

Optimizing packet capture on symmetric multiprocessing machines

Original

Optimizing packet capture on symmetric multiprocessing machines / Degioanni, L.; Baldi, Mario; Risso, FULVIO GIOVANNI OTTAVIO; Varenni, G.. - STAMPA. - (2003), pp. 108-115. (15th IEEE Symposium on Computer Architecture and High Performance Computing (SBAC-PAD03) Sao Paolo (Brazil) Nov. 10-12, 2003) [10.1109/CAHPC.2003.1250328].

Availability:

This version is available at: 11583/1417047 since:

Publisher:

IEEE

Published

DOI:10.1109/CAHPC.2003.1250328

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

Optimizing Packet Capture on Symmetric Multiprocessing Machines

Gianluca Varenni, Mario Baldi, Loris Degioanni, Fulvio Rizzo

Dipartimento di Automatica e Informatica

Politecnico di Torino

Corso Duca degli Abruzzi, 24 – 10129 Torino, Italy

{gianluca.varenni, mario.baldi, loris.degioanni, fulvio.rizzo}@polito.it

Abstract

Traffic monitoring and analysis based on general purpose systems with high speed interfaces, such as Gigabit Ethernet and 10 Gigabit Ethernet, requires carefully designed software in order to achieve the needed performance. One approach to attain such a performance relies on deploying multiple processors. This work analyses some general issues in multiprocessor systems that are particularly critical in the context of packet capture and network monitoring applications. More important, a new algorithm is proposed to coordinate multiple producers concurrently accessing a shared buffer, which is instrumental in packet capture on symmetrical multiprocessor machines.

1. Introduction

Many of today's applications and communication systems rely on data networks. Consequently, seamless operation and proper performance of the network have come to affect everyone's daily business and personal life. This has led, in the last years, to a significant increase in network capacity and reliability and the need of tools to continuously monitor network behavior and performance. Generally speaking, such tools snoop packets being transferred through the network, possibly store (part of) them for later analysis, and collect various measurements and statistics.

Even though different solutions can be devised for specific applications, in general the critical (on-line) path of network monitoring and analysis includes the following operations, usually executed in the order shown below.

1. *Filtering*: each packet traveling on the network is matched against a set of rules in order to determine whether it is interesting for a particular application. Since the amount of traffic traveling on a network segment, especially in the network core, can be huge, filtering out irrelevant traffic is an essential step to

reduce the demand in terms of storage and processing power on the monitoring and analysis tool. Filtering usually implies extracting relevant fields from each packet and using their value to evaluate the rules. Filtering is a special case of *classification*[10] that is used by various networking functions to separate packets that need to be processed differently.

2. *Processing*: while traffic monitoring and analysis does not modify packets, a large number of counters used to provide measurements and statistics might have to be updated for each accepted packet¹.
3. *Storing*: packets or relevant parts of packets are stored in system memory or mass memory for later, off-line processing or inspection by a network manager. This operation is essential in packet capturing tools.

Considering the continuous increase in network traffic, performance is a key issue in network monitoring and analysis tools. One approach in addressing such issues relies on special-purpose hardware that assists in the execution of the above operations. Another approach focuses on optimizing the software implementations of the above operations so that their execution on general-purpose hardware can cope with the traffic at hand. The former approach leads to hardware monitoring and analysis tools, while the latter results in software tools.

The work presented in this paper was motivated by an effort to improve the performance of WinPcap [1][9], a public domain Windows library widely used world-wide for the implementation of applications that require low level access to network services. Monitoring and analysis tools, like the companion Analyzer [4], WinDump [5], and Ethereal [6], are examples of such applications. Even though this paper focuses on monitoring and analysis applications, the presented work is relevant also to a plethora of other applications, such as network address translation (NAT) engines, routers, firewalls, and

¹ It is worthwhile noticing that the work presented in this paper is relevant to many other network tools, such as network address translation (NAT) engines and firewalls that are more demanding in terms of per packet processing.

intrusion detection systems (IDS). In fact, WinPcap has been used in the implementation of several such systems.

