How To Schedule Unlimited Memory Pinning of Untrusted Processes
Or
Provisional Ideas About Service-Neutrality

Jochen Liedtke Volkmar Uhlig Kevin Elphinstone Trent Jaeger Yoonho Park

IBM T. J. Watson Research Center

About This Paper

You can read it as a paper that treats a concrete problem motivated in Section 1: How can we permit untrusted user processes to pin their virtual pages in memory most flexibly and as unlimited as possible? From this point of view, the paper presents a general solution that is theoretically and experimentally reasonably substantiated.

However, you can also read the paper as an approach to solve the more general problem of how an existing system can be extended by new operations while preserving the original system's QoS properties. From this point of view, the paper is highly speculative. The presented principle of service-neutral operations can successfully solve the concrete problem of dynamic pinning. However, we still have no sound evidence that it is useful for a much broader class of problems. Nevertheless, we strongly suspect it.

1 Reasoning About Pinning

Traditionally, pinning memory is a privileged capability that is not available to normal user tasks but only to the OS kernel and some device drivers. Pinning is controlled by the facts that these components are statically known, that they are trusted, and that they have mostly restrictive policies minimizing pinning.

Could applications benefit from more liberal pinning policies? There seem to be three basic classes of applications that can profit from pinning: (a) device drivers that use DMA, (b) real-time systems that rely on no-page-fault guarantees, and (c) database-like applications that perform drastically better when some application-specific pages are pinned for a shorter or longer time interval. In all cases, we see need for static and dynamic pinning.

Static pinning is used for pinning driver or real-time code and data, certain buffers, etc. Basically, pages are pinned forever or at least for a very long time. The method is necessary to guarantee the basic functionality of some components. Static pinning can be controlled by traditional quota-based methods. Static pinning is not a topic of this paper.

Dynamic pinning has two characteristic properties: pages are pinned for short periods only, and it is dynamically decided which and how many pages are pinned. We show four motivating examples:

1. For transmitting a file, the network system pins the file partially (or entirely) in memory for the transmission period. This enables zero-copy transmission, i.e. avoids copying data into network buffers, and thus increases the performance substantially.
2. For communicating data to a non-real-time component, a real-time application temporarily requests pinned memory.
3. A real-time application temporarily asks for pinned memory. Of course, the application would meet its deadlines and fulfill its minimal quality-of-service contracts without this memory. However, additional buffers can substantially improve its quality of service or reduce the overall system load, i.e. make more processor time available for non-real-time tasks.
4. Based on its application-specific knowledge, a database system decreases overall swapping by pinning dynamically some of its pages.

In these examples, we find that for dynamic pinning, applications negotiate with the system whether, *how many* and *how long* pages can be pinned for them.

1.1 Problems

Dynamic pinning has to be controlled by a policy that (a) coordinates the requests of multiple independent applications, (b) restricts the negative effects that pinning has on other (paged) applications, and (c) can be enforced against untrusted applications. In summary, dynamic pinning needs scheduling.

Assume that any user task can dynamically ask to pin some of its currently mapped pages for a certain period of time. An according scheduling algorithm in the pager then decides whether the request will be granted, denied, or granted with less pages or for a shorter period. Such a dynamic-pinning scheduler should have some properties:

1. It should work in combination with any reasonable page-replacement algorithm.
2. It should not rely on pinning-related trust in the applications. For instance, restricting pinning to device drivers is not acceptable.
3. It should be robust against denial-of-service attacks. Pinning must not offer new possibilities for monopolyzing a system.
4. It should adapt quickly to changing system load. An application should be granted a high amount of pinned pages if enough unused memory is available. Accordingly, less pinning should be granted if memory is getting short.
5. The basic scheduler should enable to add specific policies that, e.g., handle applications that are prioritized or weighted differently.

Due to (3), pinning must always be bound by a time limit. Furthermore, because of (2), we cannot rely on applications unpinning their pages on time. Therefore, unpinning must be enforced if necessary.

(3) and (4) together exclude such simple solutions as to give each application a fixed quota of n/m pinned pages in a context of n physical pages and m applications.

