INF-2201 P5 Report: Virtual Memory

Kristina Kufaas
kku020@uit.no
kkufaas@github

Eindride Kjersheim
ekj012@uit.no
eika123@github

***Abstract*—In this project we study virtual memory mechanism in form of design and implementation of mapping between physical and virtual addresses, dynamic swapping to/from disk, page fault handling and page allocation, using a FIFO eviction strategy.**

I. Introduction

This report describes an implementation of virtual memory on an Intel i386 processor with a simple mechanism for providing separation between processes and swapping out fixed size chunks of memory called pages that don't fit in the installed physical memory.

II. Theoretical Background

The main aims of virtual memory is to make sure that the each process has its own isolated environment with a uniform memory address space across processes, and that the total memory space used by programs running on the computer may exceed the installed physical memory. In this way, the programmer doesn't need to think about other programs running on the computer, or be concerned with hard limits on memory. It also solves problems related to external fragmentation (discussed in lectures, see also Tanenbaum and Bos [1, pp. 243–245]) that occur when processes finish running, making a hole in the memory region the finished process occupied. If processes must fit contiguously in physical memory, this becomes a problem. Virtual memory and paging avoids this problem altogether.

A. Memory translation

According to Tanenbaum and Bos [2], there is a translation layer between the CPU and main memory called the Memory Management Unit (MMU), that translates the memory addresses used by the processor to addresses in physical memory. An address used by the processor is called a virtual address, and the translation to physical addresses is done by lookup in page tables, which may be organized in hierarchies. For a 32-bit system a two-level hierarchy like in figure 1 a) is typically used, where the virtual address is broken up into three parts like in figure 1 b). For instance on our chosen x86-32 architecture, the ten most significant bits determine the index into the top-level page directory, which provide a pointer to the second level page tables. The following ten bits are used to index the page table, while the offset is used to determine which byte in the page one is to read. The page tables are stored in memory, and translation is handled by the MMU, which also contains a cache called the *translation lookaside buffer* (TLB) for speeding up the translation process.

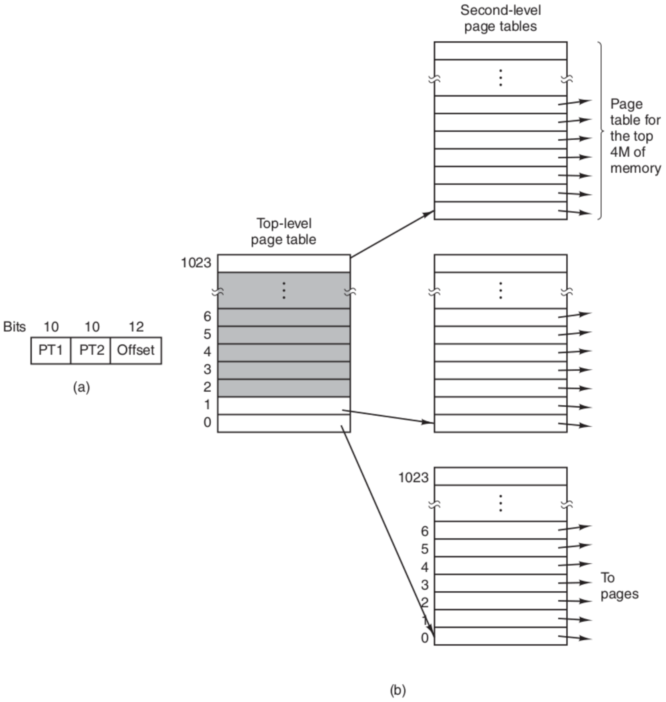


Fig. 1: *Page table lookup, Tanenbaum and Bos [1, p. 206]*

Note that as the memory is divided into fixed size chunks, the page tables only need to address the base of each page, while the offset of the virtual address provides the entry into the individual page. Therefore, not all 32 bits in the page tables are needed for addressing. Instead, the ”superfluous” bits are used for keeping track of the state of the pages they reference. The most important bits in this paper are the bits for marking a page as present, written to (dirty) and the privilege level required to access the page. If a page is not present in the page table being referenced, an interrupt called a *page fault* occurs.

B. Handling page faults (swapping)

Page faults can occur when a program tries to access a physical memory that is not mapped in its page tables, or on access violations due to either privilege or the type of access like read, write or execute. A page fault handler must receive some information about the cause of the page fault from the machine.

When a page fault occurs because of a nonpresent page, the operating system fetches the page from disk into memory and updates the page table. In this process it is also possible that another page must be *evicted* from memory. The dirty bit indicates whether this page was modified since it was loaded into memory. If it was, it should be written back to disk to keep the state of its process intact. A more detailed overview of page fault handling is found in Tanenbaum and Bos [2, pp. 233–234].

Even though a process must be in main memory to be executed, it can be placed temporarily to a disk and then brought back into memory for continued execution. This is known as swapping, like in figure 2, except that instead of swapping out an entire process as was done in some early systems, individual pages may be swapped out. As Silberchatz points out, swapping increases the possibilities of multiprogramming on a system, as processes don't need to fit entirely in memory, and the sum of required memory can be larger than the physical memory.

