previous version.) What happens next depends on the chosen editor.

For TPU, a scratch file is created and the contents of any initialisation file copied to it. Commands are added to position the cursor at the place where TEX spotted the error. The scratch file is then used as the TPU initialisation file. After exiting the editor, this scratch file is deleted.

The process for EDT is similar: an initialisation file specified via TEX$EDIT_INIT is copied to a scratch file which is used to initialise EDT. However, it is not possible to position the cursor exactly at the erroneous text automatically (EDT is somewhat lacking in this respect), only to the right line. So the command sequence GOLD M is defined, which can be used to position the column correctly by hand.

Since both TPU and EDT are callable, but only one can be used in a particular TEX session, it is obviously somewhat inefficient to have both permanently linked (they are both quite large). Fortunately, both editors are implemented as *sharable images*. This allows TEX to determine which editor to use via TEX$EDIT, then load the appropriate sharable image using the run-time library routine LIB$FIND_IMAGE_SYMBOL before invoking the editor.

With the possible exception of LSE, which is TPU-based but not available to the author, the other DEC editors are not callable, and must be invoked, as would a non-DEC editor, by spawning a DCL command. Of the non-callable editors, only TECO can position the cursor in its initialisation file. However, input to TECO is split into pages (i.e., TECO makes a single pass through the file with a buffer of finite capacity), so it is not wise to position the cursor automatically. Instead, a macro is defined in q-register '1' to perform the positioning.

Any other editor is executed with a fixed sequence of command line arguments, separated by spaces: the file to be edited; the erroneous line; the erroneous column; and the initialisation file (if any). This allows a DCL procedure to be specified as TEX$EDIT, permitting editor-specific processing. For example, the trivial procedure for use with SOS would be:

```
$ DEFINE/USER SYS$INPUT SYS$COMMAND:
$ SOS 'P1'
```

The change file and editor-specific code for TEX 2.95 can be obtained by contacting the author at either alien@uk.ac.essex.ese or alien@uk.ac.kcl.ph.ipg. Both these addresses are on JANET, the U.K. academic network. The change file also features a large (>64K) memory, to enable the production of PICTEX graphics and halftone images.

The Virtual Memory Management of PubliC TEX

Klaus Thull

Last summer in TEXeter, I promised a public domain TEX for the PC. At that time I had solved the compiler related (arithmetic and idiosyncratic) problems and had passed the trip test. For a production version, capable of LATEX, PICTEX and AMS-TEX, I still needed a Virtual Memory scheme which was promised me at TEXeter but never arrived. This I did then on my own, following some advice from "The Art of Computer Programming," tested it thoroughly, and completed a production version last autumn. For a while now this "PubliC TEX" is up and running, and has passed some few tests and productions.

This TEX does pass the trip test, I am proud to announce. On all accounts it is a fully developed specimen, capable of heavy work, and has proven reasonably stable. It can be configured with full memory and font space since these two are virtualized. The other table spaces must fit into real memory but even under Novell conditions which leave ca. 450-500kB this seems to be sufficient for generous sizes. The setting I use now has grown out of some experimenting with large runs in narrow conditions. Some of those large runs have been done with the new PubliC TEX.

As yet, this TEX is still slow. Its speed is ca. one fourth of that of its big commercial brother. On a 10MHz NCR AT-Compatible, it takes about 20 seconds for a plain page, and 30 for a LATEX page.

This TEX does not need the co-processor anymore. Since TURBOPASCAL's emulation knows only a 6-byte *real* datatype, some hand-coded conversion is used for *float* and *unfloat*.

TEX is accompanied by the complete TEXware and the complete GF- and PK-ware. MFT and METAFONT are still missing but I am working at that. I won't do anything about PXL-ware, yet I intend to do a PKtoCH/CHtoPK pair in order to have some font editing facility.

The entire sources are publicly available at LISTSERV@DHDURZ1.BITNET.