1.2 Related Work and Systems

Some systems permit in-kernel modules to pin pages dynamically, e.g. NT's VIA [6] and FreeBSD [5]. Safety and scheduling of pinning are not addressed. The compensated pinning methodology of this paper permits untrusted processes to pin pages and schedules competing pinning requests of concurrent applications. The latter feature would be helpful even in purely trusted pinning systems. Posix 1003.1b [2] specifies an *mlock* operation that lets the superuser pin pages up to a predefined system limit. Normal users cannot pin memory.

Solaris complements superuser pinning with a quota system. In theory, this could be extended to normal user processes. However, quotas are very static and have to be allocated very carefully, typically by a human system administrator. In contrast to compensated pinning, they do not permit overbooking and make fast adaption to the current system behavior impossible.

1.3 Time-Constrained Pinning

As a first approach, we introduce a *time-constrained* pin operation *SimplePin (RequestedPages)* $\rightarrow$ $(\tau,$ *PinnedPages)*. By this operation, any application can request that some or all of its currently mapped virtual pages are pinned in physical memory. The system returns a time τ for which it guarantees pinning and the set of pages that could be pinned successfully. (Unmapped pages are not pinned.)

SimplePin could, for example, be used in a user-level protocol-stack implementation to avoid copying data into send buffers. The protocol stack requests pinning for the according user data. Depending on the returned time constraint τ, it lets the network driver (also user-level, of course) work on such an amount of data that it certainly finishes within time τ.

If the driver permits running operation (including DMA) to be aborted and ϵ is the maximum latency for such an abortion, we can use a more sophisticated *AdaptivePin* algorithm:

```
AdaptivePin (pageset, $\hat{\tau}$) :
    $T$ := now + $\hat{\tau}$ ;
    **do**
        SimplePin (pageset) $\rightarrow \tau$ ;
        **if** $\tau < \epsilon$ **then** return failure **fi** ;
        work for ($\min(\tau - \epsilon,\ T$−now))
    **until** now $\geq T$ **od** .
```

By repeated *SimplePin* operations, it tries to pin the (mapped) pages over a user-specified period $\hat{\tau}$. If the system no longer guarantees sufficiently large τ's, the algorithm terminates with a failure. In this case, however, that pages are still pinned for a time ϵ such that the protocol stack can securely stop transmission (including DMA operations).

After τ expires, the pages are no longer pinned. If the application ignores τ, something will not function properly. Is this a new safety/security leak? We do not think so. If, for instance, the protocol stack or network driver continuous working and runs DMA although τ expired, it can obviously transmit the wrong data and penetrate privacy. However, someone who is able to access physical memory can do such things even without pinning. So no additional trust is needed.

The really hard problem remains: How can the system guarantee sufficiently large τ's and simultaneously guarantee that the entire system still works nicely?

2 About Service-Neutrality

Untrusted pinning can easily make a system's behavior unacceptable. Not only explicit denial-of-service attacks are crucial in this context. Without proper scheduling, even normal and correct applications could easily degrade the system's performance unacceptably or even let the system starve. Besides pinning, other operations, in particular resource-reserving ones might have similar problems. Therefore, in this Section, we discuss the according scheduling problems.

The most general question is: If we make a new operation X available for untrusted applications, how does this change the system's QoS properties. Unfortunately, this question seems to be too hard to answer, particularly because we see no reasonable and general possibility to decide which is the "better" one of two system behaviors. Therefore, we limit ourselves to a slightly less general problem:

How can we extend a given system by a new operation X *such that the original system's QoS-properties are always preserved?*

2.1 Basic Terms

A *system* consists of a scheduler and a set of applications $A_0, A_1, A_2 \ldots$ that are executed concurrently. While the system executes, the scheduler produces a *schedule*, i.e. it maps each point in time per application to the set of *current resources* that the application can use at this time.

The *service* that an application A_i gets is the projection of the entire schedule on to A_i, i.e. mapping time to the sequence of resources for A_i.

Quality of service can then be described as a function that maps service(s) of one or more applications into a multidimensional space of qualities, typically R^n. Examples of QoS measures are number of mapped pages per time, cpu time per real time, maximum latency, etc. While preserving only specific QoS properties can be compliant with changed services, preserving *all* QoS properties requires to preserve the services.

2.2 Strict Service-Neutrality

Assume that we want to extend a given system by offering a new operation X. Usually, some system services have to be modified for this purpose, e.g. the page-replacement policy to enable pinning. Since all system policies can be regarded as a part of the scheduler, we basically talk about how to extend the original scheduler for X.