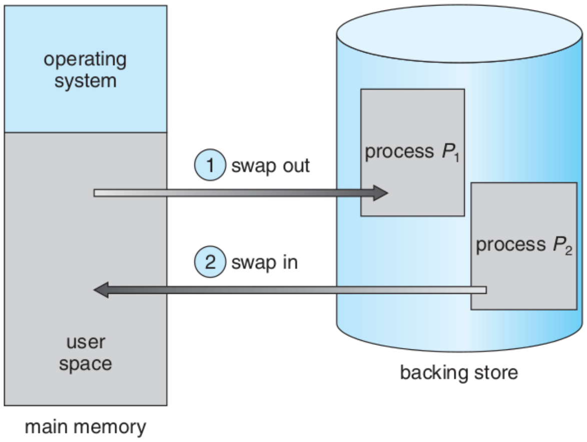


Fig. 2: *Swapping of two processes using a disk as a backing store, Silbershatz, Galvin and Gagne [3, p. 358]*

C. Physical page frame allocation

Accessing data from disk in standard swapping is much slower than from RAM, leading to increased system latency. However, modern operating systems use advanced algorithms to predict and optimize which data to swap in and out, improving performance over standard swapping.

When there are no free frames available, the operating system must decide which page to evict from physical memory to make room for a new page. Common eviction algorithms include random, that simply selects a page at random from the set of all pages currently in memory, LRU, which evicts the page that has not been accessed for the longest time, assuming that pages used recently will likely be used again soon; and FIFO that evicts the oldest page in memory, based on the time it was loaded, regardless of its use since loading. See e.g Tanenbaum and Bos [2, pp. 207–221] for details.

When choosing a candidate for eviction one must typically need a way to keep track of what process is using a physical page frame and what virtual address is related to that page frame, so that the page tables of that process can be updated to indicate that the frame is no longer present. This requires a suitable data structure.

D. Setting up new process's address space

When a new process is created, the OS allocates pages for a page directory, required page tables and a suitable number of pages for the new process to keep track of the mapping between the virtual and physical addresses used by the process and the physical memory addresses where the actual data resides.

Not all pages of the address space need to be loaded into physical memory at once. They may be loaded on demand—i.e., only when they are accessed.

1) Page access and interrupt handling: In order to handle interrupts efficiently without juggling page directories, the kernel is usually mapped in each processes page directory. Protection flags dictate what kind of operations (read, write, execute) are permissible on each page, and a privilege bit indicates what privilege level is required [4]. In this way a process cannot access memory in ways it should not even if it is mapped in its page table structure, and different processes are isolated from each other by switching out the mapping on context switches and making sure the memory accessible to the processes in user mode maps to different regions of physical memory. The mapping to the kernel must be uniform across processes, since on an interrupt most CPUs will jump a predefined memory location where the interrupt table resides, as described by Intel's i386 manual [5, p. 209]. This jump is done without knowledge of address translation, so the easiest way is to use the identity map for the kernel, which is fancy language for setting virtual and physical addresses equal for the kernel.

Note also that on context switches the TLB cache must be invalidated to keep process separation intact as described by Tanenbaum and Bos [2, p. 233].

III. System description

A. Design and Implementation

1) Setting up new process's address space: The kernel is identity mapped with suitable privilege bits, except that the video memory area is mapped with user access instead of requiring system calls. A page is allocated for the page directory, page tables for the kernel space and user space, and a page for the user stack. All these pages are marked as pinned upon allocation such that they cannot be swapped out.

To keep track of what page frames belong to what processes in user space and the virtual addresses related to said page frame, a hash-map-like array of structs containing this information is provided, where the physical base address of the frame is mapped into an index in the array of structs. Knowing