The Compiler: The compiler of my choice was Borland's TURBOPASCAL when version 4 was announced. This was the version introducing large memory model, multi-module compilation and 32-bit integers.

For once, I experienced a μ compiler which deserves the name (but then, I am a spoiled mainframe user). Where I do not have to wait many hours for a compilation. In fact, I have to wait 5 *minutes* of which 4 are `TANGLE` time. Where I didn't have to spend several weeks only to trace down compiler bugs. Where I/O, type conversion and arithmetic behavior is handled somewhat sensibly. Where code generated is decently small. TeX's size is about 180k. Imagine my tears.

Of course, there are idiosyncrasies, and one (maybe-)bug. There was, of course, *line_break*'s standard feature turning *looseness* of −2 into 65534 which is displayed by every 16bit compiler. There was one new casting bug in `tangle` which spoiled *try_break*. Some more things like that.

Nevertheless, this compiler has passed its TeX test with all jets flaming, I think. A thousand thanks to its makers.

Virtual Memory TeX: *mem* and *font_info* are the two tables virtualized. These two are the largest, and—alas—the ones accessed most, I suspect. I chose to devise a swapper governing one real memory page pool to serve both tables. This scheme might be extended to include other tables one day. Next on the list are INITeX's hyphenation generator tables since for INITeX conditions do get narrow.

For testing memory access, I had an early, small, pre-VM version of TeX, sufficient for `WEB`s, output a memory access log for *mem* and *font_info*. Then I took a couple of 10-20 page web logs as input data for statistics and simulation.

Here are some results of investigating said data as well as of experiments with the completed Virtual Memory TeX.

Basically, one printed page takes about 200000 memory accesses. This number of course grows for PiCTeX, and also for huge paragraphs. The maximum record is held by my own prime number plotter with 6 million accesses, followed by a certain PiCTeX page with 2 million accesses. About 3 to 4 consecutive accesses are on the same 256-cell memory page, in the average. This fact is yet to be exploited to construct the 'very fast' memory access.

TeX's memory access behavior itself may be deemed "semi-local" which loosely means: of a row of consecutive memory accesses, some portion of them access a locally limited area. Over the long run, the area may change, but then the new area is another locality area. In the case of TeX, the access pattern is clear: the paragraph, the formula, the one macro under construction, each make for locality of accesses; non-locally there are macro, font, and glue references.

```
 hh   ighfkj b   f  h           ah   fj
 ii   ffifil        i            g   gj
 ii     ilhjf       i            i   gk
 ii    dkld  f      h            h   gk
 ije   glie         h            g   fj
 kihf  hl f         i            h   gj
ilhkc  hiji         i            i   gj
 jilh giijg         h            h   gj
 iihg ilh c         h            i   gj
 ihgkjjji jf g      i            h   gj
 ih kkffi ig e   e  h            h   fj
eii jlh i g         i            i   gj
 ihdilkdi fg        h            j   gk
 iifgklfigdif    g  i            i   gk
 ii dikgicfkh       i            g   gk
fhefhfighhhkhgfdch  f          d    gei
dheeb  ejjcbcd
dgcihfhi ggb d
bcbbhjidebegb    b
 dbe ehfijhjed
fifiijhihghd d
ekichijhhccigi iegjia       h fa     f i
djie ehgd cihi  ghih        h f      g i
 iiceikhcdcjeh  cejf        h f      g j
 ijegkkh dejdi   ekgf       h g  h   gej
 hh gkif  cg h    je        g e      g i
dii jlhff fh i    jgg       h g      h j
 iigkkifg jicifd hjg        h gc     h j
chgchjjffcehee  degf
chgejkjffchhee ddfgf                b e
 deekligi ijghia kg            bd fi ej
cdcbjjfcf gfbbbb hb             e  c  h
ccbbhkf e ffbb   gb                   d
```

Figure U: This is the memory access log over the last three pages of a 20 page `.web` (containing the index, the section list, and the TOC) using the original memory access scheme recorded in *The TeXbook*.

