The DragonFlyBSD Operating System

Jeffrey M. Hsu, *Member, FreeBSD and DragonFlyBSD*

***Abstract*— The DragonFlyBSD operating system is a fork of the highly successful FreeBSD operating system. Its goals are to maintain the high quality and performance of the FreeBSD 4 branch, while exploiting new concepts to further improve performance and stability. In this paper, we discuss the motivation for a new BSD operating system, new concepts being explored in the BSD context, the software infrastructure put in place to explore these concepts, and their application to the network subsystem in particular.**

***Index Terms*— Message passing, Multiprocessing, Network operating systems, Protocols, System software.**

I. Introduction

THE DragonFlyBSD operating system is a fork of the highly successful FreeBSD operating system. Its goals are to maintain the high quality and performance of the FreeBSD 4 branch, while exploring new concepts to further improve performance and stability. It departs from its predecessor in a number of ways, most notably, in place of the symmetric multiprocessing support being added to the upcoming FreeBSD 5 branch, DragonFlyBSD uses the concepts of partitioning and replication layered on top of a message passing system to implement lock-free scalability on symmetric as well as non-symmetric NUMA multiprocessors. The lightweight message passing system is also used as the basis for user-land messaging. This allows for a degree of extensibility and application-specific customization not possible in traditional monolithic kernels. In this paper, we discuss the motivation for a new BSD operating system, new concepts, the software infrastructure put in place to explore these concepts, and their application to the network subsystem in particular.

II. Project overview

The DragonFlyBSD project was started by Mathew Dillon and announced in July of 2003. It is currently comprised of 16 committers with write privileges to the source repository and a community of outside contributors who submit bug patches and help with testing. The source base was forked off the RELENG_4 branch of FreeBSD on June 16, 2003 and consists of 19,037 source files spread out among 2122 directories with slightly over 8 million lines of code, 2 million of which are in the kernel.

The project has a number of resources available to the public, including an on-line CVS repository with mirror sites, accessible through the web as well as the cvsup service, mailing list forums, and a bug submission system.

III. Motivation

A. Technical Goals

The DragonFlyBSD operating system has several long-range technical goals that it hopes to accomplish within the next few years. The first goal is to add lightweight threads to the BSD kernel. These threads are lightweight in the sense that, while user processes have an associated thread and a process context, kernel processes are pure threads with no process context. The threading model makes several guarantees with respect to scheduling to ensure high performance and simplify reasoning about concurrency. We shall describe the lightweight kernel threading model more fully later on.

Another technical goal is to implement a message passing system for use both within the kernel as well as between kernel and user land threads. System calls can then be implemented as messages and new application-specific system calls can be added easily. The message passing system is also integral to future plans for clustering and single system image (SSI) support.

The next goal is to add multiprocessor support to the kernel using a thread serialization paradigm where resources are owned by a particular processor and messages passed to perform an operation on that resource. Through judicious application of partitioning and replication along with lock-free synchronization techniques, we believe we can achieve greater scalability as the number of processors increase than a system that has locking overhead and contention for shared resources.

From an implementation point of view, the multiprocessing work requires a cleanup in the existing code base of old assumptions and dependencies that no longer hold in an MP environment, such as being able to access the current running process. We want to go further along these lines and decouple some of the dependencies in the I/O subsystem on the current address space, generalizing them to work with virtual memory objects instead. The end goal is to be able to run device drivers and large subsystems such as the VFS layer in user-land. This facilitates development of new drivers and filesystems. When application-specific customizations are factored in, running in user-land can potentially result in an

Manuscript received March 11, 2004.

Jeffrey M. Hsu is with the FreeBSD and DragonFlyBSD projects, California, 95135 USA (phone: 408-270-6175; fax: 408-528-5773; e-mail: hsu@ freebsd.org or hsu@dragonflybsd.org).

increase, rather than decrease, in end-application performance.

Also along the theme of pushing functionality out to user-land, DragonFlyBSD will implement a threads package where thread scheduling is performed in user-land. Kernel support in the form of shared memory regions and batched message passing will allow similar efficiencies to the Scheduler Activations model [10] adopted by earlier versions of Solaris and by FreeBSD 5.

