PROJECT MAC November 20, 1973

Computer Systems Research Division Request for Comments No. 44

A STRUCTURED PROGRAM DESCRIPTION OF MULTICS PAGE CONTROL

by Bernard Greenberg

This document was written as Appendix A to my forthcoming thesis. "An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory". It presents the functioning of the fault and interrupt sides of the Multics virtual memory management algorithm as it existed in May, 1973, on the Honeywell 645.

Since it was developed as a part of my thesis, it does not display some of the fairly involved mechanisms within page control relating to segment control or error handling. Within the paths shown here, this results in only a few small omissions.

I have taken the liberty of creating a new language to explain page control which I explain within. I feel that this language conveys the general class of manipulations described herein with a maximum of clarity and succinctness.

I have used names for objects, in many cases, which are more mnemonic than the original names of corresponding quantities in Multics. A table is provided to assist in correlating the two. I have also made minor modifications to control flow, and subroutinized routines which were originally not subroutines where I felt that clarity would be aided. In any case, the algorithm as given is functionally identical to the actual assembler-code algorithm in use at the time of the experiment, with respect to state, sequencing, and side-effects.

The plus sign (+) denotes references to routines explained in detail elsewhere in this document.

Table of Contents of Appendix A

A Brief Overview

Multics manages both core and drum (the latter known as the "paging device", or "pd") by approximations to the least-recently-used algorithm. Two lists, the core used list and the paging device used list are maintained for this purpose, the top of each list designating the least recently used, page (which is the best choice for replacement), and the bottom of each list designating the most recently used page (which is the worst choice for replacement) on the respective device. How these lists are maintained can best be learned by reading the program that we have provided. The core used list contains logical descriptions of core frames, including pointers to descriptions of logical pages and/or paging device records when such entities may be associated with the core frame. Similarly, the paging device used list contains logical descriptions of paging device records, including pointers to descriptions of logical pages and core frames, when such entities may be associated with the paging device record.

Multics tries to maintain copies of the most recently used P pages (where P is the size of the paging device, in records) of the storage system on the paging device. The most recently used C pages (where C is the size of core memory in page frames) are to be in core, as well. (It is assumed that C is less than P).

Thus, pages being ousted from core may be written to the paging device, even if a good copy exists on disk. This fact should be kept strongly in mind when reading "try_to_write_page". Except for the case where the paging device has no copy, pages which are identical to pages in secondary storage are never written out. Pages or zeros are never written out, but their logical description is so modified that they are created in core when faulted on.

The processor hardware maintains usage information about a logical page in a hardware descriptor. Specifically, the occurence of usage and/or modification is noted in the descriptor.

A page fault is resolved by finding a page of core into which to bring the page, and bringing it in. Finding a page of core consists of reorganizing the core used list to reflect the latest usage information, and finding the least recently used page frame, and using it. Pages which have benn marked as modified cannot be claimed in this way, but are written out. When the writing is complete, at some future time, the page will be in the same state as a page which has not been recently used or modified, and will be claimed in the handling of some future page fault. Note that this 'writing' consists of initiating the physical operation, but not waiting for it to complete. It is at this writing time that secondary storage is allocated,and pages containing zeros are noted. It is at the time that zero pages are noted and that secondary storage is deallocated, At the beginning of page fault handling, housekeeping is performed on the paging device, which consists of trying to insure that at least ten records are either free or in the process of being freed. This is done by removing as many of the least recently used pages on the paging device as necessary. When a page is so moved, it is checked (via software-maintained switches) to see if it is identical to a copy on disk. If so, it may simply be deallocated from the paging device. If not, a sequence known as a read-write-sequence (rws) must be performed. This sequence consists of allocating a page of core to be used as a buffer, reading the page into it from the paging device, writing it to disk, and deallocating the paging device copy. The core buffer is then freed.

A page fault which occurs on a page for which a read-write sequence is in progress causes an event known as as rws-abort to occur. The freeing of the buffer page and the paging device page are inhibited, and the buffer page is used as the core copy of the page, and the fault is resolved.

An Explanation of the Language used to Express this Description

The language which we have used to describe Page Control is a bastardization of PL/I, with new primitives for some basic operations (enqueue, masked procedures, etc.), and an Algol 68-like formalism for represeenting relationships among structured entities.

Underlined words are language keywords. Lower-case identifiers represent names of subroutines, functions, or labels. Identifiers beginning with an upper-case character represent references to cells, which will be described below. Statement syntax is essentially the same as PL/I, but ":=" is used for assignment, and "=" is used to test equality. There is no lexical nesting of procedure or begin blocks.

A program consists of begin blocks, entered from the outside world in some unspecified way, procedures and functions, and declarations. declare (dcl) declarations may appear anywhere, including outside of blocks, and are global in scope. They define the class and type of variables, and the types of Objects used by the program. local declarations appear within blocks, and define a local scope of variables, identical to that produced when a variable is used as a formal parameter in a procedure or function.

The point of this language is to associate cells with values. The domain of values is the space of Objects. Objects are unique. Two cells have equal values if and only if their values are the same Object.

There are three classes of Objects: primitive Objects, structured Objects, and set Objects. Within each class, there are different types of Objects. Objects have no names. Only primitive Objects can be referred to explicitly, i.e., other than by reference to a cell having the desired Object as a value, or a function returning the desired Object.

Primitive Objects can be of three types. The first is boolean. There are exactly two boolean Objects. One can be referred to explicitly as true, the other false. The second is arithmetic. There is a first-order infinity of these objects, which are actually the integers. They can be referred to explicitly as 75, 16777216, -283, etc. The third is literal. They are simply arbitrary primitive Objects, whose only useful property is their uniqueness. They can be referred to explicitly as "foo", "bar", "no stuff", etc. They are not character strings in any sense, but simply unique primitive Objects of type literal.

Structured Objects consist of a finite number of cells. Any cell can have as a value only one type of Object (implied is one class, as well.) These cells are called components of the Object. These cells do have names, and they are specified in a declaration which describes the concerned type of structured Object.

