Chapter 17
Design and Implementation of Blocking Shared Memory for Satellite Navigation Application Processing System

Weijie Sun, Enqiang Dong, Jidong Cao and Xiaoping Liu

Abstract Being the computing center of satellite navigation system, application processing system is charged with lots of high-precision computing tasks including orbit determining, time synchronizing, ionosphere model calculating, difference and integrity parameters processing, and it has a large number of features, such as mass and multi-sort data, heavy computing tasks, high precision of calculation, so there are much rapid mass-data transmissions between different processes. As the primary means of communication between processes, shared memory has the characteristic of rapid mass-data transmission. Therefore there are lots of shared memories to transfer data among different processes in application processing system. This paper describes the realization principle of shared memory, focuses on parsing the blocking shared memory design principle, achieves the blocking shared memory template class based on the principle, and through the test verifies the reliability of the template class.

Keywords Blocking shared memory · Satellite navigation · Application processing system

17.1 Introduction

With the continuing development of the satellite navigation technology, the satellite navigation system has been used in every aspect of our society. As the data processing center of the satellite navigation system, the application processing system undertakes the high-accuracy computation of the satellite orbit determining, time synchronizing, ionosphere model, difference and integrity parameters

W. Sun (✉) · E. Dong · J. Cao · X. Liu
Beijing Satellite Navigation Center, Beijing, China
e-mail: sun_weijie@sina.com.cn

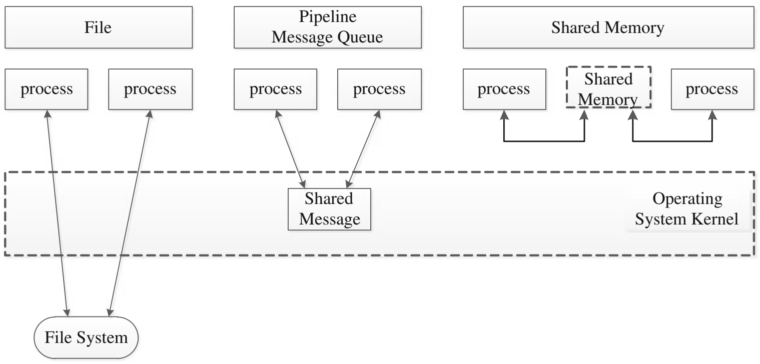

Fig. 17.1 Interprocess communication modes

processing and so on. Consequently, the application processing system has the characteristics like the high-level receiving data, the multi-types data, high-accuracy computation requirement and the heavy computation task. To ensure the real-time ability, the parallel computation technology is used to realize the high-speed processing of information by using multi- process and multi-thread technology. At the same time, the reliable communication between different processes to is required.

The most common used communication methods between processes include file, pipeline, message queue and shared memory, which theories are illustrated in Fig. 17.1.

- File

The communication between two processes is realized by shared files saved in the file system. Each process needs to traverse kernels like read, write, lseek functions etc. The synchronization mechanism is needed to realize the read and write protection between processes.

- Pipeline and message queue

The communication between two processes is realized by accessing shared messages maintained by operating system. The accessing operation by each process involves a system calling to kernels.

- Shared memory

The two different processes map their virtual addresses into the same shared memory addresses, regarding as their own virtual memories to realize communication between processes. Once the address mapping is established, the kernel is never needed on the data transfer between processes. The synchronization mechanism is needed to ensure the safety between processes.

The shared memory is common used in the application processing system of the satellite navigation because of the characteristic of rapid mass-data transmission. The System V mechanism of shared memories is illustrated in the paper.

17.2 The implementation Principles of Shared Memory

The shared memory is the most fast intercross communication type [1]. The virtual address of process can be mapped into any physical address. If the virtual addresses of two processes are mapped into the same physical address, the communication between processes can be realized by using their own virtual addresses [2]. As explained above, the kernel is never involved in the data transfer process Fig. 17.2.