B. *Efficiency through scale*

The effects of size on an organization are well-noted in many diverse fields [1][7]. For example, as a corporation gets bigger, it is no longer able to move as nimbly and introduce new products in new markets as quickly as its smaller counterparts [1]. In classroom settings, teachers know well that smaller class sizes lead to more class interaction. The FreeBSD organization has grown tremendously and rapidly in its successful 10 year run --- so much so that it now suffers from many of the same symptoms that large organization do. By reorganizing in a smaller group, we aim to improve interaction, to exchange ideas more freely, and to recapture the rapid pace of innovation that FreeBSD had in its earlier days.

IV. Concepts

A. *Non-uniform Memory Access*

The illusion of the symmetric multiprocessor (SMP) hardware model where memory access costs are uniform throughout the memory space that hardware designers have implemented in the past for the benefit of systems people to simplify system software is getting harder to maintain as processor speeds continue to increase faster than memory or I/O bus speeds. In fact, already the Intel Itanium and the AMD 64-bit processors are decidedly NUMA in nature.

The SMP software model where any processor can field any interrupt and any processor can run any available process has severe cache performance penalties when run on a NUMA architecture because modern CPUs only run well out of cache. SMP systems usually have to resort to some sort of scheduler modifications to gain cache affinity and run reasonably well.

Rather than starting with a SMP viewpoint and then trying to match it to NUMA reality, DragonFlyBSD starts out with a NUMA-centric view of the world and explicitly partitions the workload among multiple processors. For threads, this show up in the form of guarantees that a running thread will never be pre-empted by another processor nor will it ever be pre-empted by a non-interrupt thread. This means code can be written to effectively utilize and cache per-cpu global data without obtaining any locks. We shall see later on how the network subsystem takes advantage of this to explicitly partition TCP connections among multiple processors.

B. *Partitioning and replication*

DragonFlyBSD adopts a similar approach to the IBM K42 research operating system [2] in preferring the techniques of partitioning and replication along with lock-free synchronization techniques [3][27] over mutex locking and other traditional forms of SMP concurrency control. Unlike K42 which was written in C++, provides a Linux application environment [4], and only runs on 64-bit processor architectures, DragonFlyBSD applies these techniques directly to a BSD kernel running on both 32-bit as well as 64-bit processor architectures.

C. *Application-specific customization*

Much of the OS research for the past 10 years has dealt with application-specific customizations [5][6] and the huge performance benefits associated with closer integration between user-land and OS facilities. For example, by exploiting extensibility, the Exokernel project found in [11] a 2x improvement in web-server performance. For a filesystem implemented in user-land, it found no performance degradation on most operations and even a 4x improvement on one operation. To allow for a similar degree of application-specific customization, DragonFlyBSD plans to export the message passing facility to user-land. Because both kernel and user-land threads are based on the same underlying LWKT infrastructure, there is no appreciable difference between passing a message to a user-land thread versus a kernel thread.

V. Infrastructure

A. *Lightweight Kernel Threads*

The Lightweight Kernel Threads (LWKT) system decouples the traditional Unix notions of an execution context from a VM address space. This is similar to what many other systems such as Solaris [8] and FreeBSD [9] have done as part of their MP and threads support. The API for the LWKT system is shown in Table I.

The LWKT system has a number of features designed to remove or reduce contention between processors:

- Each processor has its own self-contained threads scheduler. Threads are tied to a processor and can only move under special circumstances.
- A thread can only be pre-empted by an interrupt thread. Both fast interrupts, where the interrupt is handled in the current thread context, and threaded interrupts, where the LWKT scheduler switches to the interrupt thread and back when it's done, are supported by the LWKT system.

Cross-processor scheduling is implemented via asynchronous inter-processor interrupts. Because these messages can be batched for a given interrupt, the system exhibits graceful degradation under load.

