an integer between 1 and `NUMBER_OF_CHILDREN` and is unique for each child process
[2] through the `gettimeofday(...)` function
[3] global paremeter, defined in `parent.c`

- When `writeRequest` is signaled by some child process, this will imply that a segment request has been written into the shared memory. Therefore, the parent can now proceed to view the requested segment and load it into the shared memory.
- Afterwords, it will signal the semaphore `segmentLoaded`, implying that the segment is available to read, and it will wait until the semaphore `doneReading` is signaled, by some child.
- Once that happens, the parent process writes the time measurements of the segment into the report file and it reiterates the previous process, until the number of serves reaches the total number of requests : `NUMBER_OF_CHILDREN * requests_per_child`

Process Synchronization: A Child's Perspective

- When a child produces (internally) a request for a segment, there are two cases: Either it is the first process to request that specific segment, or another process has already requested the same segment.
- In the first case, the child process will not have to wait for `sem_array[requested_segment]`, because it was initialized to 1. Therefore, it will increment the readers counter variable of that segment (`sharedMemory->reader_counter[requested_segment]`), in order to indiacate to future incoming processes that they are not the first to request that segment. Afterwords, the process will wait for `availabletoServe`.

  Once it is signaled, the process can write the segment request into the shared memory, then, signal the semaphore `writeRequest`, and, then, wait for `segmentLoaded`.

  Once `segmentLoaded` is signaled, the process must notify the other processes that are waiting for the same segment. This is done by signaling `sem_array[requested_segment]`. Remember that the process had downed the semaphore as soon as it produced the request. This means that other processes with the same segment request would wait for the semaphore to be singaled.

  Next, the process would access the segment in the shared memory, and, when it finishes, it will decrement the readers counter, after waiting for `sem_array[requested_segment]`, in order to ensure that no other process is altering the counter at the same time.
- In the second case, as mentioned, the child process will have to wait for `sem_array[requested_segment]` to be signaled by the first reader. As soon as the first reader signals the semaphore, the process will increment the reader counter variable of the segment and signal `sem_array[requested_segment]` once again.

  After that, it can access the segment data and once it is done, decrement the reader counter. If the counter becomes 0, then the process must notify the parent that no other processes are waiting for that segment. This is done by signaling `doneReading`

Observations

- For every segment `i`, its corresponding semaphore, essentially, ensures that no two processes can alter the value of `sharedMemory->reader_counter[i]` at the same time. The first process that requests segment `i` will alter the value of the counter and, while it's waiting for `availabletoServe`, all the other processes that also request `i` will halt, as if the first process is still changing the value of the counter.
- This entire setup operates in a FIFO manner [4]:

  Assume that two processes that request different segments appear. Both of them will down the semaphores of their segment and they will both wait for `availabletoServe`. The first one to arrive will be served first by the parent and it will signal the segment semaphore again, while the second to arrive will have to wait until `availabletoServe` is signaled again.

[4] as long as `sem_wait()` respects FIFO priority

If, on top of these two, another two processes arrive, identical to the previous two, the one whose segment is being served will access the segment immediately, while the other will have to wait for `sem_array[requested_segment]` to be signaled by the process which is currently waiting for `availabletoServe`.

- Notice that the parent process is completely oblivious of the state of `sem_array[]`. The parent only interacts with processes that arrive first. This means that the processes that arrive first have the "responsibility" to notify the others when their segment is loaded.