The shared data needed by different processes are saved into the shared memory of IPC. The application process can get or establish a shared memory of IPC by using system function *shmget()* with corresponding identifier returned. For every established shared memory, a *shmid_ds* structure is maintained by kernel to describe the shared memory. The *shmid_ds* structure of every shared memory is saved into the *shm_segs* array. The access privilege, capability and physical address of shared memory are described by the *shmid_ds* structure.

```
struct shmid_ds
{
    struct ipc_perm shm_perm;          /*IPC access permission*/
    size_t shm_segsz;                  /* shared memory size*/
    struct vas* shm_vas;               /* virtual address entry list*/
    pid_t shm_lpid;                    /* current process pid*/
    pid_t shm_cpid;                    /*creating process pid*/
    time_t shm_atime;                  /*last shmat() time*/
    time_t shm_dtime;                  /*current shmdt() time*/
    time_t shm_ctime;                  /*last shmctl() time*/
    shmatt_t shm_nattch;               /*Number of processes connection*/
    unsigned long int shm_npages;      /*number of shared memory pages*/
    unsigned long int* shm_pages;      /*shared memory pages list*/
        …
}
```

Every process which use shared memory map its virtual memory into shared memory using system function *shmat()*. The mapping relation is described by the newly established *vm_area_struct*. When the process visit the shared memory at the first time, the system will distribute a physical page and establish the page table (pointed by *shm_pages*) and its entrance. This entrance is saved into the *vm_area_struct* corresponding to the process. When the next process uses the shared memory in the first time, the dealing function can be used directly to direct into the pages of the physical memory. Consequently, the first visiting process establishes the physical memory page and the succeeding processes can use the physical memory pages directly.

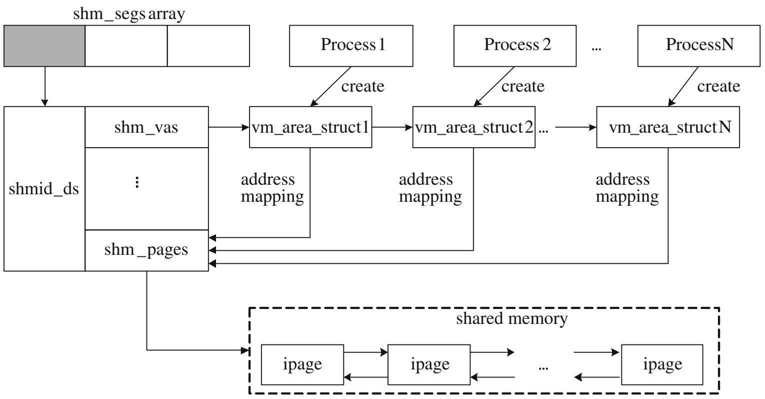

Fig. 17.2 Schematic diagram of shared memory

When the shared memory is not used by the process, the process call the *shmdt()* function to cancel the mapping of the shared memory. The corresponding *vm_area_struct* will be deleted from the *shmid_ds* structure. After the last process released its virtual address space, the system release the distributed physical page and delete the *shmid_ds* structure.

The *shmctl()* function realize the control operation of the shared memory.

17.3 Design and Implementation of Blocking Shared Memory

17.3.1 Design Principle

The two model of the shared memory can be summarized by analyzing the using method of share memory in the application processing system of the satellite navigation. One model is that one process writes and several processes read. For example, the pre-processed data received by the data receiving process is need to send to several calculation process, such as the satellite orbit determining, time synchronizing, ionosphere model etc. The second is that one process writes and only one process reads, such as the data receiving process only send its data to the pre-processing process.

At the same time, the synchronization mechanism is needed to ensure the communication safety between processes otherwise the condition that several processes read or write the shared memory simultaneously can be generated. The following errors may be generated [3]: the saved data in disorders, the taken data incomplete and the read process executes before the end of the write process. Consequently, the synchronization mechanism between different processes is needed to ensure the read or write safety of shared memory. In the paper, the value of signal is used as the synchronization mechanism between different processes.