This work reports the challenges and the outcome of the application to WinPcap of a basic and well-known approach to performance improvement of software systems: *parallel execution*. This work has been particularly challenging since traffic monitoring and analysis is inherently non-parallel because packets travel serially through the network links and are received in sequence by a single Network Interface Card (NIC). In Section 2 the above-presented critical operations are analyzed in more detail from both the logical and physical points of view. This analysis aims at finding the processing steps whose parallel execution can positively affect performance. Section 3 focuses on the most critical — from the point of view of parallel execution — of such steps: buffer management. A novel buffer management algorithm that allows *multiple producers* and multiple consumers to *concurrently* access a common buffer is presented and compared to well-known solutions enabling different degrees of parallel execution. The proposed buffer management algorithm is currently implemented in an internal version of WinPcap. Conclusions are drawn in Section 4.

2. A Closer Look at Traffic Monitoring and Analysis

This section gives a closer look at the operations involved in traffic monitoring and analysis to provide a model of the corresponding logical and physical system architecture. Since this work was done in an effort to improve the performance of WinPcap, the presented system architecture is closely related to the one of Windows NT-based operating systems.

2.1. Logical System Architecture

Figure 1 shows the logical system components involved in capturing network traffic. Packets received by a NIC are stored in a portion of the system memory (RAM) usually known as the *NIC driver buffer*. This operation is usually carried out directly by the NIC by means of a bus-mastering data transfer and without intervention of the system CPU.

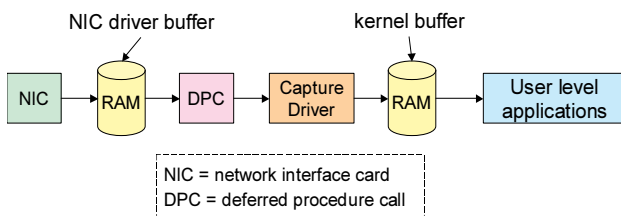


Figure 1. Logical architecture of a monitoring and analysis system.

The NIC signals the availability of new data in the NIC driver buffer through an interrupt request. In Windows NT-based systems interrupt requests are usually served by triggering the execution of a very short handling routine that does little more than queuing a procedure pointer for later execution. This mechanism is called deferred procedure call (DPC)[3][11]. The DPC is dequeued by the kernel when all the pending interrupt requests have been served, and it is responsible for calling the user-specific code (e.g. the capture driver) through an entry function that is usually called `tap()`[2]. The capture driver runs at kernel-level as well and it starts the custom processing. For instance, packets are compared against a set of rules that select the packet interesting for the application (*filtering*). If they match the rules, they are copied to another system memory location (called kernel or ring buffer), from where they are delivered to the application that can perform a further processing on them.

The motivation underlying this work is improving the performance in executing the code of the capture driver by means of the parallel execution capability of a symmetric multi-processing (SMP) machine. As shown in Figure 2, multiple processors could concurrently execute the same instance of the capture driver `tap()`, each one handling a different packet stored in the NIC driver buffer. In Windows NT-based SMP systems each CPU holds its own DPC queue, and, when it is not busy serving an interrupt request, picks a procedure from the head of its DPC queue and executes it. The possibility to execute concurrently two `tap()` functions brings a considerable advantage in case the amount of processing in it is high. The parallelization can be highly efficient in case the two execution paths do not have any dependencies among them (e.g. state variables).

After the `tap()` processing, both CPU must copy their data into the kernel buffer. To complete the advantages brought by the SMP processing, also the copies of the capture drivers toward the kernel buffer should be made more efficient. In this approach the kernel buffer is a shared resource concurrently accessed by multiple processes writing (capture drivers) and reading (user level applications) data. This is the well-known *producer and consumer* problem; a widely known algorithm addressing this problem is presented in Section 3.1, while some variations to increase the degree of parallelism among the producers (and the overall speed), are presented in the remainder of Section 3. These algorithms are able to further improving the parallelization of the whole packet processing path.

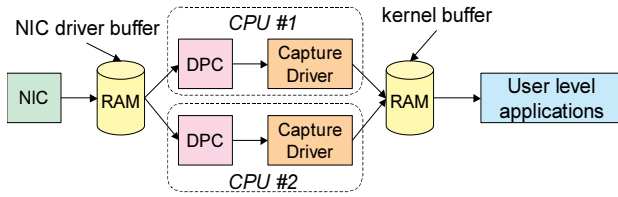


Figure 2. Logical architecture of a monitoring and analysis system with multiple processors.