A lot of work went into separating the LWKT scheduler from the user process scheduler. The LKWT thread scheduler is MP-safe and utilizes a fast per-cpu fixed-priority round-robin scheme with a well-defined API for communicating

TABLE I
LWKT API

Function	Description
lwkt_alloc_thread	Allocate a new thread on cpu. Called by lwkt_create(). thread_t lwkt_alloc_thread(struct thread *td, int cpu)
lwkt_free_thread	Deallocates thread. void lwkt_free_thread(thread_t td)
lwkt_create	Create a new thread. int lwkt_create(void (*func)(void *), void *arg, struct thread **tdp, thread_t template, int tdflags, int cpu, const char *fmt, ...)
lwkt_exit	Destroy a thread. void lwkt_exit(void)
lwkt_switch	Switch to the next runnable thread on this processor. void lwkt_switch(void
lwkt_yield	Yield to equal or higher priority threads. void lwkt_yield(void)
lwkt_block	Block on the specified wait queue until signaled. void lwkt_block(lwkt_wait_t w, const char *wmesg)
lwkt_signal	Signal a wait queue. void lwkt_signal(lwkt_wait_t w, int gencnt)
crit_enter	Raising a thread's priority above the highest possible interrupting priority. Synchronous switching and blocking are allowed while in a critical section. void crit_enter(void)
crit_exit	Leave critical section. void crit_exit(void)

with other processors' schedulers. The traditional BSD multi-level scheduler is implemented on top of the threads scheduler. Additionally, the LWKT subsystem provides a clean API for implementing alternative user process schedulers if desired.

B. *Message passing system*

The message passing system is comprised of ports on which threads send and receive messages. The API for the message passing system is shown in Table II.

Cross-processor message passing is currently implemented with a software crossbar switch and a lock-free ring buffer. This extends the lock-free path all the way down from the kernel subsystems to the messaging layer.

TABLE II
MESSAGING API

Function	Description
lwkt_initmsg	Initializes *msg* with command. The reply port is set to the message port for the current thread. void lwkt_initmsg(lwkt_msg_t msg, int cmd)
lwkt_domsg	Send a message synchronously. void lwkt_sendmsg(lwkt_port_t port, lwkt_msg_t msg)
lwkt_sendmsg	Send a message asynchronously. int lwkt_domsg(lwkt_port_t port, lwkt_msg_t msg)
lwkt_replymsg	Sends a reply to the message. void lwkt_replymsg(lwkt_msg_t msg, int error)
lwkt_getport	Retrieve the next message from the port's message queue, return NULL if no messages are pending. The calling thread must own the port. void lwkt_getport(lwkt_port_t port)
lwkt_waitport	Waits for a message to arrive on a port. void *lwkt_waitport(lwkt_port_t port, lwkt_msg_t msg)
lwkt_initport	Initialize a port for use and assign it to the specified thread. void lwkt_initport(lwkt_port_t port, thread_t td)
lwkt_initmsg_rp	Initializes *msg* with a port and command. void lwkt_initmsg_rp(lwkt_msg_t msg, lwkt_port_t rport, int cmd)
lwkt_reinitmsg	The message flags are all cleared except for MSGF_ASYNC, which retains the old setting, and MSGF_DONE, which is always set. The message port is reset to the passed in value. void lwkt_reinitmsg(lwkt_msg_t msg, lwkt_port_t rport)
lwkt_beginmsg	Puts the message onto the port. int lwkt_beginmsg(lwkt_port_t port, lwkt_msg_t msg)
lwkt_forwardmsg	Puts the message onto the port. int lwkt_forwardmsg(lwkt_port_t port, lwkt_msg_t msg)
lwkt_waitmsg	Waits for reply to message. int lwkt_waitmsg(lwkt_msg_t msg)
lwkt_abortmsg	Aborts the message. void lwkt_abortmsg(lwkt_msg_t msg)

VI. Network subsystem

Previous efforts on speeding up networking can be classified into two categories, algorithmic enhancements to the network protocols and advantages accrued from implementation details, whether in hardware or software or from some special interaction among the two. Under the first category of protocol enhancements, DragonFlyBSD has incorporated some of the more recent networking enhancements such as the NewReno [16] set of corrections, larger congestion window sizes on connection startup [17], Eifel detection for spurious retransmits [18], right-edge recovery, also known as Limited Transmit [19], and early retransmit of lost data [20]. Collectively, these greatly improve network performance under a range of real-life network situations. For example, Balakrishnan et al. found in [30] that under the conditions where Limited Transmit applies, NewReno plus Limited Transmit is more than twice as fast as SACK. Additional algorithmic improvements are planned for high-speed networking as they come down the standards track [21][22][23][24].

