Online journal ISSN - 1676-5680
GMR Genetics and Molecular Research
EVOLUTION AND TECHNOLOGY

An I/O device driver for bioinformatics tools: the case for BLAST

Renato Campos Mauro and Sérgio Lifschitz

Departamento de Informática PUC-RIO,
Pontifícia Universidade Católica do Rio de Janeiro,
Rua Marquês de São Vicente, 225 Gávea,
22453-900 Rio de Janeiro, RJ, Brasil
Corresponding author: S. Lifschitz
E-mail: sergio@inf.puc-rio.br

Genet. Mol. Res. 4 (3): 563-570 (2005)
Received May 20, 2005
Accepted July 8, 2005
Published September 30, 2005

ABSTRACT. There are many bioinformatics tools that deal with input/output (I/O) issues by using filing systems from the most common operating systems, such as Linux or MS Windows. However, as data volumes increase, there is a need for more efficient disk access, *ad hoc* memory management and specific page-replacement policies. We propose a device driver that can be used by multiple applications. It keeps the application code unchanged, providing a non-intrusive and flexible strategy for I/O calls that may be adopted in a straightforward manner. With our approach, database developers can define their own I/O management strategies. We used our device driver to manage Basic Local Alignment Search Tool (BLAST) I/O calls. Based on preliminary experimental results with National Center for Biotechnology Information (NCBI) BLAST, this approach can provide database management systems-like data management features, which may be used for BLAST and many other computational biology applications.

Key words: BLAST, Driver, Databases

INTRODUCTION

There are many tools for bioinformatics. Most of them involve reading data from text files or from binary files, always accessed directly from the operating system. This nostalgic scenario leads us to remember the era before the widespread use of database management systems (DBMS). Biological databases are increasing exponentially and efficient data manipulation is necessary.

Operating systems provide efficient data access, including buffer management and caching techniques. However, operating system algorithms were designed for a wide range of application classes, in order to achieve good performances for average applications, not for specific ones. Every application is viewed as a system process, with no particular characteristics taken into account. Consequently, an efficient computational resource management specialized for bioinformatics applications is a very important research issue. The challenges for managing the required computational resources include efficient secondary memory organization, appropriate access methods and specific buffer management policies. Particularly, the DBMS research area is concerned with specialized data management techniques that enable good performance.

We propose an operating system device driver, which developers can use to implement their own data management approach, such as an *ad hoc* buffer page replacement policy. The main difficulty to transform these experimental results into an actual implementation is the fact that current implementations are not easy to change in order to manage all input/output (I/O) calls. Even when this is possible, users would have to change their current (stable and reliable) software versions and migrate to the modified versions. Our approach is to keep current application codes unchanged.

In order to test our approach, we have directed our device driver to work with Basic Local Alignment Search Tool (BLAST) I/O requirements. BLAST (Altschul et al., 1990) is one of the most popular programs used to search and compare biological sequences. To the best of our knowledge, all current BLAST implementations (e.g., NCBI-BLAST (NCBI, 2005), WU-BLAST (WU-Blast, 2005)) execute flat formatted files stored in conventional operating system file systems.

The new device provides a conventional file system interface from which BLAST can open files as if they were regular files. The device captures the file system calls (e.g., open, seek) allowing programmers to implement specific memory management policies (Lemos and Lifschitz, 2003). The main advantage of this approach is the fact that it is not intrusive. Researchers can still use their favorite BLAST program in its current version.

DEVICE DRIVER IMPLEMENTATION

Our implementation was developed for a Linux operating system, but the solution proposed can be implemented in other Unix-like operating systems.

What is a device driver

A device driver, often called a driver for short, is a computer program that is intended to allow another program (typically an operating system) to interact with a hardware device. Think of a driver as a manual that gives the operating system instructions on how to use a particular piece of hardware.

Modern operating systems (Tanenbaum, 2001) use device driver technology to separate specific hardware characteristics from the operating system implementation. Printer Device drivers is one popular example. In Microsoft Windows, programs do not send commands directly to the printer. Instead, they send commands to an abstract layer managed by the operating system and the driver program (usually provided by the printer manufacturer) transforms the abstract command to one or more commands acceptable by the actual hardware.

The file management and access architecture are the same. We can use device drivers to hide hardware implementation details. In programs written for the UNIX family operating system, writing data in a text file or printing one string in a video terminal is the same system call, but internally the write implementation is completely different. The difference is implemented by the device driver (Figure 1).

