developerWorks®

Logging in multi-threaded applications efficiently with ring buffer

How to find and analyze bugs in your applications

Manish Katiyar
Devendra Jadhav

August 14, 2007

No software is bug free, and application users can encounter unexpected results during the run time of programs. To analyze and find the cause of problems, logging is a method widely used by programmers. In this article, learn how to use a ring buffer for efficient logging with memory operations in place of file operations. Choosing an appropriate size for the buffer ensures that relevant messages are dumped, which can help when debugging.

Introduction

"There are two ways to write error-free programs; only the third works."—Alan J. Perlis

Logging is a very important activity in the lifetime of critical computer applications, especially when symptoms of failures aren't readily apparent. Logging provides the maximum amount of detail of the application state before the failure, such as value of variables, return value of functions, and so on. A monotonically increasing amount of trace data is produced over a period of time, which is written continuously to a text file on the disk. A significant amount of disk space is required for effective logging, and it increases many fold in a multi-threaded environment where numerous threads write their trace.

Two major problems with regular file logging are: availability of space on the hard disk and slow disk I/O while writing data to a file. Continuous writing to a disk can largely degrade the performance of a program, causing it to run slowly. Often the space issue is solved by using a log rotation policy, where logs are kept in several files that are truncated and overwritten as they reach a certain number of predefined bytes.

To overcome the space issue and minimize disk I/O, some programs log their trace data in memory, dumping it only when requested. This circular, in-memory buffer is known as *ring buffer*. This article discusses common implementations for ring buffer and proposes some ideas for enabling a ring buffer mechanism in multi-threaded programs.

Ring buffer

A ring buffer is a logging technique for applications whereby the relevant data to be logged is kept in memory instead of writing it to a file on a disk every time. This data in memory can be dumped to disk when required, such as when a user requests to dump memory data in a file, a program detects an error, or a program crashes due to an illegal operation or signal received. Ring buffer logging consists of a fixed size of allocated memory buffer that is used by the process for logging. As the name suggests, the buffer is implemented in a circular fashion. As the buffer fills with data, instead of allocating more memory for new data it is again written at the start, thus, overwriting the previous contents. See Figure 1 for an example.

Figure 1. Writing to a ring buffer

Figure 1 shows the states of a ring buffer when two entries of a log are written into it. After writing the first log entry, shown in blue, when the process tries to write the second one, shown in red, the buffer doesn't have enough space left. The process writes whatever data it can until the end of the buffer, and the remaining data is copied to start overwriting the previous log entry.

See Related topics for an example of a ring buffer implementation.

Reading from a ring buffer is accomplished by keeping a read pointer; this pointer and the write pointer are moved accordingly to ensure that the read pointer never crosses the write pointer while reading. To improve efficiency, some applications keep raw data instead of putting the formatted data in the buffer. A parser is required in such cases, which generates meaningful log messages out of those memory dumps.

Advantages of a ring buffer

Why would you use ring buffers when you can simply write to a file? Since you are overwriting the previous contents in a ring buffer, you lose data by doing so. A ring buffer provides the following advantages over the traditional file logging mechanism.

- It is fast. Writing to memory is much faster than doing an I/O to disk. Flushing the data is only performed when needed.
- Continuous logging can fill up space on the systems, causing other programs to also run out of space and fail. In such cases, either logs have to be manually removed or a log rotation policy has to be implemented.
- Once you've enabled logging, the process continues to fill up space on the hard disk, whether you need it or not.

- Sometimes you just need the data before the program crashed, rather than the complete history of the process.
- Common functions used for debugging, such as printf, write, and so on, can sometimes change the behavior of a program in the case of multi-threaded applications, making them hard to debug. Using these functions can cause the application to not reveal certain bugs that are noticed otherwise. The functions are cancellation points and might cause a pending signal to get delivered in a threaded environment when the program is not expecting it. The hypothetical sample (pseudocode) in Listing 1 and Listing 2 below make it clearer.

Listing 1. Code without debugging enabled

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL);
/* I should not be cancelled in the below section */
var=5;
#ifdef DEBUG
 write(fd,"Value of var = 5\n",17);
#endif
var=pow(var,2);

/* I can be cancelled now */
pthread_testcancel();
```

Listing 2. Code with debugging enabled

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL);
/* I should not be cancelled in the below section */
var=5;
#ifdef DEBUG
write(fd,"Value of var = 5\n",17);  <========= Cancel delivered here!
#endif
var=pow(var,2);

/* I can be cancelled now */
pthread_testcancel();
```

Using a ring buffer in multi-threaded programs

Sometimes ring buffer logging can come to the rescue when other traditional logging methods fail. This section discusses some important points to consider when using a ring buffer to enable logging in a multi-threaded application.