Each line logs, for 10000 accesses, their distribution over the 256-cell memory pages. Each letter denotes the $\log_2$ of the number of accesses of that memory page (a:1, b:2, c:4 accesses, etc.).

This picture covers *mem*'s single node area only. The right part covers the macro area and the left part the character node area.

Investigating swap decision algorithms, the most important factor happened outside the swapper: locality gets lost over the run of printed pages. Free list gets scrambled, and after, say, five printed pages locality is virtually non-existent. To restore locality, I constructed a free list sorter. Indeed, on the PC the sorting decreased the number of page faults by 10% under favorable conditions (100 available memory pages) to 1/2 under narrow conditions (30 mem pages). Figures U and S, which

```
ljkkiaecajaflj  fh  h           ah   gj
kl       j          i            g   gj
 m       k          i            h   gk
 kl      j          h            h   fj
  m      k          i            h   gk
  lj     k          i            g   gk
   m     j          h            i   gk
   mk    j          i            i   gk
   jl    k          i            h   gk
 d  l    j  ge   e  i            h   gj
    jm   k          i            i   gk
     m   j          i            j   gk
     kl  k          i            h   gk
fe ddem  j   dfd h  i          d g  ggk
ijgc  i    dh    e                    f
  jj
   iji      d
h  c jj    dh
gjjj
   ijji     d
      h
 jjjjjj     d
jaahgaijklkkjj ifhjia       h  fa   f i
k        k   h    jh        h  f    g j
kk       k   i    jg        h  g    g j
hl       k   i    jgf       h  g h  hej
hkl      j   i    kgg       h  g    h j
hjldc a  j   i d hjg        h  g    h j
ikii  f  h    f  fgf           ec   e f
ijkf     h        gf
ijkjjjj  h        gf
klhfgajjkjkkkjidfka            bd fibei
jlc   g  b    bb i              e  f  i
hk    f          f
hjjjjkjjkjkkkj  fhgf                  d
```

Figure S: This image records the memory accesses during the same pages as in Fig. U to the same memory area, using the same recording conventions, but this time a free list sort is employed at the end of every *ship_out*.

Note the locality visibly in effect now. Note also the increased number of accesses vs. the decreased number of accessed pages per line.

---

are compiled from corresponding mem access logs, demonstrate the difference between the unsorted and the sorted case.

Also, when I installed the free list sorter on PCS Cadmus which now is a paging system, it seemed to me that throughput increased by 10% while the sorter itself adds ca. 1% of CPU time. I would like to see this measured under controlled conditions (Size of the Working Set? Number of page faults?). At PCS, I cannot do that.

Memory page size of 256 cells (which is 1k) is just the right compromise. A larger size increases the number of swaps while the average number of consecutive accesses of the same memory page never exceeds 4. A smaller size increases page translation table size which is now 3.3k (Memory page size × page translation table size = const).

When memory page size is 256 cells and memory is 512k (which is likely under Novell) then INITEX obtains 12 pages swappable memory, and VIRTEX about 120 pages. In that case, one average LaTeX run takes about 100 memory page swaps per printed page. This is a tolerable low swap rate, I think, and so I won't spend too much time in speeding up swapping.

1. PubliC TeX's Memory handler. Here the two memory handling components are given in detail since they are the central part, I think, of the PC port. The solutions presented here may transcend TURBOPASCAL, and may allow for porting the WEB-to-C stuff, which is on the tape now, to small machines. Furthermore I would like to see others improve it.