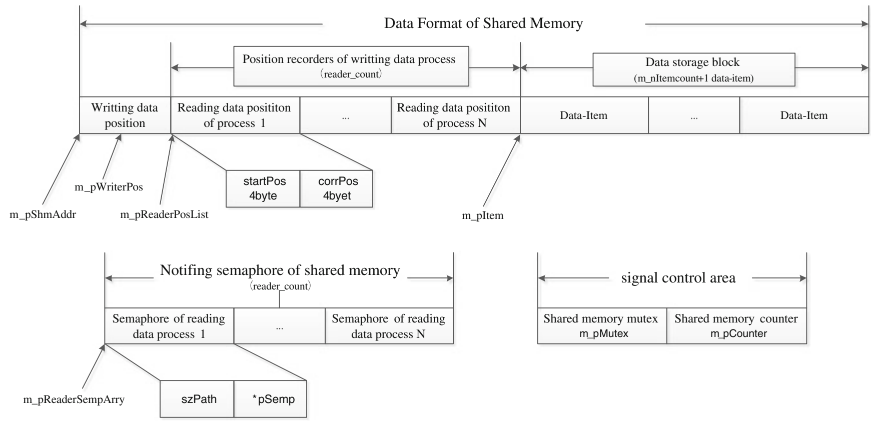
Data Format of Shared Memory
Position recorders of writting data process
(reader_count)
Data storage block
(m_nItemcount+1 data-item)
Writting data position
Reading data posititon of process 1
...
Reading data posititon of process N
Data-Item
...
Data-Item
startPos 4byte
corrPos 4byet
m_pShmAddr
m_pWriterPos
m_pReaderPosList
m_pItem
Notifing semaphore of shared memory
(reader_count)
Semaphore of reading data process 1
...
Semaphore of reading data process N
m_pReaderSempArry
szPath
*pSemp
signal control area
Shared memory mutex m_pMutex
Shared memory counter m_pCounter

Fig. 17.3 Design diagram of shared memory

The blocking shared memory is designed in the application processing system of the satellite navigation based on the general principle. The one to many or one to one communication type can be set flexibly by the configuration file. The design structure of the blocking shared memory is illustrated in Fig. 17.3. The shared memory includes three parts: the area of the shared memory, the data notice semaphore of the shared memory and the area of the signal control.

The shared memory includes the location of writing, the record of the location of the read process and the data protection area. The location of the data writing *m_pWritePos* is used to indicate the index of the written data in the data protection area. As the processing times are different among different applications, the un-synchronization of reading is generated. The indicator of the location of the reading processes is used to show the particular location of several reading processes. The data protection area is used to save the written data in the shared memory. The written process writes the data into the record item one by one. When the last item is written, the first item is written again for the next circulation. As a result, the whole data protection area is used as a circulation queue.

In the process of using the shared memory, if the shared memory is established firstly, the data written location is directed into the first data item and the read locations are set to zero. If the shared memory exists already, the pointer of the shared memory, *m_pShmAddr*, is got by using the function *shmat()*. Then the particular data written location and the information of the reading process can be got through the *m_pShmAddr* pointer.

The semaphore of every read process is set by the semaphore of the shared memory. Every semaphore includes the indication of the signal lamp *szPath* and the pointer *pSem*. When the data is written into the shared memory, the data notice signal of the shared memory is used to inform every reading process that the written process is over. The function *acquire()* is used to block and wait the coming of the data until the function *release()* inform the write can be done.