2.2. Physical System Architecture

Being RAM shared among processors through the so-called *memory bus*, as shown in Figure 3, the various CPUs cannot concurrently access the NIC driver buffer and kernel buffer while executing the DPCs and the capture driver `tap()` function, as shown in Figure 2. As a consequence, the idea of improving performance by having multiple CPUs concurrently accessing the kernel buffer seems unfeasible since all memory accesses are serialized going through the BUS.

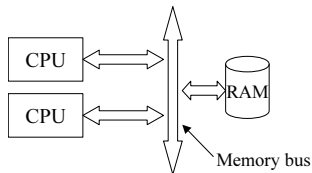


Figure 3. Logical access to memory.

Nevertheless, a closer look at computer architecture shows that physical transfer of data to and from memory involves a third element, the *cache memory*, which is local to each CPU, as shown in Figure 4. Consequently, even if the memory is not strictly accessible in parallel, in most cases the various CPUs actually work concurrently on a copy in their cache memory.

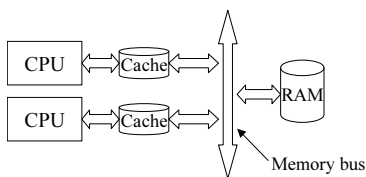


Figure 4. Physical access to memory.

Particularly, when the `tap()` executed on a CPU begins accessing a packet stored into RAM (in the NIC driver buffer), the packet is usually transferred immediately into the CPU cache memory. When the CPU then writes the packet to memory (in the kernel buffer), it is actually copied inside the CPU's cache and only later the memory bus controller, without intervention of the CPU, will actually write the data to RAM. If the processing into the capture driver lasts enough, the caches can transfer their content to the RAM (serially) while the

CPU are performing some new tasks (e.g. processing the next packets).

3. Algorithms

The producer-consumer problem is one of the best-known examples of synchronization among several execution paths sharing a common resource. In this problem, two (or more) entities communicate by means of a shared memory: one entity – the producer – writes data in the shared memory, while the other one – the consumer – retrieves data from the shared memory.

3.1. Single producer solution

Several solutions to this problem were proposed in the literature. One of the best known approaches [8] uses a ring buffer as shared memory, two pointers, P and C in Figure 5, to ensure proper handling of the information stored in the memory, and two semaphores to synchronize the access to it. For the sake of brevity in the following presentation of solutions to the producer-consumer problem, the ring buffer is considered organized in fixed size storage units, hereafter called cells. The presented algorithms can be easily generalized to variable size storage units.

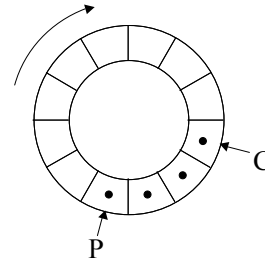


Figure 5. Ring buffer used for the single producer problem.

As shown by the pseudo-code in Figure 6, the producer waits on a semaphore `free` counting the number of free cells in the ring buffer. If the buffer is full the producer is blocked by the wait operation. Otherwise, the producer fills the cell pointed by P, modifies P to point to the next cell, and signals semaphore `occupied` counting the number of full cells in the buffer.

The consumer waits on semaphore `occupied` and is blocked if the buffer is empty. When the producer signals the semaphore `occupied`, the consumer wakes up, retrieves data from the cell pointed by C, updates C to point to the next cell, and signals semaphore `free`, possibly awaking the sleeping producer.

Semaphore `free` avoids overrunning the buffer since the producer can write some data only if there is at least one free cell in the ring buffer. Semaphore `occupied`

avoids underrunning the ring buffer since the consumer can read only if the ring buffer contains at least one full cell. However, this algorithm does not work in case of multiple producers.

```
free=BUF_SIZE;
occupied=0;

producer()
{
    wait(free);
    buffer[P]=produce(...);
    P=(P+1)%BUF_SIZE;
    signal(occupied);
    return SUCCESS;
}

consumer()
{
    wait(occupied);
    consume(stream[C]);
    C=(C+1)%BUF_SIZE;
    signal(free);
    return SUCCESS;
}
```

Figure 6. Single producer-consumer problem.