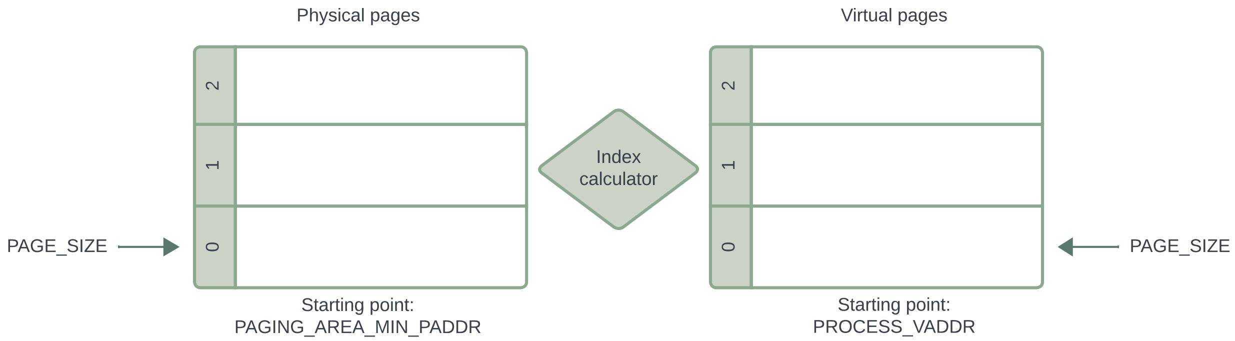


Fig. 3: *Memory mapping*

the bottom address for each physical page frame location, we could connect virtual and physical addresses using the index of each page frame info structure. See figure 3. The structure is somewhat similar to an inverted page table as described by Tanenbaum and Bos [2, pp. 206–207], but the map is from physical addresses to their process and virtual address, not the opposite way around. The overhead of keeping such a record is small. Our struct contains 6 32-bit integer fields as references other structs, the physical and virtual address, and a bitfield containing information about the struct such as wether it belogns to a user process or the kernel, wether it is pinned or not etc. The total is therefore $6 \cdot 4 = 24$bytes/struct disregarding alignment. Suppose that the entire address space is pageable on a 32-bit system. Then one needs $4\text{GiB}/4\text{KiB} = 1024^2$ pages. The size needed by these structs on a 4GiB system is then approximately $24 \cdot 1024^2$bytes $= 24$ MiB, using only 0.6% of the total memory. This shows that this structure scales well.

2) Handle page faults (swapping): Our system provides a simple page fault handler. Unless the page fault is due to a page not being present, the page fault handler outputs a suitable error message and quits the OS.

Upon a page not being present, we conjecture that it must be loaded from disk as we do not support shared pages between processes. Therefore a simple handler is provided that tries to allocate pages, load the requested page from disk and update the page tables. This is also where most of the eviction logic is handled.

The initial setup retrieves the faulting address and the current page directory.

The faulting address on the x86-32 architecture is provided in the CR-2 register which we read into a variable on the handlers call stack, and the reference to the faulting process control block is read into another local reference on the call stack so that the computer may continue to handle other interrupts during the handling of the page fault. A lock is used to ensure that only one page fault is being handled at anyone time. Other processes that page fault while another one is being handled will block on this locks queue.

When the memory is full, pages need to be evicted from memory. We opted for the FIFO algorithm for eviction. Although this required the addition of extra data structures and functions, the behaviour of a FIFO queue proved to be more predictable compared to the random algorithm. Furthermore, we developed a specialized debugging function that displays the current state of the FIFO queue after each iteration of the page fault handler. This allows us to clearly understand which process will be evicted next.

To keep track of all pageable pages we use global data structures, such as linked list of free pages in form of page frame info blocks, all page frame info blocks consistent to number of pageable pages and fifo-queue to determine the next page suitable for eviction.

3) Physical page allocation: Physical pages for page tables are allocated dynamically: memory for user processes (code and data segments) is set up to be loaded on demand. Page table pages, as well as kernel page, process directory and user stack pages are ”pinned” upon initialization, meaning they are marked to avoid being swapped out. The setup includes mapping of the process’s specific memory regions into its page table.

IV. Correctness checks and evaluation

A. Correctness checks

As in previous projects, we used Bochs’ output file via the com port to keep track of message passing during debugging. In addition, we wrote several debugging functions that provided a clear representation of the status of all pages in a human-friendly table form.

The content of the serial output log was piped into `grep` with suitable regular expressions to visually inspect that the page allocation and the FIFO queue made sense.

B. Correctness results

We find no faults in the sequence pages are evicted and brought in from disk. However, there seems to be an elusive bug in our paging that happens when a lot of swapping occurs. The two mailbox test processes are sensitive to this and will get bad data and exit. However, we have a stable solution that balances this.

First, limit the number of simultaneosly running processes. The kernel loads the process only if it estimates that there is sufficiently small competition for pages. This solution avoids thrashing, but tuned well allows some paging and swapping to occur.

Second, use an optimization where user processes share the same kernel level page table, because this page table is identical across processes. We set a dummy kernel pcb as the owner such that this table is never evicted or cleaned up by some cleaner thread.

To observe paging and swapping, spam 'fire' in the shell, or pin the shell to restrict the number of page frames by a suitable amount.

A slightly less stable solution is to unpin the shell and run 4 processes simultaneously, but using the shell while all processes run makes this very unstable, especially when we tried to load all 4 processes fast almost at the same time. The sequence of loading also may have effect, sometimes loading 1, 3, 4, 2 or 1, 4, 3, 3 causes a 'bad data' error due to incorrect smaller message size then mailbox in process 4 expects. See figure 6 for a running example.

See the `'config.h'` file to set these parameters. Finally, see the appendix with images at the end of this report.

Another possible bug is that if we use the standard 640 KiB kernel size from P4, we cannot load in more than two processes properly.

V. Discussion

Initially, the assignment description did not clearly specify the concurrent running of all four processes; we assumed they would operate simultaneously. Later, we were informed by TAs that some processes might need to be completed before proceeding to next process and therefore we decided to use the completion of a process as a trigger to clear all its pages, thereby freeing up space. We implemented this by integrating a cleanup function into the scheduler, which ensured smooth and predictable operation without halting. Essentially, the cleanup function removed the pinned flags, allowing the page table and page directory of a completed process to be evicted.
However, with this solution we couldn't run all 4 processes simultaneously as we desired, because we forgot to bit-mask the page frame reference when writing a page to disk, so it led to overlay of two neighboring page frames being written to disk, corrupting the image. The cleanup function was discarded because cleaning up a processes memory requires a cleaning daemon to avoid having a process remove e.g its own page table, as suggested by Tanenbaum and Bos [2]. However, we think this function could be reused for such a daemon with minor revisions. Also, a point of the exercise was to explore swapping and assess its correctness, which is harder to observe in the current state of our system when everything gets cleaned up after use.