Set Objects consist of an ordered set of Objects of the same type and class. All references except enqueue and dequeue, however, consider the set Object as unordered. One can add to or enqueue to a set Object, remove from or dequeue from it, ask if a given Object is a member of it, or cause a cell to be assigned successive values, each value being a different Object in the set Object, in no particular order.

Variables are the other type of cell. A variable can hold only one class and type of Object, just like the other type of cell, the structured Object component.

Assignment (performed by ":=" operation in do statements and assignment statements) consists of replacing the value of a cell with another value, i.e., changing the value of the cell. The Object which was the previous value is neither changed nor destroyed in any way.

Binding consists of saving the value of a variable when a procedure, function, or begin block is entered, and restoring it when it is exited. The latter operation is called unbinding. All assignments and bindings made between the time a variable is bound and the corresponding unbinding have a transparent effect when the block performing the binding is exited. A local declaration of a variable in a block causes such a binding to take place for that variable when the block is entered, and the corresponding unbining. Binding also takes place for variables used a s formal parameters to procedures and functions. In this case, after the old value is saved, the value of the corresponding formal argument is assigned to the variable. Hence, all calls may be seen as "call by value".

To refer to an Object, one can either refer to a cell containing it, or, if it is primitive, one can refer to it explicitly. To refer to a variable, simply state its name. To refer to a component of a structured Object, state its component name, an open parenthesis, a reference to the structured Object, and a close parenthesis.

An assignment is a reference to a cell, ":=", and a reference to an Object of the same type an class declared for that cell.

Variables need not be declared. The defaultclass of any cell is structured, with a type the same as its name. The syntax for a structured object type declaration is as follows :

[ ] : optional
{ } : select one

{declare / dcl} structured Foo (compdcl-1 , compdcl-2, ...compdcl-n);

The compdcls, or component declarations, are of the same syntax as variable declarations, except that the name is the name of the component, and the optional keyword variable is illegal.

The syntax for a variable or structured Object component declaration is as follows:

{declare / dcl} [variable] Foo [type] [objtyp] where objtyp is either boolean,literal, arithmetic , any structured Object type named in a structured Object type declaration, or set objtyp, where objtyp is, recursively enough, any possibility named in this sentence.

local declarations only name their variable, although they can declare its type as well.

do statements differ from PL/I in that any cell can be used to the left of the ":=", not necessarily variables. Theparticular form "do Foo := range Bar" means that the value of Bar is a set object, and the do is to iterate over each Object therein, in no special order.

The special constructor function construct is used to create new structured objects. The syntax of a reference to it is

construct Foo (compname-1:object-1,compname-2:object-2,...),

whose value is the new Object.

The unique Object "null" can be used as a value of any cell. It has all types and classes.

The predicate void takes as an argument a reference to a set Object, and returns true or false (boolean Objects), depending on whetheror not it is empty. The operators "=" and "≠" may be used to test if two references are equal, i.e., refer to the same Object. An appropriate boolean Object is returned as a value. The operators "or" , "and", and "not" operate on boolean Objects in the obvious way. The conventional arithmetic operators operate upon arithmetic Objects, returning an arithmetic Object with the expected value.

if statements have as their predicate a reference to a boolean Object.

A call statement consists of the word call followed by either a procedure name and an optional argument list or a complex function reference and an argument list. An argument list is a parenthesized list of (possibly zero) references to Objects separated by commas. A complex function reference is a function reference to some outside-of-the-language function which will return as a value a procedure, which one depends on the arguments to the function, which will be called by the call statement,with the arguments to the call.

The evaluation of arguments in or and and is conditional, as in Lisp 1.5 (   ) and proceeds from left to right.

A Program to Find the Man Who owns the Black House, and Have Him and His Father Switch Houses.

```
declare  structured Person
      (Father  type Person,
       House);
declare   structured House
      (Color literal,
       Owner  type Person );
declare  Son Person, House2 House;
declare Brooklyn set  House;    /*assumed to be initialized */

switch_houses:begin;
    do House := range Brooklyn;          search the set "Brooklyn"
        if Color(House)  = "black" then do;    found him.
            House2 := House(Father(Owner(House)));    find the other house.
            Son := Owner(House);                        remember who is the son.
            House(Son)  := House2;                     Son now owns House 2.
            Owner(House2)  := Son;
            House(Father(Son))  := House;              Father owns house.
            Owner(House)  := Father (Son);
            return;
        end;
    end;
end;
```

A Top-Level Programmatic View of Page Control Activity

A page fault causes the following: (page_fault)
- The paging device is housekept.
- Transient conditions such as i/o in progress or an rws on the faulted page are noticed and handled.
- A free page is claimed, and the faulted page is read or created into it.
- If i/o was started, the page is waited for.

Finding a free page consists of the following: (find_core)
- The core used list is searched for a good candidate.
- Recently used pages are not good candidates. They are skipped, and re-judged as not-so-recently used for next time.
- Pages which have been modified (stored into) cannot be claimed now. They are written out, and re-judged as not to have been modified.
- A page which has not been modified, and has been used approximately less recently than any other page, is pre-empted from its core frame, and this core frame is the new free page frame.

Writing a page out consists of the following: (write_page)
- The page's contents are checked, and if all zeros, the page is flagged as not needing to be read or written. No writing takes place, and disk and paging device space allocated to the page are freed.
- The page is given a residence on disk, if it does not already have one.
- The page is given a residence on the paging device, if it does not already have one, and one is available.
- The page is written out to its residence on the paging device, if it has one, otherwise to disk. The completion of i/o is not waited for.

Housekeeping the paging device consists of the following: (get_free_pd_record).
- An attempt is made to insure that there are ten paging device records free or being freed, which is done as follows:
- The pd used list is searched for a good candidate to pre-empt.
- The search is made starting at the least-recently used pd record.
- Records which contain pages in core are recently used. They are re-judged as such and skipped.
- Records containing pages identical to pages on disk are acceptable. The pages in them are pre-empted, and the record is now free.
- Other records have to be written back to disk, which is done by performing a read-write sequence (rws) on them.