To avoid the read/write conflicts of the shared memory, the read/write operation synchronization is realized by the signal control area. The signal control area includes the mutex and the counter of the shared memory. The mutex *m_pMutex* is mainly used to realize the data read and write synchronization. The shared memory can be accessed only when the visiting control of *m_pMutex* is got. The counter *m_pCounter* of shared memory is a group of semaphore. For every process which uses the shared memory, the counter is added one using *m_pCounter->op(1, 0, SEM_UNDO)* to record the number of the processes that use the shared memory (the record *SEM_UNDO* is set and the counter is subbed one when the process quit). When other processed quit and only the current process use the shared memory, the current process call the system function *shmdt()* to delete the shared memory.

17.3.2 Design Implementation

Using C++ language, this paper implements a template class blockSHM of blocking shared memory based on ACE middleware (Table 17.1).

template<typename TYPE>class blockSHM

There, *TYPE* is the abstract data structure of shared memory data item.
The main parameters and functions of blockSHM class as follow:

```
typedef struct READER_POS
{
        int startPos;        /* last position of reading process getting the data*/
        int currPos;         /*current position of reading process getting the data*/
}READER_POS;
typedef struct READER_SEMP
{
        char szPath[NAME_LENGTH];            /*semaphores flag*/
        ACE_Process_Semaphore *pSemp;        /* semaphore pointer*/
}
```

Table 17.1 Main parameters of the blockSHM shared memory template class

Parameter declaration	Parameter description
*char *m_pShmAddr*	*Starting address of shared memory*
*ACE_SV_Semaphore_Complex *m_pCounter*	*Counter pointer of shared memory*
*ACE_Shared_Memory_SV *m_pShmObj*	*Object pointer of shared memory*
*ACE_Process_Mutex *m_pMutex;*	*Mutex pointer of shared memory*
*int *m_pWriterPos*	*Writing data location, index of the data storage block*
*TYPE*m_pItem*	*Data item pointer*
int m_nItemCount	*The number of storage data*
*READER_POS *m_pReaderPosList*	*Starting address of location array for reading process*
*ACE_Process_Semaphore *m_pReaderSemp*	*Starting address of semaphore array for reading process*

blockSHM shared memory template class has four main functions:

init_writer () function is used to complete shared memory initialization by writing process.

```
int init_writer(const char* name, const int count, const int reader_count, bool replace)
{
    Judge the variables rationality;
    Create shared memory counter(if created, open), counter is incremented by 1;
    Create the shared memory mutex between processes;
    Loop to create shared memory semaphores base on the number of reading processes;
    create shared memory, if created, open and get the writing position in shared memory;
}
```

init_reader() function is used to complete shared memory initialization by reading process.

```
int init_reader(const char* name, const int count, const int reader_count, const short reader_sn)
{
    Judge the variables rationality;
    Create shared memory counter (if created, open), counter is incremented by 1;
    Create the shared memory mutex between processes;
    Loop to create shared memory semaphores base on the number of reading processes;
    Create shared memory, if created, open and get the reading position of the current process;
}
```

put() function is used to write data into shared memory. It will write a data-item at every calling and send signals to notify all processes that are blocked and waiting to read the shared memory.

```
int put(const TYPE &Item)
{
    Lock shared memory mutex;
    Write data;
    Loop to signal shared memory semaphores by calling release() function based on the number of
    writing process;
    Unlock shared memory mutex;
}
```

get() function get only one data-item from shared memory at every calling

```
int get(TYPE &outItem)
{
    Reading process call acquire() of shared memory semaphore to block and wait ultil the release
    signal of writing process;
    Lock shared memory mutex;
    Read data;
    Unlock shared memory mutex;
}
```

17.4 Experiment and Result Analysis

With the blockSHM template class of shared memory, this article constructs Server.cpp and Client.cpp program to test the transmission delay and functionality of shared memory. The structure of test programs is shown in Fig. 17.4.

In the test, Server program put data-item into shared memory, Client program is blocking and waiting to get data-item from shared memory when it receives the signal of the shared memory semaphore. Each data packet contains the Server program's sending data time, after receiving the sending data packet, Client program calls system function to get local time and obtain the shared memory data transmission delay by calculating the difference value between local time and sending time. Server program loop to send 3000 data packets and set 1K and 100K data packets size respectively in the test.