3.2. Pointer-based solutions with multiple producers

The most common, and straightforward, approach to the multiple producers problem consists in allowing only one producer at a time to write on the ring buffer. This can be achieved by including the buffer access code within a critical section, as shown in Figure 7².

```
free=BUF_SIZE;
occupied=0;
producers_semaphore=1;

producer()
{
    wait(producers_semaphore);
    wait(free);
    buffer[P]=produce(...);
    P=(P+1)%BUF_SIZE;
    signal(occupied);
    signal(producers_semaphore);
    return SUCCESS;
}

consumer()
{
    wait(occupied);
    consume(stream[C]);
    C=(C+1)%BUF_SIZE;
    signal(free);
    return SUCCESS;
}
```

Figure 7. Ring buffer write access within critical section.

The critical section is created through the use of a one-slot semaphore (`producers_semaphore` in Figure 7). This solution, although very simple, does not exploit parallel processing since only one producer can access the ring buffer at each given time. While this might be acceptable in general, it might significantly affect performance in packet capture where moving data from the NIC driver buffer to the kernel buffer represents a non-negligible fraction of the overall task performed by the system, as explained in Section 2.

The above algorithm can be extended to allow multiple producers to concurrently access a ring buffer, by substituting the pointer `P` with the following two pointers (see Figure 9):

- `PC`, (Consumer-head Pointer), identifying the last readable cell in the buffer
- `PP` (Producer-head Pointer), identifying the first writable cell in the buffer.

These two pointers and pointer `C` must always be consistent. A first step to maintain consistency is to make sure that every increment to these pointers be done atomically. This is usually obtained through functions provided by the operating system (such as `InterLockedXXX()` functions on Windows NT-based systems) or CPU-native atomic instructions.

```
free=BUF_SIZE;
producers_semaphore=1;

producer()
{
    wait(free);
    wait(producers_semaphore);
    myP= Pp;
    Pp= (Pp+1) % BUF_SIZE;
    signal(producers_semaphore);
    buffer[myP]= produce(...);

    // ...other code
    return SUCCESS;
}
```

Figure 8. Atomic increment of `PP`.

The pointer `PP` identifies the cell in which a new producer should deposit its data. In order to guarantee that concurrent producers do not access the same cell, `PP` must be atomically incremented by a producer, before it actually starts writing into the pointed cell, as exemplified in Figure 8.

² The pseudo-code examples presented in this paper assume a single consumer because the application on which the work focused — i.e., traffic monitoring and analysis — does not require multiple consumers. However, the extension to multiple consumers is straightforward.

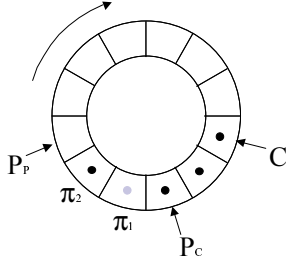


Figure 9. Pointers for the multiple producers case.

This solution is effective in synchronizing producers' access to the ring buffer so that cells are filled in an orderly fashion, without leaving empty cells between filled ones and without risk of overwriting. However, it is critical in the update of pointer P_c . Since the order in which producers finish their task is not known in advance, once done with its own cell a producer needs to know the status of the neighboring cells in order to decide whether it can update pointer P_c . For example, in the scenario depicted in Figure 9 the producer π_2 completes its task before producer π_1 , which, having started before, had obtained control of the preceding cell. Hence, upon completion, producer π_2 is not supposed to update P_c , while producer π_1 should update P_c to point to the cell filled out by π_2 , rather than its own.

Various algorithms and data structures for handling the update of P_c were proposed in [7] (one of them is included in the WinPcap 3.0 [1] implementation for SMP machines). The next Section presents a simpler and more effective approach that bypasses the criticality in the update of P_c by avoiding deploying such pointer.

3.3. The tagged cell algorithm

The algorithm presented in this Section deploys a *per-cell tag* that indicates its state. The consumer checks a cell's tag to determine whether the cell can be read. A pointer P contains the index of the first cell in which a producer can store its data.

The per-cell tag identifies one of the following states for the cell:

- *Free* — the cell is empty and can be written by producers; the consumer is not supposed to read it.
- *Booked* — the cell is booked by a producer before starting writing data to it. The cell cannot be read by the consumer or used by another producer.
- *Full* — the cell contains some valid data, thus can be safely read by a consumer. Producers cannot use this cell.
- *Padding* — a producer, after booking the cell, has failed to fill it. Other producers should not consider

this cell for writing, while consumers should process this cell by tagging it free without reading its content. The Finite State Machine (FSM) depicted in Figure 10 shows all the possible cell state transitions.