The current bug, where all 4 processes run smooth only if we launch them not too fast one by one, could happen due to a number of reasons - for instance a race condition; although we think our page fault handling is decently protected. Other reasons could be failing to invalidate a page at some point. Also, our code is unnecessarily complicated that reduces readability, maintainability and debugging. Adding some page invalidation instructions after some page mappings reduced the bad data issue significantly, but did not remove it compeletely.

We also identified several opportunities for optimizations. For instance, maintaining a list of free pages becomes redundant when using a FIFO queue. However, the system does suffer from a lack of flexibility, which complicates tasks such as changing eviction strategies with ease. Also, a future cleaner daemon could put pages of exited programs back into such a free-list, in effect putting them in front of the queue to avoid eviction when non-used pages become available. To scale such a future cleaner daemon to larger memory capacities, the process control blocks should maintain a linked list to it's used page frames. This would allow for a straightforward process during exits, where one could traverse the linked list to unpin all pages. Additionally, it's important to clear the dirty bit of page frames used by a finished process, as the data within those frames is no longer relevant.

An alternative approach could involve using the bit field in the page frame info struct to mark the page frame as currently inactive. This would allow the eviction algorithm to bypass checking the dirty bit in these instances. Currently, there is a bug in our freeing mechanism; however, we have not yet observed the potential consequences of inadvertently writing data from one completed process into the swap area of another process.

The linked list should encompass all page frames allocated to the process, suggesting the use of a doubly linked list. This configuration allows for pages that need to be evicted during the process's execution to be removed from the list seamlessly. The info structs work similar to a hashmap, which eliminates the need to traverse the entire list during eviction. This arrangement presents a compromise between the size of the info structs and the time required to clear the used memory of a finished process. However, as noted in the system description section III-A1, this setup does not significantly increase the space used.

A. Future work

Other future iterations of this project could benefit from implementing a metrics collection framework for measuring page fault rates, swap operations, and other relevant performance indicators, as well as testing of different eviction algorithms. With such data, we could better understand the impact of different page replacement strategies and make more informed decisions to optimize system performance.

VI. Conclusion

Our project has demonstrated the fundamental capabilities of a basic operating system by implementing such components as process address space setup, page fault handling, and an eviction mechanism for memory management. Throughout the project, we faced the challenge of managing limited physical memory resources efficiently and keeping system stability through swap management strategies.

Although we were unable to make the paging and swapping mechanism flawless or empirically measure the performance impact of our paging system due to time constraints, the project laid a groundwork for further exploration.

This improved our understanding of low-level system operations and sharpened our problem-solving and software development skills in a complex setting. The insights gained and the infrastructure established provide a foundation for future educational and development efforts in systems programming and OS design, and provides a good understanding of a common memory system in general.

VII. Appendix

We got a feedback from TA that they couldn't run our code because of error message ”p5-inf-2201-ekj012-kku020/target/../src/kernel/memory.c:1292:undefined reference to `log_interrupt_frame'. We provide screenshots in figures 5 and 7, showing that our implementation is able to run stable with 4 processes, however a little bit slowly due to paging.