Performing a read-write sequence on a page consists of the following: (start_rws, rws_done)
- A free page of core is obtained.
- The page is read into it from the paging device.
- When the read is completed, the page is written out to the disk.
- When the write is completed, the page of core and the paging device record are freed.
- A page fault on the page involved in the sequence at any point during it causes the sequence to be aborted at the next complete operation in the sequence, and the core page is used as the page's home in core.

A Top-Level Description of the Objects Used by Page Control

A Page Object is the logical description of some page of the storage system, as opposed to a page frame on some device.

A Descriptor Object, in actuality a "page table word", is the physical descriptor by which a processor accesses a page. It contains a core address, usage bits, and a bit which causes a fault when off.

A Coreadd Object describes a physical core block. It describes the status of this block, including, implicitly, its position in the core used list.

A PDrec Object describes a paging device record, or frame. It describes the status of this frame, including, implicitly, its position in the paging device used list.

A Devadd Object represents a physical disk or drum address, and its contents. Included in this object is an identification of the device on which this page frame resides.

An Io-status Object is a hardware-generated object, which describes an input-output operation which has completed.

An Io-program Object is a sequence of commands for the system i/o controller to give to an i/o device. It specifies the type of operation required, the record within the device concerned, and a core address concerned.

A Trace-Datum Object is a recorded datum of information about traffic between disk and core/drum, for the purpose of the thesis experiment.

A Description of the Object Types used in Page Control

Recall that the default type of a structured Object component is the same as its name.

dcl structured Page	Represents a page of some segment of Multics, as opposed to a page of core or some device.
(Descriptor,	The hardware descriptor by which processors access the contents of the Page.
Devadd,	The physical disk or pd address from which Page should be read or written to. If On_pd is true, is a pd address. Otherwise, it is a disk address. A Devadd of "null", however, represents a page full of zeros.
Coreadd,	The Core frame associated with this page. Valid only when Addressable (Descriptor (Page)) is true.
PDrec,	If On_pd is true, this is the pd record used by Page.
Event literal,	Some literal quantity unique for each Page. Used to identify the occurence of events associated with this Page in interprocess signaling.
Io_in_progress boolean,	Truth indicates i/o in progress, or at least not known to have completed, on Page.
On_pd boolean,	Specifies that Page has an allocated pd record.
Wired boolean,	Indicates that Page must always remain addressable.
Gtpd boolean);	Indicates that Page is forbidden to go on pd, for system safety reasons.
dcl structured Descriptor	Represents a page-table word, the physical descriptor by which processors access a page.
(Phys_Coreadd arithmetic,	The physical core address occupied by the page to which this descriptor belongs. Valid if and only if Addressable is true.
Addressable boolean,	Truth allows the processor to use the Phys_Coreadd. Falsity causes the procedure page_fault to be executed.
Usage boolean,	Set by the hardware whenever this descriptor is used, or more accurately, fetched into the associative memory.
Modified boolean);	Set by the hardware whenever a store-type operation is performed using this descriptor, or an associative memory copy therof. See the comment in write_page.
dcl structured Coreadd	Represents a core page frame.
(Page,	"null" represents an unallocated page frame. Otherwise, the Page contained in this frame. This is only for normal page-holding use, not rws's.
Phys_Coreadd arithmetic,	The physical core address represented by this frame.
Next type Coreadd,	The next more recently used core frame.
Io_read_or_write literal,	If Io_in_progress(Page(Coreadd(--))) is true, or Rws_in_frame, tells which direction of i/o is being performed.
Rws_in_frame boolean,	Signifies an rws in progress in this frame.
PDrec);	Used only if Rws_in_frame is true. Locates PDrec of this rws.

dcl structured Devadd	Represents a physical device address.
(Device literal,	Identifies a secondary storage device.
Phys_devadd arithmetic);	Identifies some physical record number on a physical device.
dcl structured PDrec	Represents a paging device (pd) record.
(Page,	If In_use is true, describes the page on this record.
Diskaddr type Devadd,	If In_use is true, describes the disk address occupied by our Page.
Devadd,	The pd address of this pd record.
Next type PDrec,	Describes the next more recently used pd record.
Coreadd,	When Rws_in_progress is true, describes the core frame being used as an rws buffer.
Event literal,	A unique literal associated with this pdrec. Used to identify this PDrec in interprocess signaling.
In_use boolean,	Tells if PDrec is in use or free.
Rws_in_progress boolean,	Signifies that an rws or rws abort is in progress in this PDrec.
Incore boolean,	Signifies that the page in this PDrec is in core right now. Used for maintaining LRU ordering.
Modified_from_disk boolean,	Truth indicates that the pd copy of Page is different than the disk copy.
Abort_flag boolean,	Turned on to start an rws abort by some process faulting on an rws'ing page.
Abort_complete boolean);	Signifies that post_page (q.v.) has aborted an rws, and a cleanup is expected.
dcl structured Io_status	Represents a completed i/o operation in an i/o control routine.
(Phys_devadd arithmetic,	The physical device address which participated in this operation.
Phys_coreadd arithmetic,	The physical core address which participated in this operation.
Io_program,	
Coreadd);	The Coreadd Object associated with Phys_Coreadd. Although not actually present, the one-to-one mapping between these entities lets us use this here for clarity.
dcl structured Io_program	A portion of a channel program.
(Direction literal,	Indicates read or write.
Phys_Devadd arithmetic,	Physical device address involved.
Phys_Coreadd arithmetic,	Physical core address involved.
Next type Io_program);	Next program in channel queue.
dcl structured Trace_datum	An item of trace data for the experiment.
(Devadd,	The disk address concerned.
Type literal);	The direction of motion. Can be "read", "write","virtual","pd virtual","delete".

The Global Variables used by Page Control

dcl

Page_table_lock literal, — A quantity used to insure that only one processor at a time is in page control. A process desiring to "hold" this lock loops continuously until it is unlocked, and then locks it.