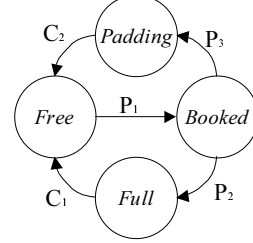


Figure 10. Per-cell tag FSM.

Initially, every cell is marked *Free*. A producer wishing to write data books a *Free* cell by marking it as *Booked* (transition P_1 in Figure 10). When the producer is done with a booked cell, it marks the cell as *Full* (transition P_2 in Figure 10). If the producer for any reason fails to complete its writing operation by storing valid data in the cell it booked, then it marks the cell as *Padding* (transition P_3 in Figure 10). In case the buffer is full, the consumer can either return failure (as it is in Figure 11) or block until a cell becomes *Free*. However, the former is the preferred solution: if no buffers are available, the received packet is simply lost.

A consumer can consider a cell only if it is in a *Full* or *Padding* state³; moreover, only a cell marked as *Full* contains valid data, hence a *Padding* cell must be skipped. When a consumer has processed a cell (whether a *Full* or *Padding* one), it marks the cell *Free* (transitions C_1 and C_2 in Figure 10, respectively).

Figure 11 shows the pseudo-code of the tagged cell algorithm: pointers P and C are used to identify the next cell to be considered for writing and for reading, respectively.

³ In case of a *Booked* cell, the consumer can either return a failure or block until the cell changes its state to *Full* or *Padding*; this is an implementation choice.

```

producer()
{
    lock(global_lock);
    if (buffer[P].flag == FREE)
    {
        buffer[P].flag = BOOKED; /* P1 transition */
        myP = P;
        P=(P+1)%BUF_SIZE;
    }
    else
        myP=-1;

    unlock(global_lock);

    if (myP == -1)
        return FAILURE_BUFFER_FULL;

    buffer[myP] = produce(..., retcode);

    lock(global_lock);

    if (ret_code == OK)
        buffer[myP].flag = FULL; /* P2 transition */
    else
        buffer[myP].flag = PADDING; /* P3 transition */

    unlock(global_lock);
    return retcode;
}

consumer()
{
    lock(global_lock);
    myflag = buffer[C].flag;
    unlock(global_lock);

    if (myflag == FREE || myflag == BOOKED)
        return FAILURE_EMPTY;

    if (myflag == FULL)
        consume(stream[C]);

    lock(global_lock);
    stream[C].flag = FREE; /* C1 or C2 transition */
    unlock(global_lock);

    C=(C+1)%BUF_SIZE;

    return SUCCESS;
}

```

Figure 11. Tag cell algorithm.

The pointer P and the cell tag, shared among the various producers, are atomically modified inside a critical section enclosed between `lock(global_lock)` and `unlock(global_lock)`.

The tagged cell algorithm has two major advantages over each of the pointer based solutions presented in Section 3.2.

- The producer uses two small critical sections, whose sole purpose is to have a global pointer atomically modified, while the producer code that actually writes data into the ring buffer — i.e., the function `produce(...)` in the presented pseudo-code example — is not in a critical section; therefore it can be concurrently executed by multiple producers.
- The coordination of producers and consumers is straightforward to understand and implement: it is a matter of atomic handling of the cell tags and pointers P_p and C .

On the other side, this solution presents two shortcomings, whose impact on the systems performance can however be considered negligible, especially when the algorithm is applied in the context of packet capture:

- when a producer does not successfully complete its writing procedure (in other words, when the function `produce(...)` in Figure 11 fails), the cell previously booked by the unsuccessful producer cannot be used by another producer until it has been processed by a consumer⁴. In the context of packet capture, the possibility of a producer being unable to successfully complete its task, while not null, is fairly low. Thus, this shortcoming can be neglected, i.e., the number of cells possibly being unused is negligible compared to the ring buffer size, and even more to the number of cells successfully filled up;
- if one or more *Booked* cells precede a *Full* cell, the latter cannot be accessed by a consumer until the respective producers are done with the *Booked* cells. Such a scenario is depicted in Figure 12: a consumer accessing the buffer is given a buffer empty notification even though the buffer contains a cell ready to be read.