We are obviously only interested in scheduler extensions that, loosely speaking, preserve as far as possible the properties of the original scheduler. Such *service-neutral* scheduler extensions should be characterized by two properties:

- X-free applications are served under the control of the extended scheduler in the same way as under the original scheduler's control.
- When some applications use X-operations, all concurrently active X-free applications should experience a service that they also could have experienced in the original system. (I.e., the same service could have happened even without the presence of X-operations.)

We now define more formally the term *strict service-neutrality*: Assume that a system runs the application A concurrently with a set of competing applications C_i. A invokes the new operation X with parameters x_j. (We make no assumptions about the use of X in the other applications.)

- Operations $X(x_j)$ are *strictly service-neutral* iff there exists an X-free application A' such that for every possible set of competing applications $\{C_i\}$, the service for every C_i remains the same if we replace A by A'.
- A scheduler extension is *strictly service-neutral* iff it rejects any application that invokes a not strictly service-neutral $X(x)$.

Obviously, strictly service-neutral scheduler extensions preserve all QoS properties of the original system. In particular such extended schedulers are compliant to the original scheduler, i.e., they behave exactly as under control of the original scheduler.[1]

Note that we do not require a strictly service-neutral scheduler extension to accept *all* service-neutral invocations $X(x)$. For practical relevance, it nevertheless should accept "enough" invocations. Constructing such a scheduler extension depends heavily on the addressed operation X. In Section 3, we describe a solution for pinning.

2.3 Probabilistic Service-Neutrality

Strict service-neutrality is a very restrictive requirement. We suspect that it is too restrictive to be useful on its own. It might turn out that strict service-neutrality is applicable only to few and uninteresting operations X.

In similar situations, probabilistic methods have been very useful. For example, complexity theory uses probabilistic complexity; probabilistic properties are helpful in load distribution [1] and for probabilistic broadcasts [3], stochastic capacities [4] describe cache properties independent of specific benchmarks.

Accordingly, we define *probabilistic service-neutrality* as a weaker but more powerful concept:

- Operations $X(x_j)$ are *service-$\mathcal{P}$-neutral* iff there exists an X-free application A' such that for a randomly picked set of competing applications $\{C_i\}$, the service

[1] *Proof sketch:* Since the scheduler extension accepts any X-free application, any such application is accepted iff the original scheduler would accept it. Furthermore, it is served exactly as if it would run under control of the original scheduler. (Take the empty application for A and A'. Then arbitrary X-free C_i are serviced in the same way as under the original scheduler.)

for the C_i remains the same with probability p (usually chosen close to 1) if we replace A by A'.

- A scheduler extension is *service-$\mathcal{P}$-neutral* iff it rejects any application that invokes a not service-$\mathcal{P}$-neutral $X(x)$.

(In the following, we use the term *service-neutral* to denote strict and/or probabilistic service-neutrality.)

If the scheduler accepts only service-$\mathcal{P}$-neutral $X(x)$, the QoS properties of the system are preserved with probability p. This is acceptable for timesharing and soft-real-time QoS properties but in general not for servicing hard-real-time applications. However, sometimes even hard-real-time applications can coexist with p-neutrality. Compensated pinning in Section 3 gives a good example. Although pinning is only p-neutral in general, it does not influence applications that avoid paged memory. So pinning is strictly neutral for hard-real-time applications.

2.4 Application-Locality

From the definition of service-neutrality, we can conclude that any reasonable service-neutral scheduling extension will decide whether it will accept an application A independently from the other applications C_i. In other words, a scheduler extension will work on the application in question without looking at the other competing applications.[2] This *application locality* has two remarkable consequences:

- Since the additional scheduling mechanism does not need system-global knowledge, we can expect that total scheduling complexity does not increase significantly. In particular, it should only increase when operations X are invoked.
- When trying to construct a service-neutral scheduler extension, we should focus on application-local algorithms. Section 3 illustrates how we can use this knowledge as a guideline for construction.