References

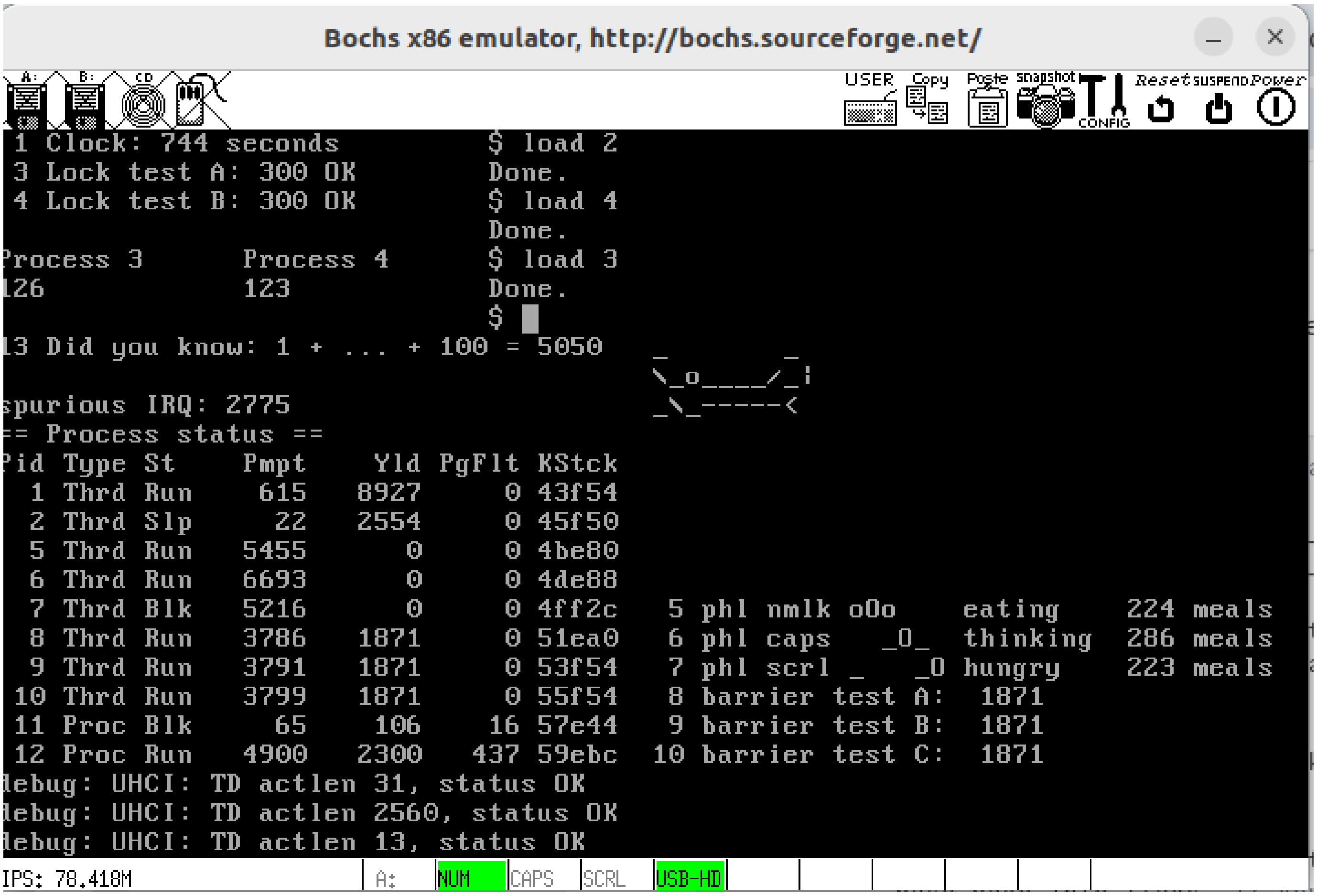
[1] A. S. Tanenbaum and H. Bos, *Modern operating systems*, eng, Fourth edition. Pearson, 2015.

[2] A. S. Tanenbaum and H. Bos, *Modern operating systems*, eng, Fifth edition. Hoboken: Pearson, 2024, ISBN: 9781292459660.

[3] A. Silbershatz, P. B. Galvin and G. Gagne, *Operating System Concepts*, Ninth edition. Wiley, 2013, ISBN: 978-1-118-06333-0.

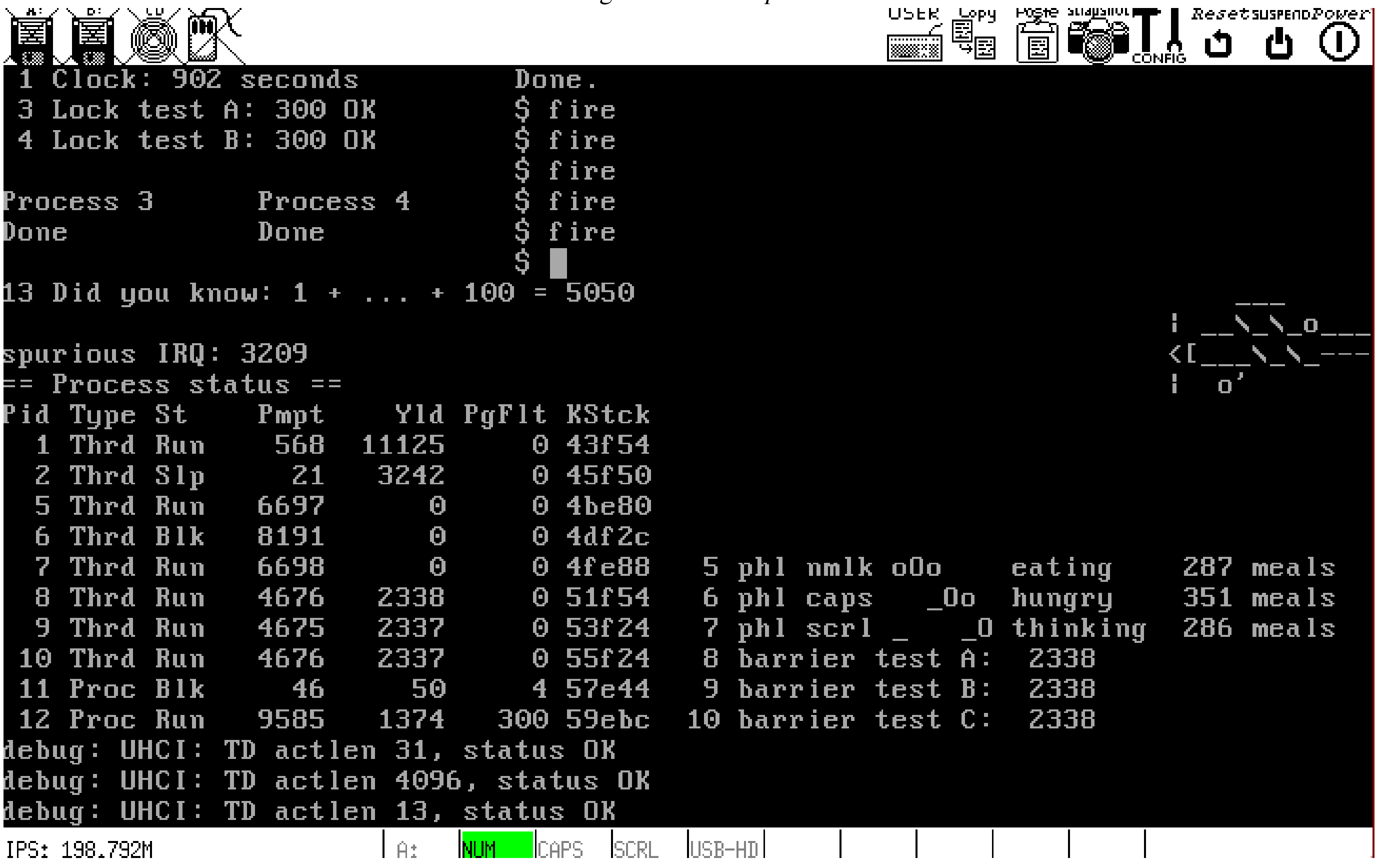
[4] StackOverflow. "Why is kernel mapped to the same address space as processes?" (2024), [Online]. Available: https://stackoverflow.com/questions/13013491/why-is-kernel-mapped-to-the-same-address-space-as-processes (visited on 17/04/2024).