The test work selects a HP rx4640 server, 4 × 1.6G CPU, 8G memory.

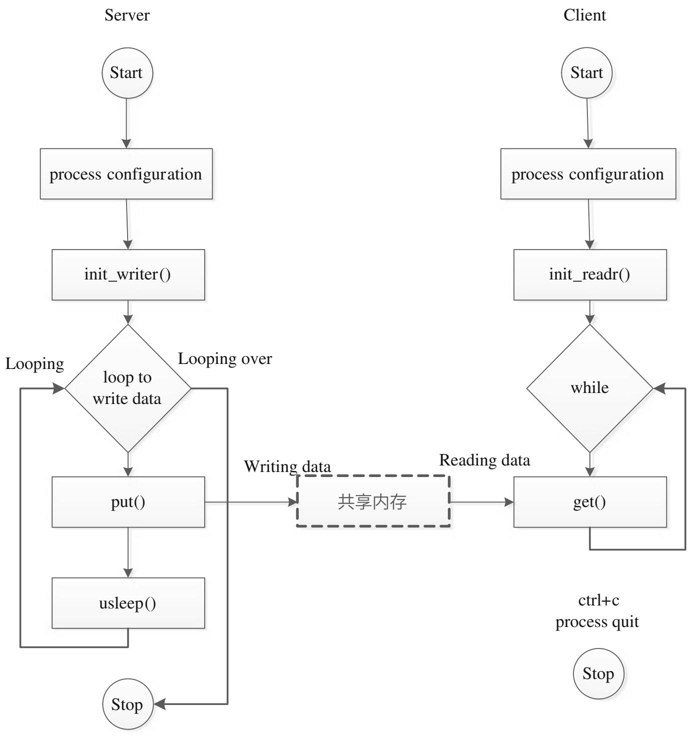

Fig. 17.4 Structure diagram of test programs

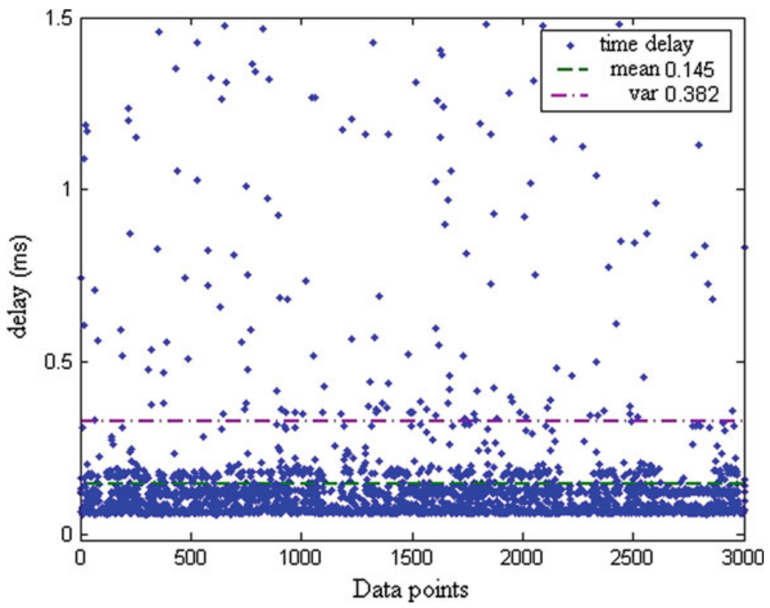

Fig. 17.5 Shared memory transmission delay: 1K data packet

17.4.1 Capability Analysis

1K data packet, shared memory transmission delay results are show in Fig. 17.5.

100K data packet, shared memory transmission delay results are shown in Fig. 17.6.

Test results of shared memory data transmission delay are shown in Table 17.2.