Synchronization has always been an indispensable part of multi-threaded programs while accessing a common resource, and logging is no exception. As each of the threads try to write to the global space, it is necessary to take care that they write to the memory in sync, or else the messages will get corrupted. Typically, each of the threads take a lock before writing to the buffer, which is released on completion. You can download an example of a ring buffer using locks to write to memory.

There is a downside to this approach: If you have an application with several threads and each thread is logging at a verbose level, the overall performance of the process is affected because the threads spend most of the time acquiring the lock and releasing it.

Synchronization issues can be completely avoided by having each thread write into its own chunk of memory. When a request comes from a user to dump the data, each thread takes a lock and dumps it to a central place. Since the lock is acquired only while flushing data to disk, performance is not extensively affected. In such cases, you might need an additional program to sort the log information either on thread ID or timestamp in chronological order to analyze it. You can also opt to write only the message codes instead of writing the complete formatted messages in memory, which are later converted to meaningful texts by parsing the dump using an external utility.

Another way to avoid synchronization issues is to allocate a large chunk of global memory and break it into smaller slots, where each slot is to be used by one thread for logging. Each thread can read and write only to its own slot, rather than to the complete buffer. As each thread tries to write data for the first time, it tries to find an empty slot of the memory and marks it as busy. The usage of slots can be tracked using a bitmap set to `1` when a particular slot is acquired by the thread, and it is again reset to `0` when the thread exits. A global list of the currently used slot numbers is maintained, along with the thread information of the threads using it.

To avoid situations where a thread died without resetting the bitmap of the slot to `0`, you need a garbage collector thread that traverses the global list and polls at fixed intervals for the thread based on thread ID. It frees the slot and modifies the global list. See Listing 3 below for an example.

Listing 3. Sample pseudocode for garbage collector thread

```
void Check_and_free(List *ptr){
    int slotno,ret_val;
    LockList();
    while(ptr){
    if ( ((ret_val = pthread_kill(ptr->thread_id,0)) == ESRCH) ){
       /* Thread has died */
       slotno=ptr->slotno;
       Free_slot(ptr->thread_id);
       Mark_bitmap_free(slotno);
      }
      ptr=ptr->next;
    }
    UnlockList();
    return ;
}
```

Thread IDs are reused often, so there could be a case where a thread died without releasing the slot, and comes up and allocates a new slot before garbage collector could free it. It's very important for the new threads to check the global list and reuse the same slot, if it was used by its previous instance. Since both the garbage collector thread and the writer thread can try to modify the global list at the same time, some kind of locking mechanism also has to be used.

As the user signals the process to dump the ring buffer data, the thread that handles the signal stops other threads from changing the contents of the buffer anymore and dumps the contents of the used slots in a file. Listing 4 and Listing 5 show samples of writing the data to a ring buffer and dumping its contents to file. The `is_dumping` global variable is used to stop other threads from changing the contents of the buffer once a signal has been received to dump data.

Listing 4. Sample pseudocode for writing to slot *'i'* of a ring buffer

```
void Write_to_buffer(char *msg){
     read_atomically(&is_dumping);
     if(!is_dumping)
       memcpy(slot[i]->ptr,msg,strlen(msg));
     return;
}
```

Listing 5. Sample pseudocode to dump data of a ring buffer

```
void Dump_data(int fd){
     change_atomically_to_true(&is_dumping);
     for i in each_used_slot {
        write_slot_data_to_file(fd,slot[i]);
      }
     change_atomically_to_false(&is_dumping);
     return;
}
```

Conclusion

A ring buffer makes logging efficient by using memory operations in place of file operations. Choosing an appropriate size for the buffer ensures that relevant messages are dumped, which can help when doing a post-mortem analysis of a program. A ring buffer is an ideal solution for programs that log continuously, and you don't need a complete history of the program when debugging. This article discussed a few approaches and things to be aware of while implementing ring buffer in multi-threaded programs.

Downloadable resources

Description	Name	Size
Sample C program for a ring buffer	au-buffer.zip	2KB

Related topics

- Ring buffer: Learn how to implement a general ring buffer.
- trace: The AIX®`trace` command is used to trace system events. The `-I` command-line option enables the circular in-memory buffer for logging.
- AIX client logging and tracing discusses circular memory based tracing used by SANFS clients on AIX.
- Check out other articles and tutorials written by Martin Brown:
    - Across developerWorks and IBM
- AIX and UNIX: The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- Search the AIX and UNIX library by topic:
    - System administration
    - Application development
    - Performance
    - Porting
    - Security
    - Tips
    - Tools and utilities
    - Java™ technology
    - Linux
    - Open source
- IBM trial software: Build your next development project with software for download directly from developerWorks.