A few details are left off, like most of the **debug**s, the procedure call cross referencing needed for TURBOPASCAL's *unit* mechanism, and the *use_assembler* switch, since they just clog up the text without adding clarity.

format *debug* ≡ *begin*
format *gubed* ≡ *end*
format *stat* ≡ *begin*
format *tats* ≡ *end*
format *fakebegin* ≡ *begin*
format *fakeend* ≡ *end*

2. For a change, TURBO allows clean memory management due to an undocumented feature. This feature is not PASCAL as defined but, at least, it is cleaner than other constructs I saw used on 16bit machines. (O ye nameless compilers, get you gone into oblivion, and speedily.) If, say, $memp(x)$ is a function returning a pointer of some type, then TURBO accepts $memp(x)\uparrow \leftarrow something$, and it does the right thing. This feature comes in handy here.

define $mem(\#) \equiv memp(\#)\uparrow$
define $font\_info(\#) \equiv fmemp(\#)\uparrow$

⟨Types in the outer block 2⟩ ≡
p_memory_word = ↑*memory_word*;
mem_pc_index = 0 .. *max_mem_piece*;
mem_piece = **array** [*mem_pc_index*] **of** *memory_word*;
p_mem_index = *p_mem_min* .. *p_mem_max*;
p_fmem_index = 0 .. *p_fmem_size*;

```
mem_index = mem_min .. mem_max;
fmem_index = 0 .. font_mem_size;
```

See also section 5.

3. Here, of course, is the original reason for just that mem page size: these functions yield one-byte moves. Everything else would result in some kind of shift.

```
define mem_piece_size = 256
        { size of a memory piece, must be 256
        in order to use lo and hi }
define max_mem_piece = mem_piece_size − 1
        { min_mem_piece = 0 }
define mdiv(#) ≡ [␣hi](#)
define mmod(#) ≡ [␣lo](#)
define fdiv(#) ≡ [␣hi](#)
define fmod(#) ≡ [␣lo](#)
```

4. The swapper. TeX's *mem* and *font_info* cannot be made smaller than, say, 70000 cells or 300kB memory if TeX is to be more than a toy. Yet, with the program proper being 180k and the other tables somewhere around 200k, this clearly exceeds a PC. So some form of memory pager must be provided.

We let *mem* and *font_info* use same slot pool, same swapper, and same external memory. Later, when we build 64-bit TeX, INITeX's hyphenation generator tables and/or *eqtb* may be included. One would do this by record variants on *contents*.

We let slot space and external memory grow with use.

```
define max_slots = 300
```

5. Central Intelligence Agency is the page translating table. It contains entries for the page slot allocated (or *no_slot*) and the external memory index (or *no_page*).

```
define no_slot ≡ nil
define no_page = −1

⟨ Types in the outer block 2 ⟩ +≡
time_stamp = integer;
page_p = ↑page_rec; slot_p = ↑slot_rec;
page_rec = record
        { page translation table record }
  slot_ptr: slot_p;  { pointer into slot allocated,
        or no_slot }
  ext_nr: no_page .. 511;
        { pointer into external memory }
  end;
slot_rec = record
        { inline memory page descriptor }
  page_ptr: page_p;
        { pointer into page allocated }
  follower: slot_p;  { to simplify traversing }
  stamp: time_stamp;  { or the RU-bit }
  contents: mem_piece;  { the actual page }
  end;
```

6. These are the page translation tables for *mem* and *font_info*.

```
⟨ Locals for virtual memory handling 6 ⟩ ≡
p_mem: array [p_mem_index] of page_rec;
p_font_info: array [p_fmem_index] of page_rec;
slot_rover: slot_p;
new_slot: slot_p;
slot_count: 0 .. max_slots;
      { number of slots allocated hitherto }
page_count: 0 .. 511;  { number of external pages
      allocated so far }
clock: integer;
  stat swap_no: integer; tats
```

See also sections 7 and 20.