On the implementation side, DragonFlyBSD has made a number of improvements to clean up the code and speed up network operations. For example, UDP transmissions no longer need to do a temporary pseudo-connect, a major bottleneck which Partridge had identified to be one third of the cost of UDP transmissions [28][29]. TCP connection setup through the syncache no longer does a number of relatively expensive processor priority level changes and some unnecessary allocation failure checks and their attendant recovery code have been removed. DragonFlyBSD also takes full advantage of hardware support when available. For example, it supports segmenting TCP packets in hardware[1]. But perhaps the most radical work has been done on taking advantage of multiple processors.

The work to distribute network processing across multiple processors was carried out in several stages. The first stage was to create per-protocol handling threads. While the run-to-completion style of protocol processing [12][13] may have been in vogue several years ago, now, the key to getting modern processors to run fast is definitely I-cache footprint [14] and sending messages to protocol threads allows for a form of cohort scheduling [15] where protocol processing occurs in batches, helping to improve cache locality.

So, the network subsystem uses messaging extensively. The bottom half of the kernel, the interrupt thread, sends a message to the protocol thread to hand off an incoming packet. From the top-half of the kernel, system calls made by user processes are turned into messages and dispatched to the appropriate protocol thread. Because the top-half and the bottom-half of the kernel use the same method for selecting which protocol thread to dispatch a request to, access to a given connection is effectively serialized by this process and no locks are required for synchronization. In addition, because a protocol thread is started on each processor for TCP, two or more TCP connections can be processed in parallel. Furthermore, Selahi found in [31] that this form of parallelism scales better than the SMP locking approach as the number of processor increases.

Protocol thread management is delegated to the individual protocol module, so the rest of the kernel and networking stack does not need to know how many threads nor which processor those threads were bound to. There is a function pointer that the generic network dispatch routine calls to determine which protocol thread to dispatch a packet to. Figure I shows the code for the dispatch function for the IP protocol. The generic network dispatch code calls this routine on receipt of an IP packet. Note how each protocol has control over which type of traffic it wishes to distribute to multiple processors and how. The UDP dispatch function currently uses the same demultiplexing logic as TCP for non-multicast UDP packets, but it could just as readily do round-robin, which would implement the hybrid IPS strategy described by Selahi in [31].

```
static __inline int
INP_MPORT_HASH(in_addr_t src, in_addr_t dst, in_port_t sport,
in_port_t dport)
{
    return ((src ^ sport ^ dst ^ dport) & ncpus2_mask);
}

/*
 * Map a packet to a protocol processing thread.
 */
lwkt_port_t
ip_mport(struct mbuf *m)
{
    struct ip *ip = mtod(m, struct ip *);
    int iphlen;
    struct tcphdr *th;
    struct udphdr *uh;
    lwkt_port_t port;
    int cpu;

    iphlen = ip->ip_hl << 2;

    switch (ip->ip_p) {
    case IPPROTO_TCP:
        th = (struct tcphdr *)((caddr_t)ip + iphlen);
        cpu = INP_MPORT_HASH(ip->ip_src.s_addr, ip->ip_dst.s_addr,
            th->th_sport, th->th_dport);
        port = &tcp_thread[cpu].td_msgport;
        break;
    case IPPROTO_UDP:
        uh = (struct udphdr *)((caddr_t)ip + iphlen);
        if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr)) ||
            in_broadcast(ip->ip_dst, m->m_pkthdr.rcvif)) {
            cpu = 0;  /* multicast data structures not parallelized yet */
        } else {
            cpu = INP_MPORT_HASH(ip->ip_src.s_addr,
                ip->ip_dst.s_addr, uh->uh_sport, uh->uh_dport);
        }
        port = &udp_thread[cpu].td_msgport;
        break;
    default:
        port = &netisr_cpu[0].td_msgport;
        break;
    }

    return (port);
}
```