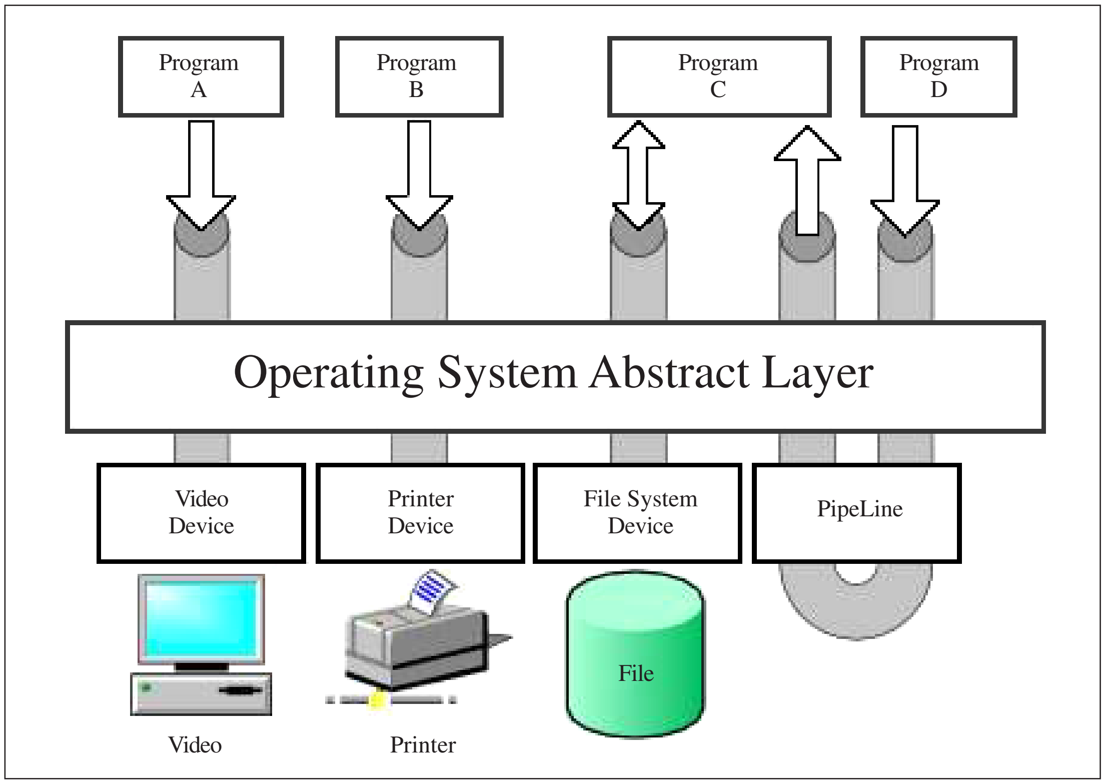


Figure 1. Device driver and file access. There are four programs (A, B, C, and D) reading data (C), writing data (A, B, D) or both (C). The arrows specify the data flow direction.

In this operating system device driver architecture (Figure 1), it does not matter what real objects the programs are connected to. They access them with the same interface, the file system interface, regardless of how they are really implemented. The device driver is responsible for adapting the abstract file system interface to the real device.

Device classification

One device can be classified as physical (hardware connected) or soft (when two or more programs talk with each other using the file system interface). We are proposing a soft device, where we have the original BLAST on one side, and the BLAST data provider on the other side, where the buffer manager is implemented.

Writing Linux devices

In Linux, the target operating system for our project, to write a device we have to write a kernel module. A kernel module is a C program compiled and installed in the kernel. This program must use kernel-specific API functions (ANSI standard functions are not allowed), and it must follow specific kernel interfaces.

To write a device that has the same regular file behavior, we basically have to redefine I/O operations in our device implementation. The I/O functions that need to be redefined are: open, read, write, seek, and close.

To register a device, a function register_chrdev is called in a function init_module. A structure containing a pointer to file operation functions (callbacks) is used as a parameter. The callback functions implement the file system interface. We redefined open, read and write operations, in order to implement the BLAST device driver (Figure 2).

Once compiled and installed, the operating system assigns a unique identifier to the device. This identifier is used to link the device to a file system. We can create a file name to access the device. If one program opens this fake file, instead of accessing a regular file, the read and write operation will be redirected to the device read and write functions, which will decide what to do.