7. Our external memory is on disk. We collect all the operations at this place so you can devise something different if you so wish.

```
define write_ext_mem_end(#) ≡
      fakebegin write(mem_file, #)
      end
format write_ext_mem_end ≡ end
define write_ext_mem(#) ≡
      begin seek(mem_file, integer(#));
      write_ext_mem_end
define read_ext_mem_end(#) ≡
      fakebegin read(mem_file, #)
      end
format read_ext_mem_end ≡ end
define read_ext_mem(#) ≡
      begin seek(mem_file, integer(#));
      read_ext_mem_end
define open_ext_mem ≡ set_ext_mem_name;
    assign(mem_file, name_of_file);
    rewrite(mem_file);
    write_ext_mem(0)(new_slot↑.contents);
define close_ext_mem ≡ close(mem_file)

⟨ Locals for virtual memory handling 6 ⟩ +≡
mem_file: file of mem_piece;
```

8. It is convenient to pre-allocate one slot to a convenient page, probably the *mem_bot* page which is the first one to be accessed anyway. Actually, external memory must be opened here.

```
define for_all_mem_pages_do ≡
    for i ← p_mem_min to p_mem_max do
format for_all_mem_pages_do ≡ xclause
define for_all_fmem_pages_do ≡
    for i ← 0 to p_fmem_size do
```

```
  format for_all_fmem_pages_do ≡ xclause
  define first_page_no ≡ p_mem_min
⟨ Get virtual memory started 8 ⟩ ≡
  { initialize entire system }
for_all_mem_pages_do
  with p_mem[i] do
    begin ext_nr ← no_page; slot_ptr ← no_slot
    end;
for_all_fmem_pages_do
  with p_font_info[i] do
    begin ext_nr ← no_page; slot_ptr ← no_slot
    end;
  { now allocate first slot }
new(new_slot);
with new_slot↑ do
  begin follower ← new_slot;
  page_ptr ← addr(p_mem[first_page_no]);
  stamp ← 0;
  end;
slot_rover ← new_slot; slot_count ← 1;
  { and connect it to first page, also aquire first
    page of external memory }
open_ext_mem;
with p_mem[first_page_no] do
  begin slot_ptr ← new_slot; ext_nr ← 0;
  end;
page_count ← 1;
clock ← 0;
stat swap_no ← 0; tats
```

See also section 21.

9. We investigated five different swapping algorithms. Essentially, they are variants of the *First In First Out* (FIFO), the *Least Recently Used* (LRU) and the *Not Recently Used* (NRU) algorithms.

- The FIFO algorithm throws out the page which has been in memory longest.
- The LRU algorithm sets a time stamp per access and, in case of swapping, the slot with lowest stamp is thrown out. The subcases concern resetting of timestamp at swap time.
- The NRU algorithm sets a stamp per access and, in case of swapping, looks for a null stamp and clears a selection of stamp. The subcases concern the nature of that selection.

10. Only two of the algorithms studied so far turned out to be worthwhile, namely the LRU without clock reset, and the NRU following Knuth's modification. So these two we keep. The NRU showed ca. 5% more page faults than the LRU but is a trifle faster in the non-page-fault access. So in case there are few page faults and/or a fast swapper, NRU might prove the faster, else LRU—contest is still open.

Objection to LRU may be the fear of the clock overflowing with huge or intricate jobs. The simple WEB file I logged showed ca. 200000 accesses per printed page, and, while I still wait for a chance to log a large table or a PiCTeX job, let's assume 1000000 accesses for a page at the worst, and you still have two thousand pages to go! Which leaves one to meditate on the magnitude of a 32-bit integer.

A variant not investigated yet is to step the clock at swap time only.

As it turned out, a PiCTeX page does take about two million accesses, and my own Third Root of Unity Primes Generator took *six* million accesses (and 12000 swaps).

```
  define use_LRU ≡
  define use_LRU_end ≡
  define use_NRU ≡ @{
  define use_NRU_end ≡@}
  format use_LRU ≡ begin
  format use_LRU_end ≡ end
  format use_NRU ≡ begin
  format use_NRU_end ≡ end
```