[5] Intel, *Intel 80386 programmer's reference manual*, 3065 Bowers Avenue, Santa Clara, 1986. (visited on 21/04/2024).



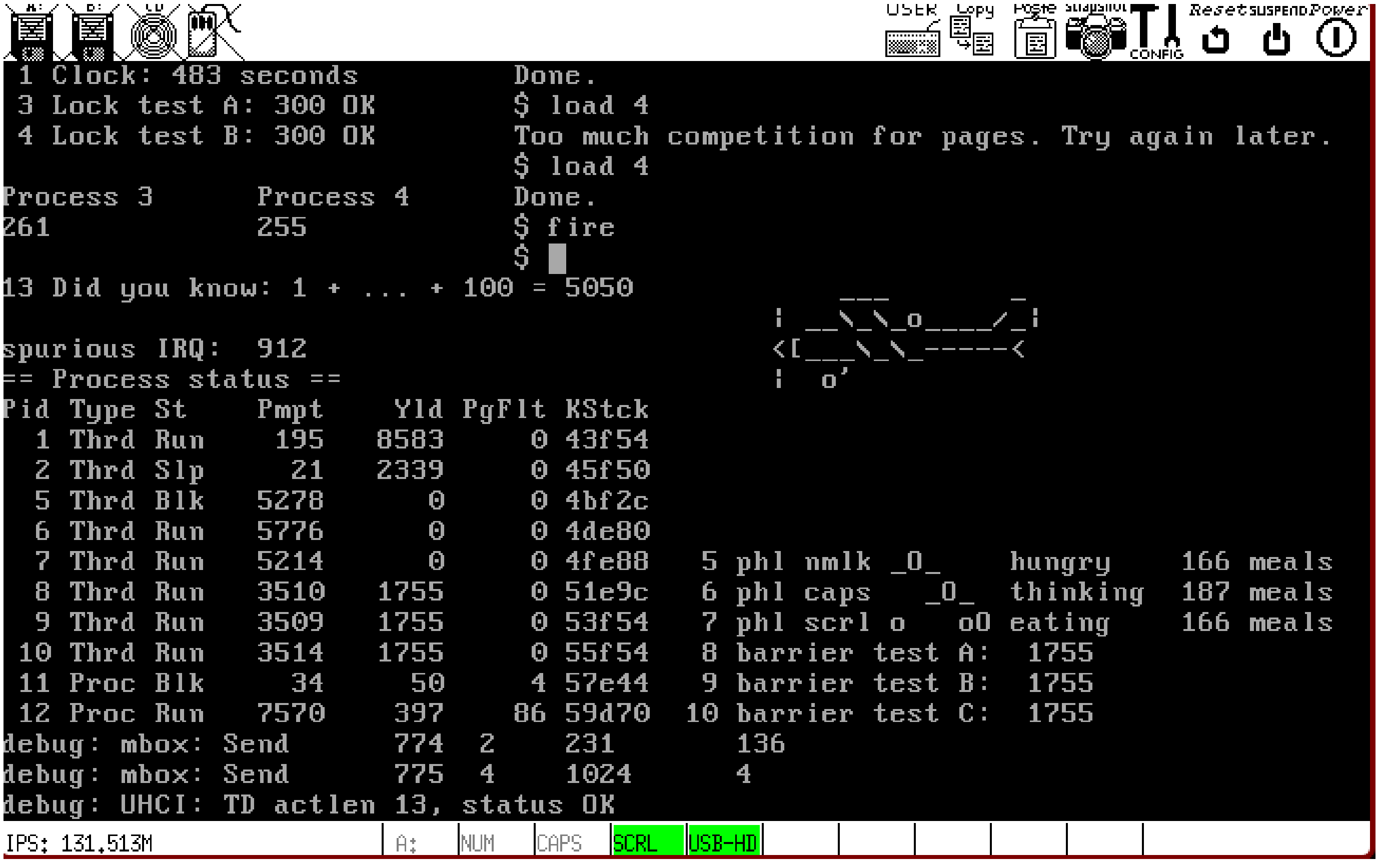
```
 1 Clock: 744 seconds         $ load 2
 3 Lock test A: 300 OK        Done.
 4 Lock test B: 300 OK        $ load 4
                              Done.
rocess 3       Process 4      $ load 3
126            123            Done.
                              $
13 Did you know: 1 + ... + 100 = 5050   _       _
                                        \_o____/_¦
spurious IRQ: 2775                      _\_-----<
== Process status ==
Pid Type St    Pmpt    Yld PgFlt KStck
  1 Thrd Run    615   8927     0 43f54
  2 Thrd Slp     22   2554     0 45f50
  5 Thrd Run   5455      0     0 4be80
  6 Thrd Run   6693      0     0 4de88
  7 Thrd Blk   5216      0     0 4ff2c   5 phl nmlk oOo    eating    224 meals
  8 Thrd Run   3786   1871     0 51ea0   6 phl caps    _O_  thinking  286 meals
  9 Thrd Run   3791   1871     0 53f54   7 phl scrl _   _O hungry    223 meals
 10 Thrd Run   3799   1871     0 55f54   8 barrier test A:  1871
 11 Proc Blk     65    106    16 57e44   9 barrier test B:  1871
 12 Proc Run   4900   2300   437 59ebc  10 barrier test C:  1871
lebug: UHCI: TD actlen 31, status OK
lebug: UHCI: TD actlen 2560, status OK
lebug: UHCI: TD actlen 13, status OK
```