3 Compensated Pinning

Now we switch back from the abstract operation X to the original *pin* operation. From the prior analysis, we can draw some conclusions, step by step:

1. We have obviously to modify the original page-replacement algorithm such that whenever it picks a page frame holding a pinned page the replacement is redirected to another page frame. (We call such redirections *compensated replacements* or *compensations*.)
2. So the basic problem is: how can we select the compensations service-neutrally, i.e. without affecting the system's paging behavior?
3. Due to the application-locality of service-neutral policy modifications, compensations must always redirect replacement to (mapped) pages *of the same application* that pinned the page that originally should be replaced.
4. Therefore, an application has to have more pages mapped than it requests to pin. The addititional pages are required for compensation.
5. After no more pages are left for compensation, pinning would be no longer service-neutral. The new neutrality-ensuring scheduling mechanism has therefore to determine τ such that the application will either never run out of compensation pages in the following time interval τ (strictly service-neutral) or that this will happen with a probability of at most $1 - p$ (service-$\mathcal{P}$-neutral).
6. A strictly service-neutral strategy is totally infeasible. We have to focus on probabilistic neutrality.
7. Due to the application-locality, τ must for any given p be derivable from the application's current set of pinned and compensation pages.

The conclusions clearly lead us to the idea of *compensated pinning*: Assume that an application wants to pin k of its pages that are currently mapped. To get those pages pinned for a certain time, the application has to offer x other pages for compensation. These *compensation pages* must be pages owned by the same application and they must be currently mapped to physical memory. For page replacement, the pager can then use the compensation pages instead of the pinned pages. The following paragraphs show in detail how we can compensate the pinning costs to nearly 100% for any reasonable page-replacement algorithm.

3.1 The Compensation Algorithm

Assume an application A wants to pin k pages and offers x compensation pages. (All $k + x$ pages are owned by A and are currently mapped.) A touches all $k + x$ pages to classify them as recently used for the page-replacement algorithm. Then it asks for pinning while offering the compensation pages.

We modifiy the original page-replacement algorithm by adding the following compensation algorithm. Initially, all $k+x$ pages are marked as *unlinked*. The x pages are put into the set X_A of A's unused compensation pages. (Note that

[2] *Proof sketch:* If the acceptance decision would depend on the current $\{C_i\}$, then there would exist at least one other $\{C_i'\}$ that would let A to be rejected. However, by definition neutrality holds for all $\{C_i'\}$. Therefore, the C_i-dependent algorithm would be either wrong or could be improved by simply ignoring the current $\{C_i'\}$.

this does *not* change the replacement status of the x compensation pages from the original replacement algorithm's point of view.)

Whenever the original page replacement picks one of the $k+x$ pages, i.e., a pinned page or a compensation page, the compensation algorithm is applied. Assume that page frame f is picked originally. If f is not yet linked to a compensation page, an arbitrary page f' is taken out of X_A, removed from this set, f is linked to f', and f' is used instead of f for replacement. Otherwise, the page f' that was already linked to f is used instead of f.

```
compensate ($f$) :
    **if** $f_{\text{linked}}$ = nil
        **then** $f'$ := one of $X_A$ ;
              $X_A$ := $X_A \setminus \{f'\}$ ;
              $f_{\text{linked}}$ := $f'$
    **endif** ;
    use $f_{\text{linked}}$ instead of $f$ for replacement .
```

Once X_A has fallen empty, replacing not-yet-replaced pages can no longer be compensated. The application A is called *compensated* until this happens and *uncompensated* thereafter. The number of replacements that leave A compensated is called its *compensation length* C.

The concrete compensation length can be as low as x. However, this event has a very low probability. Therefore, instead of this absolute lower bound, we use a probabilistic lower bound $C_{\mathcal{P}}$ for the compensation length. It means that with probability $\mathcal{P}$, the concrete compensation length is at least $C_{\mathcal{P}}$. For instance, we can expect that in 99% of all pinning operations, we do not run out of compensation pages for up to $C_{99\%}$ replacements. In Section 3.2, we derive the probabilistic lower bound $C_{\mathcal{P}}$ from given k and x. Accordingly, the number x of compensation pages that are necessary to pin k pages for $C_{\mathcal{P}}$ replacements can be calculated.

With a lower bound $T_{\min}$ for the minimum time required for one replacement operation, the probabilistic pinning time is at least $C_{\mathcal{P}}T_{\min}$. For k pages to be pinned and x offered compensation pages, the pager could guarantee a pinning time of $\tau = C_{\mathcal{P}}T_{\min}$. We know some better probabilistic bounds for τ than the simple $C_{\mathcal{P}}T_{\min}$, but cannot discuss them here due to space restrictions.