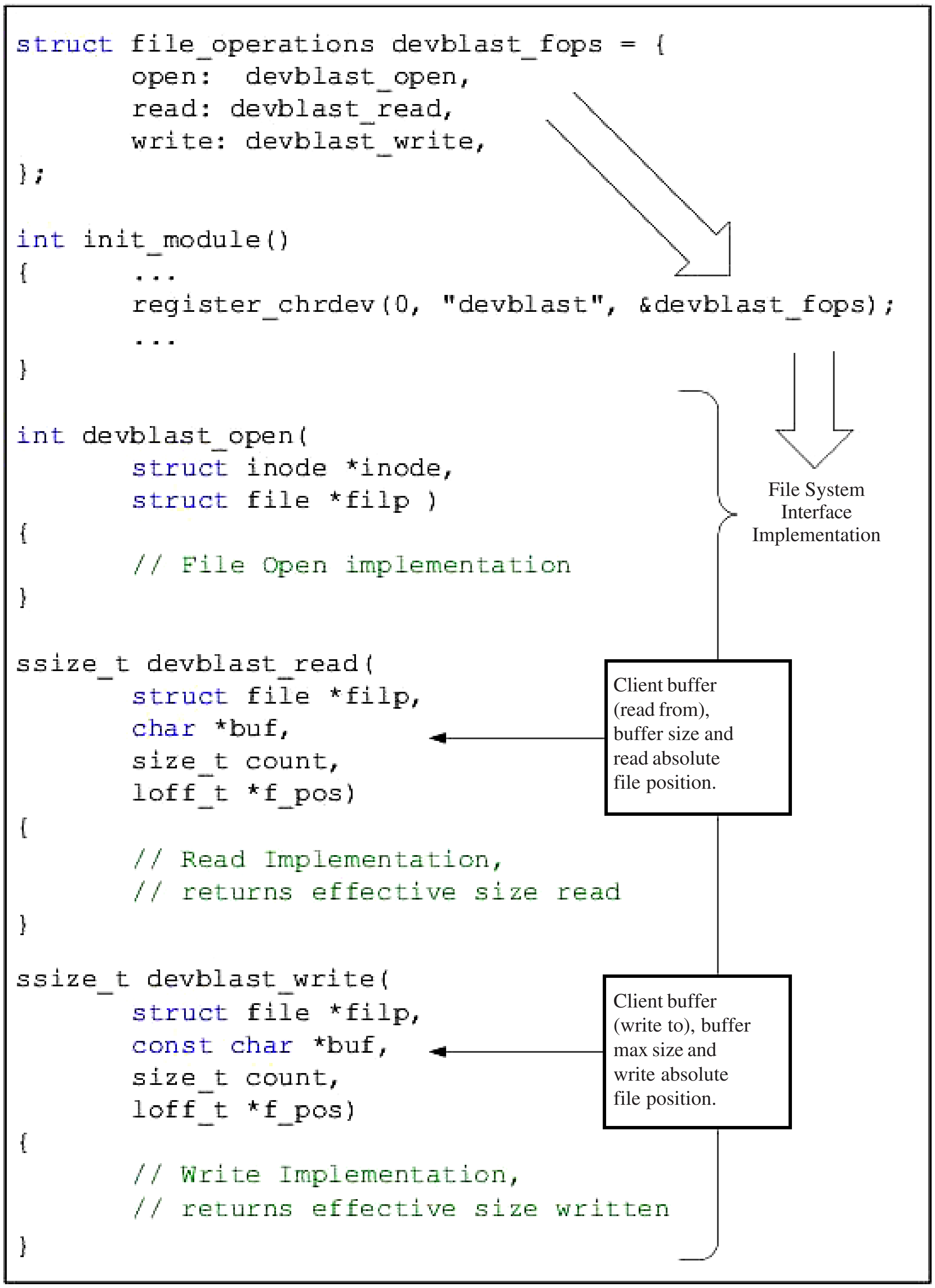
HOW BLAST WORKS

BLAST (Altschul et al., 1990) is an algorithm for comparing biological sequences, such as the amino-acid sequences of different proteins or the DNA sequences of different genes. Given a library or database of sequences, a BLAST search enables a researcher to look for sequences that either duplicate or resemble any sequence of interest.

BLAST is a sequence comparison algorithm optimized for speed used to search sequence databases for optimal local alignments to a query. The initial search is done for a word of length W that scores at least T when compared to the query, using a substitution matrix. Word hits are then extended in either direction in an attempt to generate an alignment with a score exceeding the threshold of S (NCBI, 2005).

BLAST works at least in two phases. In the first phase, a full scan is performed. In the second phase, the comparison is performed only against the sequences selected in the first phase. This means that it does not matter in what sequence the process starts. We could shuffle the sequences stored in the database and the final result should be the same: stored order is not important.