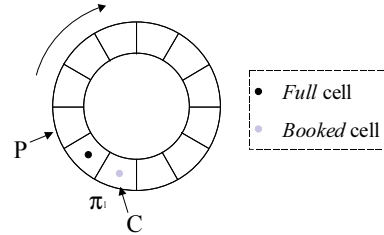


Figure 12. Blocked consumer.

In the context of packet capture, the last shortcoming is not an important issue because packet should be delivered to the application in the same order they are received by the NIC; solving that issue will arise the problem of the out of order packets. However, in case this issue matters, we should take into account that the difference in the execution time of the producers is usually small. Consequently, a situation like the one depicted in Figure 12 has a limited time span, which implies that (i) there is a low possibility that a consumer checks pointer C in such a short span, and (ii) a blocked consumer is waken up shortly. Moreover, in packet monitoring and analysis applications the speed at which the kernel buffer is emptied is not critical due to the large buffer size, needed to absorb large network bursts.

⁴ The cell state *Padding* is used to handle this situation.

3.4. General issues in multiprocessor systems

This section discusses some issues that in general exist in SMP systems, but are particularly relevant to traffic monitoring and analysis applications.

1) Preserving data order

In the realm of packet capture the problem of preserving the arrival order of the packets is a fundamental issue, particularly when dealing with multiple concurrent producers. Packets travel on a link serially and are received by all the NICs in the same order; thus it is important that packets are delivered to the application in the same order they have been received by the NIC. However, as shown in Figure 1 and Figure 2, received packets are stored in the NIC driver buffer, while packets being processed are retrieved from the kernel buffer. Consequently, it is essential that the order of packets is maintained in the transfer from the NIC driver buffer to the kernel buffer. This issue will be analyzed in detail for the tagged cell algorithm. However, the same considerations can be applied to the multiple producers algorithms presented in Section 3.

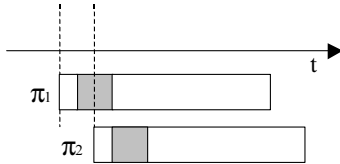


Figure 13. The booking procedure.

In the tagged cell algorithm the event of marking a cell *Booked* determines the order of cell allocation. Figure 13 shows a time diagram of the execution of two producers; the shaded area represents the execution of the booking procedure that takes place shortly after the producer starts. In the context of packet capture, a producer is executed as a `tap()` (triggered by the DPC) handling a specific received packet. Since DPCs are queued in the order the corresponding packets are received, it could be concluded that the cells of the kernel buffer are filled up in the order producers begin their execution. Instead, this is not the case of Windows NT-based SMP systems for the following reasons.

- Consider a scenario in which the DPC intended to handle a received packet is appended to the DPC queue of processor Π_1 , while the DPC of a subsequently received packet is appended to the queue of processor Π_2 . If Π_2 is available for the execution of its DPC before Π_1 , Π_2 will execute the producer code earlier and mark as *Booked* a cell preceding the one marked by Π_1 . Consequently, the two packets will not be stored in the ring buffer (i.e., the kernel buffer) in the order they were received.

- Even if DPCs are executed in the same order their corresponding packets were received (as it is the case, for example, if the processors are idle waiting to process received packets), the execution can be interrupted ("preempted") for servicing an interrupt request. As shown in Figure 14, if the execution of a producer π_1 on processor Π_1 is preempted before it marks a cell as *Booked*, a producer π_2 started on processor Π_2 after π_1 , could execute the marking instruction before π_1 . Consequently, the first received packet handled by π_1 will be stored in the kernel buffer after the subsequently received packet handled by π_2 .

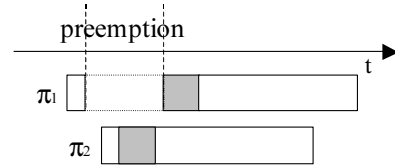


Figure 14. Booking procedure with preemption.

It is opinion of the authors that a non zero possibility of not preserving packet order can be accepted as a trade-off in obtaining high performance over SMP machines running a general purpose operating system. In any case, some of the aspects of the problem depend on the operating system, which is not under our control.

2) Synchronization time

The time spent by concurrent processes waiting for each other includes the contributions listed below. In the context of high-speed packet capture these contributions might make synchronization time a serious issue.