As long as compensation works for a period of τ, the paging behavior is unaffected by pinning the k pages. Pinning does not shrink the memory that is effectively available for swapping. However, once we experience an underflow of the compensation set, the pager must replace an unpinned page *without* compensation. As a result, the available swapping memory effectively shrinks. Fortunately, such situations occur infrequently. On average, their probability is even lower than $1-\mathcal{P}$ because in most cases the effective time for r replacements is much higher than $T_{\min}r$.

3.2 Probabilistic Compensation Length

An application pins k mapped pages and offers x also mapped pages for compensation. Let p be an upper bound for the probability that the original replacement algorithm picks a given page of the $k+x$ pages per replacement event. Typically, $p = 1/n$ is a good choice for n total page frames.[3]

We are interested in the probability $\mathcal{P}(r)$ that r replacements can be compensated. At first, we calculate $\mathcal{P}(r)$ under the assumption that the probability to be picked in one replacement for any of the $k+x$ pages is constant and equals p. Then, we show that $\mathcal{P}(r)$ is a lower bound for the case that p is an upper bound for these probabilities.

From the compensation algorithm, we can easily conclude that $\mathcal{P}(r)$ is the probability that the original replacement algorithm for r replacements picks at most x different pages from the set of the $k+x$ pages.

Assume that we select i pages from this set. Then the probability that in r replacements each of the given i pages is picked at least once and all other of the $k+x$ pages are never picked is

$$P_i(r) \;=\; \begin{cases} \big(1-(k+x)\,p\big)^r & \text{if } i=0 \\ \displaystyle\sum_{j=1}^{r-i+1} \binom{r}{j} p^j\, P_{i-1}(r-j) & \text{if } i>0 \end{cases}$$

Multiplying $P_i(r)$ by the number of combinations how i pages can be selected from $k+x$ pages and summing over all possible values of i results in

$$\mathcal{P}(r) \;=\; \sum_{i=0}^{x} \binom{k+x}{i} P_i(r)$$

Only $P_0(r)$ depends directly on p. Smaller values for p result in larger values for $P_0(r)$ and thus also for $P_i(r)$ and $\mathcal{P}(r)$. Therefore, $\mathcal{P}(r)$ is a lower bound for the probability that r replacements can be compensated if the pick probability for each of the $k+x$ pages for r replacements remains less than or equal to p.

Table 1 shows probabilistic compensation lengths for $p = 1/n$ and various memory sizes n. The margin probability is always 99%, i.e. we calculate $C_{99\%}$. If, one a 64 M-machine, e.g., 256 pages (1 Megabyte) are pinned with 16 compensation pages, in 99% at least 555 replacements can be compensated.

[3] $p = 1/n$ assumes only that the replacement probability for a currently used page is at least not higher than for an average page. (The $k+x$ pages are currently in heavy use!)

$k + x$	**16 M** n=4096	**32 M** n=8192	**64 M** n=16384	**256 M** n=65536
+ 1	37	74	128	590
+ 2	105	211	421	1683
16 + 4	293	585	1170	$>$2000
+ 8	728	1455	$>$2000	$>$2000
+ 16	1571	$>$2000	$>$2000	$>$2000
+ 2	7	14	28	111
+ 4	21	42	82	326
+ 8	57	112	223	886
256 + 16	141	279	555	$>$2000
+ 32	323	642	1278	$>$2000
+ 64	689	1370	$>$2000	$>$2000
+ 128	1360	$>$2000	$>$2000	$>$2000
+ 2	2	4	7	28
+ 4	6	11	21	83
+ 8	16	30	58	225
1024 + 16	39	74	144	568
+ 32	89	172	338	1336
+ 64	194	378	748	$>$2000
+ 128	405	796	1580	$>$2000
+ 2	n/a	2	2	7
+ 4	n/a	4	6	21
4096 + 8	n/a	9	16	58
+ 16	n/a	22	39	145
+ 32	n/a	48	90	342
+ 64	n/a	104	198	765
+ 128	n/a	219	422	1651

Table 1: $C_{99\%}$

4 Compensated Pinning in Linux

We implemented compensated pinning in Linux. Each application (i.e., a Linux process in our implementation) has a dedicated compensation set. It is empty upon the process' creation. The process can then add compensation pages by pin requests. The API offers 6 elements:

int CompensatedTau (int k, x) is a function that — based on the calculated probabilistic compensation length, the estimated swapping costs, etc. — delivers what τ could be expected when pinning k pages while x compensation pages are available. The according values are guaranteed not to change within a system run and to be the same for each application. So applications can use this function/table to decide which compensation-page resource they need.

type pageset is used to specify sets of pages.