11. These procedures describe the basic, non-swap access which must be fast. So I use **with** to stress that fact. Actually, this might be done in assembler, and *page_ptr* and *slot_ptr* kept in a register for further reference.

```
  define not_in_memory ≡ (slot_ptr = no_slot)
  define access_it(#) ≡
        begin { at this point, slot_ptr points
              to the in-memory page }
        # ← addr(contents[mmod(p)]);
        use_LRU stamp ← clock;
            use_LRU_end
        use_NRU stamp ← 1; use_NRU_end
        end
⟨ Include system and memory management
      here 11 ⟩ ≡
⟨ I need fetch_mem here 13 ⟩
function memp(p : pointer): p_memory_word;
  begin use_LRU incr(clock);
  use_LRU_end
  with p_mem[mdiv(p)] do
    begin if not_in_memory then
      fetch_mem(addr(p_mem[mdiv(p)]));
    with slot_ptr↑ do access_it(memp);
    end;
  end;
function fmemp(p : pointer): p_memory_word;
```

```
begin use_LRU incr(clock);
use_LRU_end
with p_font_info[mdiv(p)] do
  begin if not_in_memory then
    fetch_mem(addr(p_font_info[fdiv(p)]));
  with slot_ptr↑ do access_it(fmemp);
  end;
end;
```

See also sections 17 and 18.

12. We describe the basic operations for swapping. Note the nesting of **with** clauses making for simpler expressions and (hopefully) faster programs.

```
define secutor(#) ≡ #↑.follower
define more_slots ≡ ((slot_count <
        max_slots) ∧ (mem_avail > 10000))
define fakerepeat ≡
        { syntactic sugar for WEAVE }
define fakeuntil ≡
format fakerepeat ≡ repeat
format fakeuntil ≡ until
define rove_all_slots ≡
      begin s ← slot_rover;
      repeat with s↑ do
      fakeuntil
      fakeend
define rove_slots_begin ≡
      begin fakeend
define rove_slots_end ≡
      fakebegin fakerepeat fakebegin end;
        s ← secutor(s);
      until s = slot_rover
      end
format rove_all_slots ≡ xclause
format rove_slots_begin ≡ begin
format rove_slots_end ≡ end
define out_it ≡
      with page_ptr↑ do
        begin stat incr(swap_no); tats
        write_ext_mem(ext_nr)(contents);
        slot_ptr ← no_slot   { disconnect page
            from this slot }
        end
define in_it(#) ≡
      with #↑ do
        begin   { argument is a page pointer,
            slot is on slot_rover }
        slot_ptr ← slot_rover; page_ptr ← #;
            { connect new page }
        if ext_nr ≠ no_page then
          read_ext_mem(ext_nr)(contents)
        else begin write_ext_mem(page_count)
              (contents);
          ext_nr ← page_count;
          incr(page_count);
          end
        end
```

13. This describes the outline of the swapping procedure. It is not required to be streamlined if swaps are minimized since slow anyway. Yet some indication is, again, given by the use of **with**.

```
⟨I need fetch_mem here 13⟩ ≡
procedure fetch_mem(p : page_p);
  var min_stamp: time_stamp; s, t: slot_p;
    i: integer;
  begin if more_slots then
    begin ⟨Fetch a new slot, let slot_rover point
        to it 14⟩;
    with slot_rover↑ do in_it(p);
    end
  else begin   { decide which page to throw out,
        let slot_rover point to it }
    use_LRU ⟨Use the LRU 15⟩; use_LRU_end
    use_NRU ⟨Use the NRU 16⟩; use_NRU_end
        { up til now, nothing happened except
        slot_rover moving around }
    with slot_rover↑ do
      begin out_it;
          { the old page, that is. We assume, as
          in our TeX, that we cannot discern
          between read and write accesses }
      in_it(p);   { the new one }
      end;
    end;
  end;
```

This code is used in section 11.

14. This allocates a new slot.

```
⟨Fetch a new slot, let slot_rover point to it 14⟩ ≡
begin new(new_slot);
with new_slot↑ do
  begin follower ← secutor(slot_rover);
  slot_rover↑.follower ← new_slot;
  end;
incr(slot_count);
    { now the new slot is officially present }
slot_rover ← secutor(slot_rover);
end
```

This code is used in section 13.