FIGURE I

Thread creation is done during protocol initialization and each protocol has control over how many threads it wishes to create. At the moment, due to time constraints and prioritization of developer resources, DragonFlyBSD only creates multiple protocol processing thread for the widely

[1] "The performance gains offered by TCP Segmentation Offload (TSO) were so substantial in the Microsoft operating system that Intel took advantage of them in the Linux environment. … Intel has found a 60% reduction in CPU utilization and an increase in throughput." [32]

used TCP and UDP protocols. However, as illustrated by Figure I, the design allows for incremental deployment of MP support, so other protocols can readily be distributed as needed. This is in stark contrast to the all-or-nothing requirement enforced by the SMP locking approach [33].

VII. Status

Much of the infrastructure, including the lightweight threads and messaging system, has been completed, and is in use throughout the rest of the system. Additional facilities such as the user-land messaging are close to completion --- an early form has been committed to the source repository.

Work is progressing on the individual subsystems. A new name cache has been written. Portions of the device driver framework have been converted to message passing style. The network stack already distributes packets to protocol threads running on multiple processors, from both the lower-half interrupt handler and the upper-half system call handling code.

The system is available today from the project web site and its mirror web sites. It runs reliably, and, in preliminary testing, performs comparably to its FreeBSD 4 predecessor and noticeably better than FreeBSD 5. The first formal release of DragonFlyBSD is scheduled for June of 2004.

VIII. Conclusion

The DragonFlyBSD operating system applies recent concepts in operating system research to the FreeBSD 4 operating system. By building base infrastructure facilities and through careful redesign of the API between traditional kernel modules, we hope to bring the advantages of multiprocessing and other more recent advances to BSD, while maintaining or improving upon the stability and high performance of the traditional monolithic kernel.

Acknowledgment

The base LWKT and message passing architecture for DragonFlyBSD was conceived by Matthew Dillon based on his previous experience working on a number of operating systems. Matt also implemented the underlying infrastructure discussed in this paper and provided helpful comments on an earlier draft.

References

[1] C. M. Christensen, The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail (Management of Innovation and Change Series). Harvard Business School Press, June 1997.

[2] J. Appavoo, K. Hui, M. Stumm, et al. (2002, August). *K42 Overview*. IBM Research. Available: http://www.research.ibm.com/K42

[3] P. McKenney, J. Appavoo, A. Kleen, et al, "Read-Copy Update," *Ottawa Linux Symposium*, July 2001.

[4] J. Appavoo, M. Auslander, D. Da Silva, et al, "Providing a Linux API on the Scalable K42 Kernel," *Proceedings of Freenix 2003*, pp. 323-326.

[5] D. Engler, M. Kaashoek, and J. O'Toole Jr, "Exokernel: an operating system architecture for application-level resource management," *ACM Symposium on Operating System Principles,* vol 29, 3-6 December 1995.

[6] B. Bershad, S. Savage, P. Pardyak, et al, "Extensibility, Safety and Performance in the SPIN Operating System," *Proceedings of the 15th ACM Symposium on Operating System Principles,* pp. 267-284.

[7] M. Elvin, The Pattern of the Chinese Past. Stanford: CA, Stanford University Press, 1973.

[8] J. Mauro and R. McDougall, Solaris Internals: Core Kernel Architecture. Palo Alto: CA. USA. Prentice Hall, 2001.

[9] J. Evans and J. Elischer, (2000). Kernel-Scheduled Entities for FreeBSD. Available: http://www.freebsd.org/kse

[10] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, "Effective Kernel Support for the User-level Management of Parallelism," *ACM Transactions on Computer Systems,* vol 10, February 1992, pp. 53-79.

[11] M. Kaashoek, D. Engler, G. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie, "Application Performance and Flexibility on Exokernel Systems," *ACM Symposium on Operating System Principles,* October 1997.