CoreTop type Coreadd, — The least recently used Coreadd Object.

Writes_outstanding arithmetic, — The number of write operations started which have not yet been known to complete. Used as a heuristic to call post_any_io.

Rws_active_count arithmetic, — The number of read-write sequences which have been initiated and not yet known to be completed.

Number_of_free_pd_records arithmetic, — The number of paging device records free or in the process (rws) of being freed.

Top_of_pd_used_list type PDrec, — The least recently used PDrec Object.

Channel_Queue type Io_program, — The executable queue of i/o programs for a disk or drum.

Experiment_active boolean, — Tells if metering experiment is in progress.

Trace_queue set Trace_datum; — The total of all trace data accumulated by the experiment.

Undocumented Routines Referenced in this Program

page_wait (literal) — Suspends the calling process until a call to notify is made with the identical literal. page_wait also unlocks the page_table lock on the traffic control data bases are locked.

notify(literal) — Causes any process which called page_wait with the identical literal to be resumed.

clear_associative_memory — Causes all processors to clear their associative memories. This routine does not return until all processors have indicated that they have done so. Used to force access turnoffs and Modified bit turnoffs to take effect.

allocate_disk_record() — Returns an unallocated Devadd Object. Marks it as allocated.

relinquish_disk_space(Phys_Devadd) — Marks a Devadd Object as unallocated to allocate_disk_record.

start_io(Io_program) — Starts a channel executing an i/o program.

thread_to_top(Coreadd) — Changes core used list and value of CoreTop such that Coreadd is moved to the top of the Core used list (least recently used). CoreTop now = Coreadd.

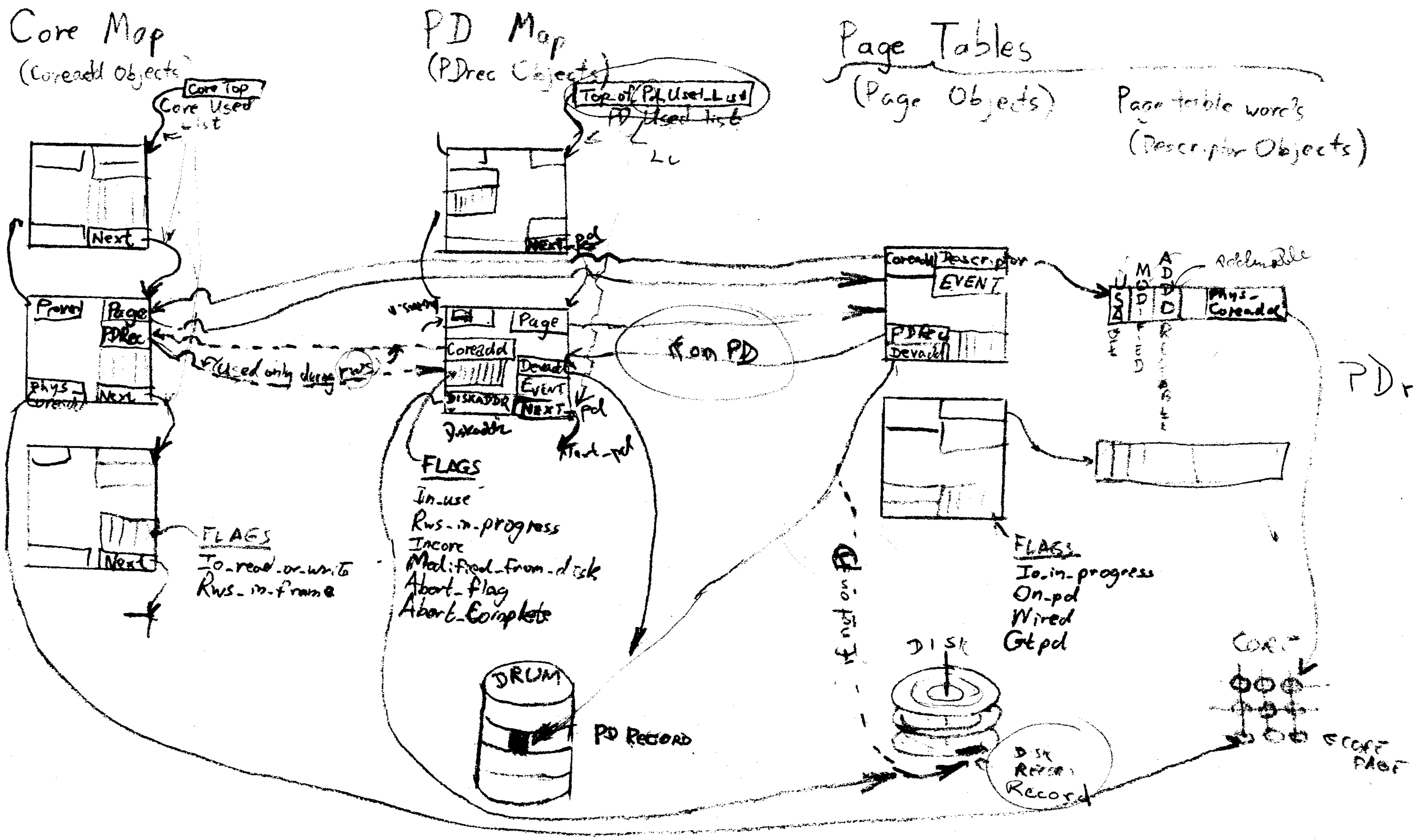
thread_to_bottom(Coreadd) — Changes core used list and value of CoreTop such that Coreadd is moved to the bottom of the core used list(most recently used). Next(Coreadd) now = CoreTop.

pd_thread_to_top(PDrec) — Changes pd used list and value of Top_of_pd_used_list so that PDrec is moved to the top (least recently used). PDrec now = Top_of_pd_used_list.

pd_thread_to_bottom(PDrec) — Changes pd used list and value of Top_of_pd_used_list so that PDrec is moved to the bottom (most recently used). Next (PDrec) now = "null";

pd_thread_out (PDrec) — Removes a pd record from the pd used list.

select_io_routine(literal,literal) — Returns a callable procedure as a value, selecting the entry of the i/o routine specified by the first argument performing the function specified by the second.