Fig. 4: *Bochs output*



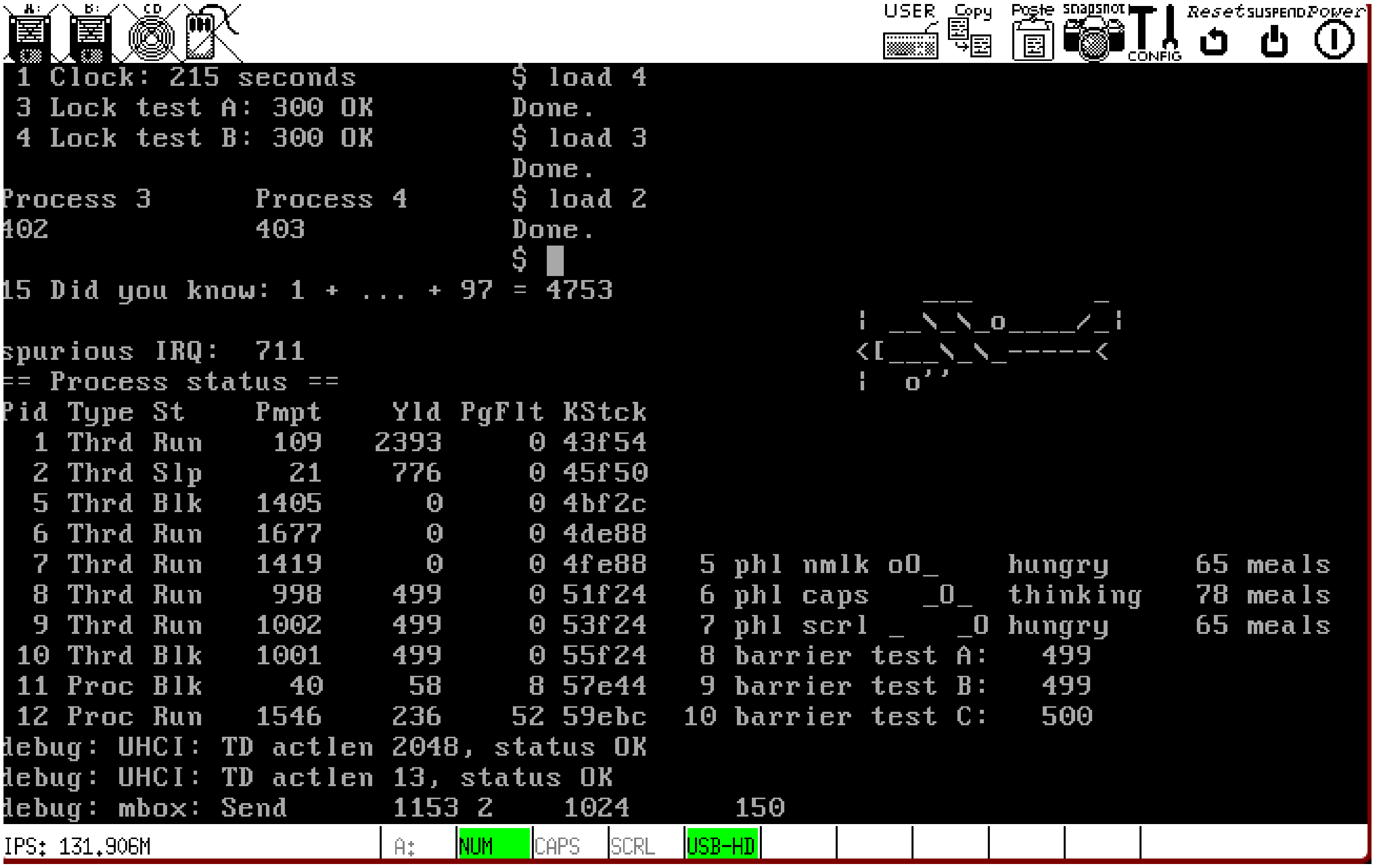
```
 1 Clock: 902 seconds         Done.
 3 Lock test A: 300 OK        $ fire
 4 Lock test B: 300 OK        $ fire
                              $ fire
Process 3      Process 4      $ fire
Done           Done           $ fire
                              $
13 Did you know: 1 + ... + 100 = 5050                                    ___
                                                                     ¦ __\_\_o___
spurious IRQ: 3209                                                   <[___\_\_---
== Process status ==                                                 ¦  o'
Pid Type St    Pmpt    Yld PgFlt KStck
  1 Thrd Run    568  11125     0 43f54
  2 Thrd Slp     21   3242     0 45f50
  5 Thrd Run   6697      0     0 4be80
  6 Thrd Blk   8191      0     0 4df2c
  7 Thrd Run   6698      0     0 4fe88   5 phl nmlk oOo    eating    287 meals
  8 Thrd Run   4676   2338     0 51f54   6 phl caps    _Oo  hungry    351 meals
  9 Thrd Run   4675   2337     0 53f24   7 phl scrl _   _O thinking  286 meals
 10 Thrd Run   4676   2337     0 55f24   8 barrier test A:  2338
 11 Proc Blk     46     50     4 57e44   9 barrier test B:  2338
 12 Proc Run   9585   1374   300 59ebc  10 barrier test C:  2338
debug: UHCI: TD actlen 31, status OK
debug: UHCI: TD actlen 4096, status OK
debug: UHCI: TD actlen 13, status OK
```