15. Least recently used.

```
⟨Use the LRU 15⟩ ≡
begin min_stamp ← clock; t ← slot_rover;
rove_all_slots
  rove_slots_begin if stamp < min_stamp then
    begin min_stamp ← stamp; t ← s;
    end;
```

```
    rove_slots_end;
  slot_rover ← t;
  end
```

This code is used in section 13.

16. Not recently used. We realize Knuth's suggestion to switch off used-bits for those pages only that are touched during the search process. Pages whose bits stay on then may be termed "recently recently used."

```
    define recently_used(#) ≡ (#↑.stamp ≠ 0)
    define un_use_it(#) ≡ #↑.stamp ← 0
  ⟨Use the NRU 16⟩ ≡
  begin slot_rover ← secutor(slot_rover);
  while recently_used(slot_rover) do
    begin un_use_it(slot_rover);
    slot_rover ← secutor(slot_rover);
    end;
  end
```

This code is used in section 13.

17. At this place, external memory should be closed, deleted, freed or whatever. We output statistics.

```
  ⟨Include system and memory management
      here 11⟩ +≡
procedure close_mem;
  begin close_ext_mem;
  stat wlog_cr;
  wlog_ln('took␣', swap_no : 1, '␣swappings␣for␣',
      clock : 1, '␣accesses␣on'); wlog('␣␣␣␣␣',
      page_count : 1, '␣memory␣pages␣and␣',
      slot_count : 1, '␣slots.');
  tats
  end;
```

18. Reorganizing the free lists. When we consider the various nodes strung out sequentially as allocated from the free lists then TEX's access is kind of local most of the time. It is clear: One paragraph of text is under consideration in one period of time, one formula, one batch of finished lines. In a paging environment (and most of the machines are today), such locality is an advantage: Consider the "Working Set", the collection of memory pages accessed during a certain period of time. With good locality, the Working Set needs be small only, and page faults few.

For TEX and other programs with similar memory management, the free list tends to be scrambled and scattered during the first few pages already so that any locality will be non-existent at all. Thus the Working Set may grow about a third again as large. The solution is to reorganize the free list(s) at certain times such as to reflect physical neighbourhood again.

This amounts to a Sort. A full sort, however, is out of question, it may take up to 16 sweeps through the list. It is not necessary even, since there is no harm in a scramble inside a memory page. So we do one sweep with as many buckets as there are memory pages, then recombine. What follows, then, is straightforward. (Really? I *did* crash. Where, Dear Reader, I won't tell you. You find out as an exercise.)

The proper place for this to be inserted is right after the grand *free_node_list* at the end of *ship_out*.

```
    define mem_page(#) ≡ mdiv(#)
  ⟨Include system and memory management
      here 11⟩ +≡
procedure reorganize_free_lists;
  var p, q, r, s, t: pointer; this_tail: pointer;
    i, a_p: p_mem_index;
        {indices of memory pages}
    v_p_min, v_p_max, s_p_min, s_p_max:
        p_mem_index; {the single and variable
        free list maximum page indices found so
        far}
  begin debug check_mem(false);
      {we suppose memory to be OK at this
      point, I simply want the was_free bits set for
      checking later}
  gubed ⟨Initialize free list reorganization 22⟩;
  ⟨Distribute variable size free list to the separate
      slots 23⟩;
  ⟨Recombine variable size free list 24⟩;
  ⟨Distribute single word free list 25⟩;
  ⟨Recombine single word free list 26⟩;
  debug check_mem(true);
      {Any non-trivial output here would mean
      trouble, but, as it turned out, the program
      crashed before reaching this point}
  gubed
  end;
```

19.

```
    define mem_page_avail ≡ m_p_avail
          {avoiding identifier conflict}
    define mem_page_tail ≡ m_p_tail
```

20.

```
⟨Locals for virtual memory handling 6⟩ +≡
mem_page_avail, mem_page_tail: array
      [p_mem_index] of pointer;
```