By knowing how BLAST reads data from a sequence database, we can develop a specific strategy to read data more efficiently than an operating system does. That is exactly what is proposed in Lemos and Lifschitz, 2003.



```
struct file_operations devblast_fops = {
        open:  devblast_open,
        read: devblast_read,
        write: devblast_write,
};

int init_module()
{       ...
        register_chrdev(0, "devblast", &devblast_fops);
        ...
}

int devblast_open(
        struct inode *inode,
        struct file *filp )
{
        // File Open implementation
}

ssize_t devblast_read(
        struct file *filp,
        char *buf,
        size_t count,
        loff_t *f_pos)
{
        // Read Implementation,
        // returns effective size read
}

ssize_t devblast_write(
        struct file *filp,
        const char *buf,
        size_t count,
        loff_t *f_pos)
{
        // Write Implementation,
        // returns effective size written
}
```

File System Interface Implementation
Client buffer (read from), buffer size and read absolute file position.
Client buffer (write to), buffer max size and write absolute file position.

Figure 2. Device driver basic structure.

A BLAST amino-acid database is composed basically of three files: the sequence file (.psq), the header file (.phr) and index (.pin). This last file assigns each sequence with its corresponding header information.

In our implementation, BLAST still reads from these files, but not directly. The file access is through our specialized device driver: the BLAST device driver. The real files are accessed by the provider module.

ARCHITECTURE AND IMPLEMENTATION

Architecture overview

In our solution (Figure 3) we want to use the original version of NCBI BLAST, without have to recompile the code. Although the current implementation only works with the BLAST database format, the main idea can be implemented for other BLAST flavors.

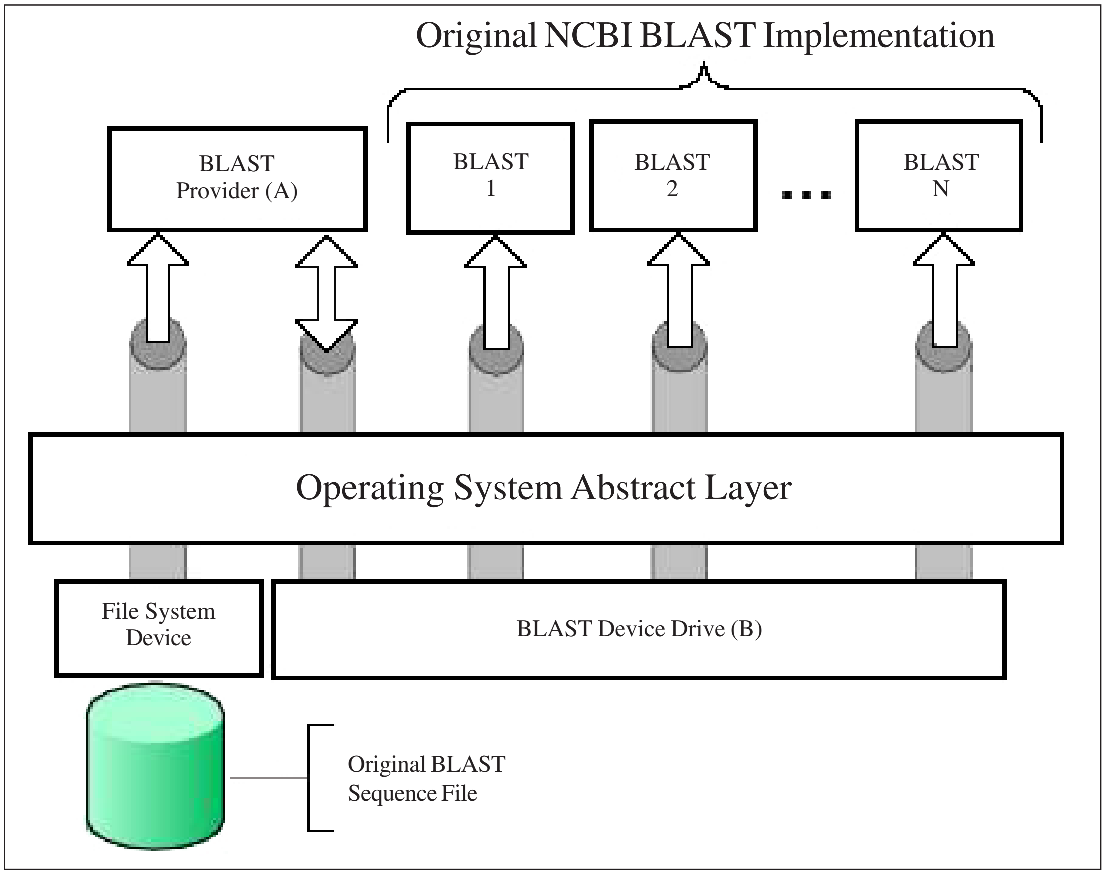

Original NCBI BLAST Implementation
BLAST Provider (A)
BLAST 1
BLAST 2
...
BLAST N
Operating System Abstract Layer
File System Device
BLAST Device Drive (B)
Original BLAST Sequence File

Figure 3. BLAST device driver architecture.