*int Pin (pageset PinReqs, CompPages, pageset *NotPinned)* requests to pin pages and simultaneously offers compensation pages. (These are added to the process' compensation set.) The function returns the time τ in milliseconds as a result and writes all pages that could not be pinned (because they are currently not mapped) into the pageset NotPinned.

UnPin (pageset Pages) unpins all of the specified pages that are currently pinned.

int FreeCompensationPages () delivers how many free compensation pages are current available for this application, i.e. have not yet been used for compensation.

*ReclaimAllFreeCompensationPages (pageset *Pages)* reclaims all compensation pages of the current application that have not been used so far. Afterwards, the application's compensation set is empty.

In most systems, e.g. in FreeBSD, the swap daemon picks page frames directly. The system has some inverted mapping table that permits efficient determination which virtual page(s) in which address space(s) is/are currently mapped to a physical page frame. Original Linux instead, has no such inverted mapping table. Thus the swap daemon walks through the virtual address spaces, selects a virtual page, and replaces the corresponding physical page frame if certain criteria are fulfilled.[4] To compensate replacement attempts on pinned pages, we need sometimes to redirect the Linux swapper to a different physical page frame. To replace it, we had to invalidate all mappings to this page frame and therefore added an inverted mapping table to Linux.

In a simple experiment, we pinned pagessets of 64 KB, 1 MB, and 4 MB on a 32 MB Linux system and measured the achievable compensation lengths. Simultaneously, 10 applications wrote randomly into 4 MB arrays such that the system was always swapping heavily. (Our testbed was a Pentium Pro 200 with 32 MB RAM running L^{4}LinuxVersion 2.0.21 and a 540 MB harddisk (IBM DALA-3540, 12ms, 96 KB Cache, Transfer: 40.5MB/s, 4500 rpm).

Table 2 show results of 5 experiments per $k + x$ variant. The results are sorted per column such that the first row shows always the minimal achieved number of replacements that could successfully be compensated. As expected, the number are much larger than the probabilistic length. Due to the small number of experiments, the effect of increasing compensation pages *on the minimum* is also not very strong.

16			**256**			**1024**		
+ 1	+ 4	+ 16	+ 4	+ 16	+ 64	+ 4	+ 16	+ 64
1974	1909	2520	1156	596	1455	751	704	738
2019	2148	2575	1277	1343	1840	775	736	778
2413	2157	2910	1550	1437	1942	796	836	809
2484	2309	2985	2024	1444	2278	841	923	834
3367	2648	3707	1550	1445	3432	845	1559	908

Table 2: Achieved Compensation Lengths.

5 Summary and Outlook

The current research laid the foundation for an untrusted dynamic memory pinning that is safe, secure, adaptive and

[4]In Linux 2.0.21, the swapper did never swap out shared pages, probably because of the lack of an inverted mapping table. So one could easily stop the system forever by blocking all physical memory indefinitely: simply create a large array, touch every page, then fork and wait forever.

preserves a system's QoS properties. The general principle of service-neutrality might be applicable for further problems.

Besides maturing the Linux implementation, we are currently working on deriving better probabilistic time bounds (τ) from compensation lengths by theoretic and heuristic methods. In a next step, the use of dynamic pinning in applications will be exploited.

References

[1] A. Barak, S. Guday, and R. Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*. Lecture Notes in Computer Science, Vol. 672. Springer–Verlag, 1993.

[2] B. O. Gallmeister. *Posix.4*. O'Reilly & Associates, 1995.

[3] M. Hayden and K. P. Birman. Probabilistic broadcast. Technical Report TR96–1606, Cornell University, Ithaca, NY, September 1996.

[4] J. Liedtke. *On the Realization Of Huge Sparsely-Occupied and Fine-Grained Address Spaces*. Oldenbourg, 1996.

[5] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD OS*. Addison Wesley, 1996.

[6] D. A. Solomon. *Inside Windows NT*. MS Press, second edition, 1998.