21.

```
⟨Get virtual memory started 8⟩ +≡
  for i ← p_mem_min to p_mem_max do
    begin {prepare the mem page buckets}
    mem_page_avail[i] ← null;
    mem_page_tail[i] ← null;
    end;
```

22. ⟨ Initialize free list reorganization 22 ⟩ ≡

```
p ← get_node('10000000000);
    { re-merge them first thing right away }
v_p_min ← mem_page(mem_end);
s_p_min ← mem_page(mem_end);
v_p_max ← mem_page(mem_min);
s_p_max ← mem_page(mem_min);
```

This code is used in section 18.

23. It appears that *rover* is not supposed to be empty ever.

```
define insert_first_var_per_page ≡
        begin mem_page_avail[a_p] ← p;
        rlink(p) ← p; llink(p) ← p;
        end
define insert_var_per_page ≡
        begin r ← mem_page_avail[a_p];
        s ← llink(r); rlink(s) ← p;
        llink(p) ← s; rlink(p) ← r;
        llink(r) ← p;
        end
```

⟨ Distribute variable size free list to the separate slots 23 ⟩ ≡

```
p ← rover;
repeat q ← rlink(p); a_p ← mem_page(p);
  if v_p_min > a_p then v_p_min ← a_p;
  if v_p_max < a_p then v_p_max ← a_p;
  if mem_page_avail[a_p] = null then
    insert_first_var_per_page
  else insert_var_per_page;
  p ← q;
until p = rover;
```

This code is used in section 18.

24. We clean up carefully behind us. One of those buckets may be reused very soon.

```
define append_this_var_list ≡
        begin r ← mem_page_avail[i];
        s ← llink(r); mem_page_avail[i] ← null;
        t ← llink(rover); rlink(s) ← rover;
        llink(rover) ← s; rlink(t) ← r;
        llink(r) ← t;
        end
```

⟨ Recombine variable size free list 24 ⟩ ≡

```
rover ← mem_page_avail[v_p_min];
mem_page_avail[v_p_min] ← null;
if v_p_max > v_p_min then
  for i ← v_p_min + 1 to v_p_max do
    if mem_page_avail[i] ≠ null then
      append_this_var_list;
```

This code is used in section 18.

25. This must be considered part of the inner loop since every single character freed after printing gets through here.

```
define insert_first_avail_per_page ≡
        begin mem_page_avail[a_p] ← avail;
        mem_page_tail[a_p] ← avail;
        link(avail) ← null;
        end
define insert_avail_per_page ≡
        begin r ← mem_page_avail[a_p];
        mem_page_avail[a_p] ← avail;
        link(avail) ← r;
        end
```

⟨ Distribute single word free list 25 ⟩ ≡

```
while avail ≠ null do
  begin q ← link(avail);
  a_p ← mem_page(avail);
  if s_p_min > a_p then s_p_min ← a_p;
  if s_p_max < a_p then s_p_max ← a_p;
  if mem_page_avail[a_p] = null then
    insert_first_avail_per_page
  else insert_avail_per_page;
  avail ← q;
  end
```

This code is used in section 18.

26. This code works even if *avail* has been empty in the first place.

```
define append_this_avail_list ≡
        begin r ← mem_page_avail[i];
        link(this_tail) ← r;
        this_tail ← mem_page_tail[i];
        mem_page_avail[i] ← null;
        mem_page_tail[i] ← null;
        end
```

⟨ Recombine single word free list 26 ⟩ ≡

```
avail ← mem_page_avail[s_p_min];
this_tail ← mem_page_tail[s_p_min];
mem_page_avail[s_p_min] ← null;
mem_page_tail[s_p_min] ← null;
if s_p_max > s_p_min then
  for i ← s_p_min + 1 to s_p_max do
    if mem_page_avail[i] ≠ null then
      append_this_avail_list;
```

This code is used in section 18.