THE PAGE CONTROL OBJECTS FOR A SINGLE PAGE

```
page_fault:begin masked;
```

This procedure is transferred to when the hardware determines that a feference has been made through a Descriptor whose Addressable bit is false. Return from this procedure causes a second attempt to make that reference. This procedure is entered in such a way that all external (i/o, etc.) interruptions are disabled when it is entered. They are reenabled when it is exited. This is because such interruptions might try to lock the page-table lock.

```
    local Page;
```

```
    set-lock(Page_table_lock);
    Page := (the faulted page);
```

Prohibit access to page control by other faulting processors

Determine from processor state at fault time which page was faulted on.

```
+   do while (Number_of_free_pd_records < 10);
        call  get_free_pd_record;
    end;
```

Housekeep the paging device - try to have some free pd records for the find_core calls which will surely follow

```
    if      Addressable(Descriptor(Page))
     then unlock (Page_table_lock);
```

It is possible that we took a page fault while the page table was locked, and the process holding the lock brought the page in. Exit if this is true (Addressable is true).

```
      else if Io_in_progress(Page)

      then call page_wait(Event (Page));
```

It is possible that we took a page fault on a page which some other process has started bringing in. Wait for it.

The system routine page_wait causes the suspension of the calling process until some process calls the routine notify with the identical Event with which page_wait was called. Page_wait also unlocks the page-table lock once he has locked his data bases.

```
      else if On_pd(Page) and Rws_in_progress(PDrec(Page))
          then do;
+           call rws_abort(Page);
```

If this page is on the paging device, a read-write sequence (rws) may be in progress for it. It must be aborted.

Abort the rws, or possibly clean up (+) an already complete abort. Unless we are cleaning up, we will have to wait for it.

```
            if Rws_in_progress(PDrec(Page))
             then  call page_wait(Event(PDrec(Page)));
```

If we have cleaned up, we are finished. If we have started an abort or noticed one in progress, we must wait for it.

Wait for rws_done (+) to complete the abort. When we get notified, we will take another page fault and then we will clean up the abort.

```
          end; else unlock (Page_table_lock);
```

We cleaned it up. We can proceed.

```
        else do;
+         call read_page(Page);
          if Io_in_progress(Page)  then
             call page_wait (Event (Page));
         else unlock(Page_table_lock);

        end;
```

Normal case -- we must bring Page in.

Start read-in of Page. If page is empty (zeros), may be done.

If real i/o was started, we must wait for Page. (see post_page

Wait for post_page (+) to post Page.

Page was all zeros. Can use it now.

```
    return;
end page_fault;
```

Restore machine state at time of fault. Retry faulting reference.

```
read_page:procedure(Page);                            This procedure is responsible for causing a faulted page
    local Coreadd;                                  to appear in core. If i/o is necessary, it is initiated. If
                                                    not, a page of zeros is created.
    if On_pd (Page)                                   Although this check is made in page_fault(+), it is conceptually
       then if Rws_in_progress(PDrec(Page))         important that it be made here, for read_page may be called by
          then return;                              other system functions. If an rws is in progress on Page,
                                                    we cannot read it, and our caller must either give up or abort
                                                    the rws.
+   Coreadd:= find_core();                            Allocate a page of core for the Page.
    Page(Coreadd) :=Page;                             Indicate that this page belongs to the Coreframe.
    if Devadd(Page) = "null" then do;                   A null devadd indicates a page defined to contain zeros.
        Phys_Coreadd(Coreadd)→ (1024 words):=00000...;  write_page (+) maintains this discipline. Zero such
                                                        a page if necessary.
        Coreadd(Page) :=Coreadd;                      Indicate that this frame belongs to Page.
+       call make_accessible(Page);                   Reset fault bit in the Descriptor, allowing processors to
                                                      reference Page.

    end;
    else do;                                          Page has non-null devadd, must be read in.
        Io_in_progress(Page) := true;                 Indicate that Page has i/o in progress. It is important
        Io_read_or_write(Coreadd) := "write";       that this be done before the i/o is initiated, so that the
                                                    i/o routine can reset these flags if necessary.

+       call device_read(Devadd(Page),Phys_Coreadd(Coreadd));     Start the read. A fast device may finish, and in
                                                    so doing, will call post_page(+), resetting Io_in_progress.
        if On_pd(Page) then Incore(PDrec(Page)) := true;     Let pd management know that this pdrec has a page
                                                    in core (recently used).
    end;
end read_page;
```

find_core:function();	This function implements the Multics core page replacement algorithm. It is called to housekeep core and return one free Coreadd Object. Its basic data base is the core used list, which is the ordering of Coreadd Objects defined by the sequence of Next components of Coreadd Objects. The variable CoreTop has as a value the least recently used Coreadd Next(CoreTop), Next(Next(Coretop)), and so on, are Coreadd Objects having seen increasingly recent use. The core used list is circular, so Next(most recently used Coreadd) = CoreTop
local Frame type Coreadd; local Page; local Io_skip_counter arithmetic, Loop_counter arithmetic;	The algorithm is due to Corbató ( ).
Io_skip_counter :=0;	Initialize check for excessiveoutstanding i/o.
do Loop_counter :=0 repeat Loop_counter + 1 while (Loop_counter < 131072);	We search the used list for aslong as necessary. We will always return a free page. The default is to crash Multics.
do while(Writes_outstanding > 30); call post_any_io;	If this routine has queued many writes, see if some have completed since we started looping (remeber interrupts are masked).
Io_skip_counter :=0;	Reinitialize i/o check. We have just done what this check could ask us to do.
end;	
if Page(CoreTop) = "null" then do;	We check the supposedly least recently used page to see if it is entirely unallocated (could happen by use for an rws or page deletion.) If so, we can take it.
Frame := CoreTop; CoreTop := Next(CoreTop);	Mark this page as the most recently used, and the next least recently used the least. Notice that this common operation is trivial only because of the circularity of the core used list.
return (Frame);	Return this core frame as useable.
end;	
if Io_in_progress(Page(CoreTop)) then Io_skip_counter := Io_skip_counter + 1;	If, in the next line, we are going to skip this frame because there is i/o going on there, other than an rws, meter the times we have done so.
if Io_in_progress(Page(CoreTop)) or Rws_in_frame(CoreTop) then	See if we must skip this frame due to i/o in progress.
if Io_skip_counter > 100 then do; call post_any_io;	If we have skipped a large number of pages due to i/o in them, see if any i/o has since completed.
Io_skip_counter := 0;	Reset this high-water mark.
end;	Repeat the loop, trying this last io-skipped page (CoreTop) again.
else CoreTop:= Next(CoreTop);	Skip over this page frame, consider Next to be LRU.
else if Wired(Page(CoreTop)) then CoreTop:=Next(CoreTop);	Skip over pages marked as never claimable.
else if Usage(Descriptor(Page(CoreTop))) then do;	See if page has in fact been recently used.
Usage(Descriptor(Page(CoreTop))):=false; CoreTop := Next(CoreTop); end;	If so, reinitialize this check for next time, and skip this page, making it most recently used. This Usage bit is set true by the hardware when the Descriptor is used