[12] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsytem Architecture for Server Systems," *Proceedings of the 2nd Symposium on OS Design and Implementation (OSDI '96),* 1996.

[13] J. Mogul, "Eliminating Receive Livelock in an Interrupt-driven Kernel," *Proceedings of the 1996 Usenix Technical Conference*, pp. 99-111, 1996.

[14] D. Mosberger, L.L. Peterson, P.G. Bridges, and S. O'Mallery, "Analysis of Techniques to Improve Protocol Latency," *Proceedings of Sigcomm 96*, August 1996.

[15] J. Larus and M. Parkes, "Using Cohort Scheduling to Enhance Server Performance," *Proceedings of the 2002 Usenix Technical Conference,* pp. 103-114, 2002.

[16] S. Floyd, The NewReno "Modification to TCP's Fast Recovery Algorithm," RFC2582, April 1999.

[17] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's Initial Window," RFC 2414, Sept 1998.

[18] R. Ludwig and M. Meyer, "The Eifel Detection Algorithm for TCP," RFC3522, April 2003.

[19] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit," RFC 3042, January 2001.

[20] M. Allman, U. Ayesta, and J. Blanton, "Early Retransmit for TCP and SCTP," draft-allman-tcp-early-rexmt-02.txt, August 2003.

[21] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgement Options," RFC 2018, October 1996.

[22] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP," RFC 2883, July 2000.

[23] S. Floyd, "HighSpeed TCP for Large Congestion Windows," RFC 3649, December 2003.

[24] S. Floyd, "Limited Slow-Start for TCP with Large Congestion Windows," draft-ietf-tsvwg-slowstart-00.txt, July 2003.

[25] M. Dillon, "DragonFly - I/O Device Oprations." Available: http://www.dragonflybsd.org:/goals/iomodel.cgi

[26] M. Dillon, "DragonFly - VFS/filesystem Device Operations." Available: http://www.dragonflybsd.org:/goals/vfsmodel.cgi

[27] M. Greenwald and D. Cheriton, The Synergy Between Non-blocking Synchronization and Operating System Structure, *Usenix 2nd Symposium on OS Design and Implementation (OSDI' 96),* Seattle: WA USA, October 1996.

[28] G. Wright and R. Stevens, TCP/IP Illustrated: The Implementation, Volume 2, chapter 23, section 6, p. 763, Addison-Wesley, January 1995.

[29] C. Partridge and S. Pink, "A Faster UDP," *IEEE/ACM Trans. on Networking*, Vol. 1, No. 4, August 1993.

[30] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz. "TCP Behavior of a Busy Web Server: Analysis and Improvements," *Proceedings of the 1998 IEEE INFOCOM Conference*, San Francisco, CA, March 1998.

[31] J. Salehi, J. Kurose, D. Towsley, "Scheduling for cache affinity in parallelized communication protocols," University of Massachusetts Technical Report UM-CS-1994-075, Amherst: MA, USA, Oct. 1994.

[32] G. Gumanow, "Keeping Pace with the Rapid Adoption of Linux," Power Solutions, pp.8-12, Feb 2003. Available: http://ftp.us.dell.com/app/1q03-Int.pdf

[33] J. Hsu, “Reasoning about SMP in FreeBSD”, *Proceedings of BSDCon 2003*, San Mateo: CA USA, February 2003.

Jeffrey M. Hsu became a member the FreeBSD project in 1994 as one of its first 10 committers. He has contributed to many sections of the operating system in areas such as the networking stack, Java, and a large number of the early ports in the language category. He was offered commit bits to both the OpenBSD and DragonFlyBSD projects when they were first being formed and is active in the DragonFlyBSD project today. He was born in Taiwan and grew up in the United States. He holds a degree from U.C. Berkeley in computer science.

In the past, he has consulted for leading companies such as the Western Software Laboratory division of Digital Equipment Corporation, Cygnus, Encanto, Netscape, ClickArray, and Firetide on wireless routing. He currently consults for Wasabi Systems on improving the performance of the NetBSD networking stack and for other companies using BSD in their products. In other activities, he enjoys teaching, writing, and giving talks.