From the results, 1k data packet transmission delay is consistent with 100k data packet transmission delay, so shared memory transmission delay is not directly related to the size of data packet size. Analysing the reason, shared memory directly operate the memory and use *memcpy()* system function to copy block data for putting or getting data, the size of the data is basically no effect on the operation delay of *memcpy()* function, And thus the transmission delay is basically the same.

There are more discrete points of large transmission delay in Figs. 17.5 and 17.6. Analysing the reason, there are other processes working in the server, the processes will be phased share system resources to run, and the running of Server

Table 17.2 Transmission delay of shared memory

Data packet size	Mean (ms)	Variance (ms)
1K	0.145	0.382
100K	0.175	0.423

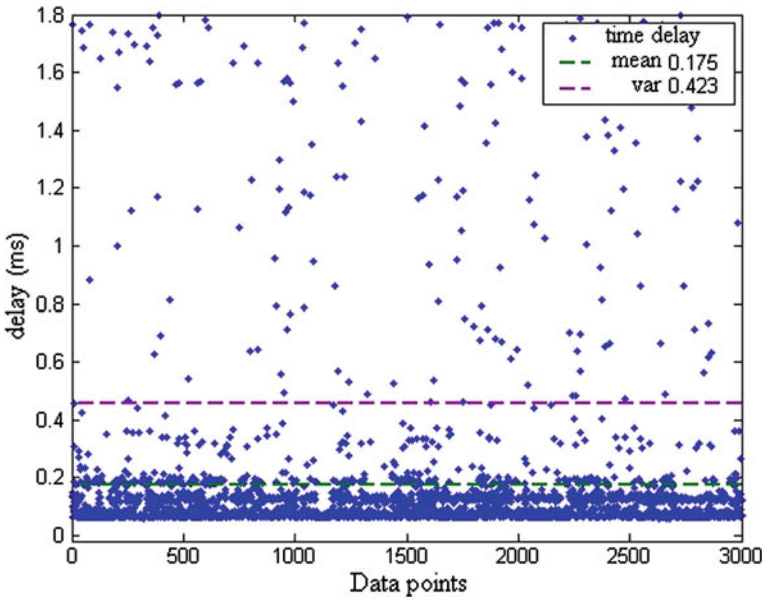

Fig. 17.6 Shared memory transmission delay: 100K data packet

and Client process need to compete for resources with other processes, therefore, the reasons for more discrete points of large transmission delay can be attributed to the waiting time of the Server process and Client process competing for resources.

17.4.2 Functionality Analysis

This paper first constructs shared memory test environment which is characterized by Server process for writing data and multiple Client process for reading data that are derived by Server.cpp and Client.cpp separately. Then, a data packet transmission experiment using different transmission frequency and size for packet is conducted to test the blockSHM obstructive shared memory. The test procedure does not demonstrate abnormal phenomenon. Moreover, the phenomenons of packet loss and data transfer that are delay too long are absence in blockSHM obstructive shared memory. The experiment indicates that template class for blockSHM shared memory shows promising robustness and stability.

17.5 Conclusion

This paper first discusses the basic principle of shared memory. Based on the analysis of inter-process communication requirement of satellite navigation business processing system, the paper focuses on principle of design and realization for obstructive shared memory, then successfully constructed template class for blocking shared memory. After that, an experimental analysis is conducted to test shared memory. The experiment verifies the effectiveness and stability of template class for the blocking shared memory, which provides basic platform components for the development of the navigation system in the future.

References

1. Stevens WR (2003) UNIX network programming, interprocess communications (vol 2, 2nd edn). Tsinghua University Press, Beijing, pp 261–262
2. Guo X, Gao S (2001) The analysis of unix system V IPC and share memory audit. Appl Res Comput 18(4):39–40
3. Zhang H, Sun C, Li J (2004) Research and implementation of synchronized shared memory in Linux. J HUNAN Indus Polytech 4(4):19–20