(find_core, page 2)

```
        else do;
```

At this stage, the page frame at the top of the core used list has not been recently used. It is a prime target for replacement. We will see if it needs to be written out (try_to_write_page will determine this.) If no i/o is in progress when try_to_write_page returns, we can claim the page.

```
            Page:= Page(CoreTop);
            Frame := CoreTop;
            CoreTop := Next(CoreTop);
```

Make the page under consideration the most recently used. If we ultimately claim it, this was the right move. If we do not, it will either be due to recent use, or the frame will have Io_in_progress true, and this was still the right move.

```
+           call try_to_write_page (Page);
            if not Io_in_progress(Page) then do;
```

See if page needs writing out. Initiate such i/o if so.

If try_to_write_page succeed in totally writing out the page (very fast paging device), this condition holds. Otherwise, move on with the do to the next page in the used list.

```
+               call make_nonaccessible(Page);
```

Turn off access to Page. We do not exit this call until all processors have verified that they have fludhed Descriptor (Page) ffom their associative memories.

```
                if not Modified(Descriptor(Page)) then do;
```

If Page has still not been modified, i.e., in the window between try_to_write_page's check and the time access was turned off, page is now free.

```
    not_mod:
```

page_is_zero(+) branches here when a zero page is found, to avoid the associative memory clear performed in the above call.

```
                  if On_pd (Page) then do;
                     Incore(PDrec(Page)) :=false;
                 /*  call PD_thread_to_bottom(PDrec(Page));*/
```

If page is on the pd, must update Incore status in PDrec.

This line does not in fact exist in the actual code, but it should. It would assert that a pd record of a page being ousted from core has seen recent use. This known bug causes pure pages (see write_page (+)) to be prematurely ousted from the pd.

```
                  end;
+                 else call meter_disk(Devadd(Page),"virtual");
                  return (Frame);
                end;
+           else call  make_accessible(Page);
```

Indicate a virtual write from core to the experiment. This frame is now free, and access is off.

Page was in fact modified in the window. Restore aceess, accept page as recently used, and move on to next page.

```
      end;end;
      end;
      call crash_system;
    end find_core;
```

End of do which checks excessive looping.
Too much looping - something very wrong.

```
try_to_write_page:procedure(Page);                       This procedure determines if a page has been
                                                      modified, and thus needs to be written out. It also
                                                      checks for the case where a page should be written to
                                                      the paging device due to its recency of use.
   if On_pd(Page)                                        If Page already has a copy on the paging device,
      then   if Modified(Descriptor(Page))            the decision to write is the same as whether or not
+            then call write_page(Page, "modified","pd_ok");  Page has been modified.
             else;                                      Page not modified, already on pd, need not write.
      else                                              Page is not on the paging device.  If it can go there,
                                                      we will put it there.
             if  Gtpd(Page) or In_use(Top_of_PD_used_list)   If Page is forbidden to go on pd, or there are no
                                                      pd records available, we cannot put it on pd.
                then if Modified(Descriptor(Page))
+                    then call write_page(Page,"modified","no_pd");       Notice that Page  cannot go to pd.
                     else;                              Page is not modified, cannot go to pd, so need not
                                                      write.
                else if Modified(Descriptor(Page))      Page must be written to pd due to recency of use,
                                                      regardless of whether or not it has been modified.
+                    then call write_page(Page,"modified","put_on_pd");  Indicate that Page must go on pd.
+                    else call write_page(Page,"not_mod","put_on_pd");
return;
end try_to_write_page;
```

write_page: procedure(Page,Modflag,PDflag);	This procedure, which is called when it is determined that a page must be written out, does so. It is defined in Multics that a page of zeros is never written out, but specially flagged. We make that check here.
declare Modflag literal,PDflag literal;	Page is the Page of interest, Modflag is either "modified" or "not_mod", telling us whether or not to turn on Modified_from_disk (PDrec(Page)), and PDflag tells us whether a new pd record should be allocated for Page.
+ if page_is_zero (Page) then return;	If page contained zeros, no write need be done, and we return.
Modified (Descriptor(Page)) := false;	* We have noted modification to page. Any modification after this point (actually the next statement) will be caught by find_core once the i/o that we will start has finished.
call clear_associative_memory;	**Let other processors who may modify this page note that we have turned off "Modified".
Io_in_progress(Page):= true;	Assert that core frame used by Page is not claimable.
Io_read_or_write(Coreadd(Page)):="write";	Let post_page(+) know what kind of i/o took place.
if Devadd(Page) :="null" then Devadd(Page) := allocate_disk_record();	If page was previously zeros, i.e., has no secondary storage home, give it one.
if PDflag = "put_on_pd" then call allocate_pd(Page); +	If page must go to pd,due to recency of use, allocate a pd record. This will save the current Devadd(Page) in Diskaddr(PDrec(Page));Devadd,On_pd, and PDrec of Page will change.
if On_pd (Page) then do;	If Page is, at this stage, on the pd, update the status of its PDrec.
if Modflag := "modified" then Modified_from_disk(PDrec(Page)) := true;	If Page has been modified in core, the pd copy that we are about to start writing is surely different than the disk copy.
call pd_thread_to_bottom(PDrec(Page));	Indicate that this PDrec has seen very recent use.
end;	
+ call device_write(Devadd(Page),Phys_Coreadd(Coreadd(Page)));	Start the write.
return;	
end write_page;	