The original BLAST programs run as if they were reading from a conventional file. But they are reading in fact from the BLAST device driver. When one BLAST instance needs to read unavailable data, the device puts the process to sleep until the data arrive from the BLAST provider. The device sends a message to the provider, indicating that the process is sleeping because it needs a chunk of data. The provider knows what BLAST instance is sleeping and what file segment is needed for each one.

The provider decides which instances will wake up, by sending to the device a list containing a process identifier and the respective chunk of data to be sent to the BLAST process.

Implementation

This section summarizes the implementation strategies used to implement the BLAST device driver and the provider.

BLAST device driver

The BLAST device driver is a Linux kernel module that implements the operation system file system interface. The BLAST device driver is accessed by BLAST and by the provider through a special system name.

The first process that opens the device for write mode is elected as the provider. All others are considered BLAST clients. When the provider requests data from a device, it sends to the provider the current system state, which includes a list of sleeping processes and also what file offset each process needs. When a client opens the file linked to the device, the client process ID is inserted into the process state list, to be sent to the provider when required. The device has an internal buffer, where the data sent by provider are stored. When a client requests data, the device checks if data in the local buffer can satisfy the client. If so, the client is served immediately. If not, client goes to a wait queue.

Provider

When a provider module starts, it opens the original BLAST sequence file in a read-only mode and opens the pseudofile assigned to the device driver for read and write. The provider uses the read channel to capture information from the device, and it uses the write channel to send data to the device. The information captured from the device is implemented as a list of active BLAST processes that are sleeping, waiting for I/O. The provider module chooses which process it will serve and then sends data to the device.

Current device implementation chooses the process that has the smallest file offset request to serve. Our intention is to define an abstract decision mechanism, allowing programmers to define their own policy strategy.

CONCLUSIONS AND FUTURE WORK

We have developed and tested a specialized device driver for BLAST. The main advantage of our solution is implementation independence: it can be used with the original BLAST program. Based on preliminary tests, our architecture does not introduce overhead into BLAST processing, even if no buffer management policy is used. The next step of our research is to implement the buffer management policy proposed by Lemos and Lifschitz, 2003, comparing simulation results with real execution. We plan to evaluate this architecture using other BLAST implementations. We also plan to write an extensive tutorial of how to create buffer management, using our solution, in order to stimulate the database community to write specialized buffer management algorithms specialized for the BLAST program.

Our database research group has been publishing and developing solutions for bioinformatics, always based upon a database approach. These solutions include a framework for the genome database (Seibel and Lifschitz, 2001), the use of agents and data allocation techniques (Costa and Lifschitz, 2003) and buffer management strategies (Lemos and Lifschitz, 2003). This latter reference suggests a circular buffer algorithm. Simulations implementing this approach showed that the BLAST throughput could be increased by 50%.

The architecture that we propose was implemented to facilitate the buffer implementation proposed by Lemos and Lifschitz, 2003. Nevertheless, this solution can be used in other contexts than bioinformatics, contemplating tools that manipulate huge amounts of data without database support.

REFERENCES

Altschul, S.F., Gish, W., Miller, W., Myers, E.W. and **Lipman, D.J.** (1990). A basic local alignment search tool. *J. Mol. Biol. 215*: 403-410.

Costa, R.L.C. and **Lifschitz, S.** (2003). Database allocation strategies for parallel BLAST evaluation on clusters. *Distrib. Parallel Databases 13*: 99-127.

Lemos, M. and **Lifschitz, S.** (2003). A Study of a Multi-ring Buffer Management for BLAST. *Proceedings of the 1st International Workshop on Biological Data Management* (*BIDM*), pp. 5-9.

NCBI (2005). NCBI - National Center for Biotechnology Information. [http://www.ncbi.nlm.nih.gov]. Last accessed June 2005.

Seibel, L.F.B. and **Lifschitz, S.** (2001). A Genome Databases Framework. *Proceedings of the Database and Expert Systems* (*DEXA*) *Conference, Lecture Notes in Computer Science* (*LNCS*), Vol. 2113, pp. 319-329.

Tanenbaum, A.S. (2001). *Modern Operating Systems*. Prentice Hall Inc., Upper Saddle River, NJ, USA.

WU-BLAST (2005). Washington University BLAST Implementation. [http://blast.wustl.edu/]. Last accessed June 2005.