Fig. 5: *Bochs output, pinned shell, limit page competition, processes share a common kernel page table*



```
 1 Clock: 483 seconds         Done.
 3 Lock test A: 300 OK        $ load 4
 4 Lock test B: 300 OK        Too much competition for pages. Try again later.
                              $ load 4
Process 3      Process 4      Done.
261            255            $ fire
                              $ 
13 Did you know: 1 + ... + 100 = 5050            ___       _
                                             ¦ __\_\_o____/_¦
spurious IRQ:  912                           <[___\_\_-----<
== Process status ==                         ¦  o’
Pid Type St    Pmpt    Yld PgFlt KStck
  1 Thrd Run    195   8583     0 43f54
  2 Thrd Slp     21   2339     0 45f50
  5 Thrd Blk   5278      0     0 4bf2c
  6 Thrd Run   5776      0     0 4de80
  7 Thrd Run   5214      0     0 4fe88   5 phl nmlk _O_    hungry    166 meals
  8 Thrd Run   3510   1755     0 51e9c   6 phl caps   _O_  thinking  187 meals
  9 Thrd Run   3509   1755     0 53f54   7 phl scrl o   oO eating    166 meals
 10 Thrd Run   3514   1755     0 55f54   8 barrier test A:  1755
 11 Proc Blk     34     50     4 57e44   9 barrier test B:  1755
 12 Proc Run   7570    397    86 59d70  10 barrier test C:  1755
debug: mbox: Send      774  2    231       136
debug: mbox: Send      775  4    1024      4
debug: UHCI: TD actlen 13, status OK
```

Fig. 6: *Bochs output, pinned shell, limit page competition, processes share a common kernel page table. Paging occurs while many processes run.*



```
 1 Clock: 215 seconds         $ load 4
 3 Lock test A: 300 OK        Done.
 4 Lock test B: 300 OK        $ load 3
                              Done.
Process 3      Process 4      $ load 2
402            403            Done.
                              $ 
15 Did you know: 1 + ... + 97 = 4753                  ___       _
                                                  ¦ __\_\_o____/_¦
spurious IRQ:  711                                <[___\_\_-----<
== Process status ==                              ¦  o''
Pid Type St    Pmpt    Yld PgFlt KStck
  1 Thrd Run    109   2393     0 43f54
  2 Thrd Slp     21    776     0 45f50
  5 Thrd Blk   1405      0     0 4bf2c
  6 Thrd Run   1677      0     0 4de88
  7 Thrd Run   1419      0     0 4fe88   5 phl nmlk oO_    hungry     65 meals
  8 Thrd Run    998    499     0 51f24   6 phl caps   _O_  thinking   78 meals
  9 Thrd Run   1002    499     0 53f24   7 phl scrl _   _O hungry     65 meals
 10 Thrd Blk   1001    499     0 55f24   8 barrier test A:   499
 11 Proc Blk     40     58     8 57e44   9 barrier test B:   499
 12 Proc Run   1546    236    52 59ebc  10 barrier test C:   500
debug: UHCI: TD actlen 2048, status OK
debug: UHCI: TD actlen 13, status OK
debug: mbox: Send      1153 2    1024      150
```

Fig. 7: *Bochs output, unpinned shell, no limit on page competition, processes share a common kernel page table. Paging occurs while many processes run.*