*We leave access on to pages while they are being written, as there is no reason why not to. When the write completes, Page's frame will be put at the most claimable position of the core used list. If Page has not been used since the "call clear_associative_memory" above, find_core will note the Usage and Modified bits off, and claim the frame immediately, turning access off. Any modification between these two times will cause the Modified bit to be turned on, invalidating the copy which we are writing now.

**A Multics processor retains in an associative memory copies of recently used Page Descriptors. If the processor modifies a page, it checks that the Modified bit is set in the associative memory copy of the Descriptor. If not, the core copy will be modified. Thus, if we turn off the Modified bit in the Descriptor in core, we must purge all of the processor associative memories so that the processors will turn it on if they modify the Page.

```
allocate_pd:procedure (Page);

    local PDrec;

    PDrec := Top_of_pd_used_list;

    Page(PDrec) := Page;
    On_pd(Page):= true;
    PDrec(Page) := PDrec;
    Number_of_free_pd_records :=Number_of_free_pd_records - 1;
    In_use(PDrec) := true;
    Incore(PDrec) := true;

    Diskaddr(PDrec) := Devadd(Page);
    Devadd(Page) := Devadd(PDrec);
    Modified_from_disk(PDrec) := false;

    return ;
end allocate_pd;
```

This function is called to assign a pd record to a page. It can only be called when there is a pd record available for allocation. The pd record which is the current top of the pd used list will be used.

We will use the top record of the pd used list, which is guaranteed to be free (see "try_to_write_page" (+)).

Page is now on pd, for all who wish to know.

And this is where it is.

Decrement count of free pd records.

This record is in use.

The page for this pd record is now in core. This bit is used to maintain the LRU ordering of the pd used list.

The previous disk home of page is now maintained in the PDrec description.

Initialize this flag to say that disk and pd copies are identical. write_page will (+) keep it up to date.