- Synchronization primitives are usually slow, particularly on multiprocessor machines. Informal test results have shown that with Windows NT-based operating systems a synchronization primitive is up to 10 times slower on a multiprocessor machine than on a single processor one. This applies to both kernel (e.g., spinlocks) and hardware (e.g., `InterlockedXXX()` function) synchronization primitives.
- The worst-case time spent waiting to enter a critical section is $O[(n-1) \cdot t_s]$, where n is the maximum number of processes trying to enter the same critical section (e.g., the number of producers in the problem addressed in this work), and t_s is the total amount of time spent in the critical section. Figure 15 shows an example of such worst case situation: producer π_3 is ready for the execution of the critical section slightly after π_1 , but it has to wait almost twice the critical section execution time ($t'_A - t_A$) before being allowed into it. Even though the critical section of the tagged cell algorithms encompasses a small number of instructions —hence featuring a short execution

time— when the number of processors in the system is high and the captured traffic is intense, the time spent while waiting to modify a cell tag can be non negligible.

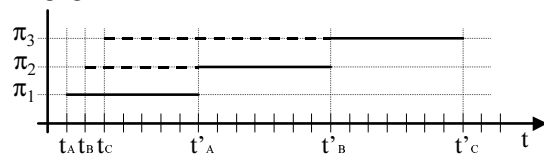


Figure 15. Time spent waiting to enter a critical section.

4. Conclusions

The work presented here was performed in the context of development of WinPcap [1], a public domain library for low-level access to network functionalities on Windows systems. Being traffic monitoring and analysis the main applications of this library (see [4][5][6]), high performance is essential to ensure proper operation with high speed interfaces, such as Gigabit Ethernet. Parallel execution of packet capture on SMP systems seems to be a possible, promising approach to high performance. However, packet capture is an inherently sequential task and the code implementing it must be carefully designed in order to take significant advantage of multiple processors without any packet reordering. Since several algorithms do not scale well with the number of processors, having multiple CPUs does not necessarily imply better performance.

The presented work has two main contributions. First, a novel algorithm for the solution of the producer-consumer problem when multiple producers *concurrently* access the shared buffer is presented. Second, some general issues in multiprocessor systems are analyzed in the context of packet capture applications.

Independently of its application in the field of high performance packet capture, the presented solution for the multiple concurrent producers problem constitutes a general, relevant theoretical contribution. Further work will include its performance evaluation also in the context of Non-Uniform Memory Architecture (NUMA) workstations, which offer a great degree of scalability because their performance are not limited by the shared bus and RAM memory.

References

[1] Computer Networks Group (NetGroup) at Politecnico di Torino, “WinPcap Web Site”, available at <http://winpcap.polito.it>, April 2003.

[2] S. McCanne, and V. Jacobson, “The BPF Packet Filter: A New Architecture for User-level Packet Capture”, *Proceedings of 1993 Winter USENIX Technical Conference* (San Diego, CA, Jan. 1993), USENIX.

[3] D. Solomon, and M. Russinovich, *Inside Windows 2000*, 3rd ed., 2000, Microsoft Press.

[4] Computer Networks Group (NetGroup) at Politecnico di Torino, “Analyzer Web Site”, available at <http://analyzer.polito.it>, March 2003.

[5] Computer Networks Group (NetGroup) at Politecnico di Torino, “WinDump Web Site”, available at <http://windump.polito.it>, March 2003.

[6] “Ethereal Web Site”, available at <http://www.ethereal.com>, March 2003.

[7] G. Varenni, “Approaches for the n-producers/1-consumer problem”, Technical Report DAUIN200302, Dipartimento di Automatica e Informatica Politecnico di Torino, Italy, Feb. 2003.

[8] A. Silberschatz, P. G. Gavin, and G. Gagne, *Operating System Concepts*, 6th ed., 2001, John Wiley & Sons.

[9] F. Risso, and L. Degioanni, “An Architecture for High Performance Network Analysis”, *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001)*, Hammamet, Tunisia, July 2001.

[10] P. Gupta and N. McKeown, “Algorithms for Packet Classification”, *IEEE Network Special Issue*, March/April 2001, vol. 15, no. 2, pp. 24-32.

[11] Microsoft Windows Driver Development Kits (DDKs), available at <http://www.microsoft.com/ddk/>.