`page_is_zero:function(Page);`	This procedure is called to determine whether or not a page frame contains all zeros. If it does, any disk or paging device records allocated to this page are relinquished.
`if Phys_Coreadd(Coreadd(Page)) →(1024 words) = 000000...`	Check if page is zeros.
`then do;`	
`call make_nonaccessible(Page);`	Turn off the addressability of the page. We will check again if page is all zeros since the page has not been addressable. We do this instead of simply turning off access and checking because [illegible] the vast majority of pages checked here for zeros are not zero, and turning off the access would cause another processor attemting to referenee this page to fault, and loop on the page table lock, which we have locked.
`if Phys_Coreadd(Coreadd(Page)) →(1024 words) = 0000000....`	See if it is still zero.
`then do;`	It is zero. Relinquish secondary storage.
`Modified(Descriptor(Page)) := false;`	Indicate that we are aware of page having been modified (to zeros). We need not clear the associative memories, because, you will recall, access is off to Page
`if On_pd (Page) then do;`	Relinquish the pd record, if there is one.
`Devadd(Page) := Diskaddr(PDrec(Page));`	Prepare to later free the disk record (see below)
`In_use(PDrec(Page)) :=false;`	The pd record is not now in use.
`call pd_thread_to_top(PDrec(Page));`	This pd record is suitable for immediate claiming by allocate_pd (+).
`On_pd(Page) := false;`	
`Number_of_free_pd_records := Number_of_free_pd_records + 1;`	
`end;`	
`call meter_disk(Devadd(Page),"delete");`	Inform the experiment of a page deletion. (Actually, this call is in "relinquish_disk_space", and any call to the latter calls "meter_disk").
`call relinquish_disk_space(Devadd(Page));`	Free the disk space allocated to Page.
`Devadd(Page) := "null";`	Indicate that page is zeros; read_page (+) interprets this null Devadd as an indication that a page of zeros is expected.
`if (we were called from the page fault sequence as opposed to post-purge or other activity) then non-local-go-to not_mod in find_core;`	Page can be considered to have been written. This non-local goto saves find_core the effort of turning off access which is already off and annoying the entire system with yet another global associative memory clear.
`return (true);`	Return the fact that Page was zero.
`end;`	
`else call make_accessible (Page);`	We lost. Page was modified since first check. Restore access.
`end;`	
`return (false);`	Return the fact that page is not zero.
`end page_is_zero;`	

```
get_free_pd_record:procedure;                              This procedure is called to increase the number of
                                                          free pd records.   It maintains the LRU discipline on
                                                          the pd used list.  It returns when it has freed one, or
                                                          has started a large number of read-write sequences. Since
                                                          starting a read-write sequence indicates  that one more pd
   local Rws_ctr        arithmetic;                       record will be available,  the free count is incremented
   local PDrec;                                           when this is done. (see "start_rws").
   Rws_counter := 0;                                         Initialize the count of rws's  that we initiate.
   do forever;                                               Repeat this loop as long as necessary.
      do  PDrec := Top_of_pd_used_list                       Loop over the pd used list, starting at the most
         repeat  Next (PDrec)                             claimable end.
         while PDrec  ≠"null" ;                              Stop at bottom of list.
         if In use (PDrec)                                   We can only free those records not already free.
            then if Incore(PDrec)                            If the page on this pd record is in core, it is
                 then call pd_thread_to_bottom(PDrec); surely among the most recently used  on the pd.
                 else if Modified_from_disk (PDrec)            By the fact that we have got to this pd record
                      then do;                            starting at the top, it should be ousted.
                           call start_rws(PDrec);              If pd copy is different than  disk copy, we must rws.
                                                                start_rws will inform the experiment, and increase
                                                          Number_of_free_pd_records.
                           Rws_ctr:=Rws_ctr + 1;               Increment this  heuristic. If we have started a large
                           if Rws_ctr> 30                 (30) number of rws's, surely we have made enough free pd
                              then return;                 records that we can return  and not be called again.
                                                           (see "page_fault").
                      end;
                      else do;                                  Disk copy same as pd copy. Can oust page with
                                                           little ceremony.
                           Number_of_free_pd_records :=Number_of_free_pd_records + 1; Increment counter.
                           Devadd (Page(PDrec)):=Diskaddr(PDrec);  Note that Page(PDrec) cannot be in
                                                           core because of a decision above.   Make sure page gets
                                                           fetched from disk.
                           On_pd(Page(PDrec)):= false;
                           In_use(PDrec):=false;                 This pd record is no longer in use.
                           call meter_disk(Devadd(Page(PDrec)),"pd_virtual"); Inform the experiment of
                           return;                          traffic to the disk.
                      end;
      end;
      call post_any_io;                                        We have hit bottom of list.  See if any rws's have
                                                            come back since we started looking, and ...
   end;                                                        Try again at the top of the pd used list.
end get_free_pd_record;
```

```
  post_page:procedure(Coreadd);                     This procedure  is responsible  for changing the state
                                                  of page control data bases when the completion of an i/o
      local Page;                                 operation is observed. It is invoked from individual device
                                                  control routines.
      Page :=Page (Coreadd);                        Identify the page for which i/o just completed.
      if Rws_in_frame(Coreadd)                      See if an rws just passed an important point.
+         then call  rws_done(Coreadd);             Handle it if so.
          else do;                                  Otherwise, this was a normal page read or write.
              Io_in_progress(Page) := false;        Turn off i/o flag.
              if Io_read_or_write(Coreadd) = "read" See if a read just oompleted.
                 then do;

                     Coreadd(Page) := Coreadd;      Page will be accessible in this frame.
+                    call make_accessible (Page);   Insert physical core address, turn on access.
                 end;

                 else do;                           Handle a write which completed.
                     Writes_outstanding := Writes_outstanding - 1;    Maintain   heuristic for find_core (+).
                     call thread_to_top (Coreadd);  Make this core frame to be the most likely candidate
                                                  for claiming. The usual reason that a write was started is
                                                  that it was a good candidate for claiming. If Page has
                                                  been used (this also covers modified) since  the  Usage bit
                                                  was turned off, find_core will not claim this page now.
                                                  Otherwise, it will be the very next page claimed.
                 end;

              call notify (Event(Page));            Cause any process waiting for the completion of this i/o
          end;                                    operation to resume.

      return;
  end post_page;
```

```
start_rws:procedure(PDrec);                          This procedure initiates the moving of a modified
                                                   page from the paging device to the disk.
      local Coreadd;

+     Coreadd := find_core();                        Get a page of core for the rws buffer.
      Rws_in_frame(Coreadd) := true;                 Mark this core frame as unclaimable. Flag also lets
                                                   post_page know what to do.
      Rws_in_progress(PDrec) := true;                Mark this pd record as having an rws in progress.
      Io_read_or_write(Coreadd) := "read";           Indicate direction of i/o for post_page.
      Coreadd(PDrec) := Coreadd;                     Set up this relation, which is only used for rws's, so that
                                                   rws_abort(+) can use the Coreadd.
      PDrec(Coreadd):=PDrec;                         Set up this relation, which is only used for rws's, so that
                                                   rws_done (+) can find the pdrec.

+     call device_read(Devadd (PDrec) ,Phys_Coreadd(Coreadd));   Start the page read.
      call pd_thread_out (PDrec);                    Thread the pd record out of the list, so it can't be claimd
+     call meter_disk(Diskaddr(PDrec),"write");      Inform the experiment of the rws.
      Rws_active_count := Rws_active_count + 1;      Maintain a heuristic used below.
      Number_of_free_pd_records := Number_of_free_pd_records + 1;   Indicate another pd record is being freed.
      Modified_from_disk(PDrec) := false;            Indicate that this pd record will be the same as the disk
                                                   copy. rws_abort (+) can change this.

      do while (Rws_active_count > 30);              If there is a large amount of outstanding rws activity,
                                                   wait for some of it to complete.
+           call post_any_io;

      end;

end start_rws;
```

```
rws_abort:procedure (Page);                          This procedure  is invoked when a page fault is taken
    local PDrec;                                  on a page which has an rws in progress.
                                                     There are three such cases. 1)No abort has been initiated
                                                  yet. We initiate one, and wait for notification from rws_done(+).
                                                  2) Another process has initiated one. We wait for it. 3) We have
                                                  been notified by rws_done, and must "clean up" the abort.
    PDrec:= PDrec(Page);                             The pd record is of intense interest here.
    if Abort_flag(PDrec)                             if so, either case 2 or 3 above.
      then if   Abort_complete(PDrec)                If this is so, case 3.  We clean up, and the rws and
          then do;                                rws abort are over.
               Abort_flag(PDrec),
               Abort_complete(PDrec):=false;         No more rws abort.
               Coreadd(Page) :=Coreadd(PDrec);       Use rws buffer as a home for the page.
+              call make_accessible(Page);           Insert physical address into descripter, turn off
                                                  fault bit.
+              call meter_disk(Diskaddr(PDrec),"read");  This rws abort represents negation of movement to disk,
                                                  hence, it is reported to the experiment as motion from disk.
               call thread_to_bottom (Coreadd(Page));   This page is now most recently used in core, ...
               call pd_thread_to_bottom (PDrec);        And most recently used on the pd.
               Incore(PDrec),In_use(PDrec) := true;     Update status of pd record.
               Rws_active_count := Rws_active_count - 1;  Maintain this heuristic, which is used by start_rws(+).
               Rws_in_frame(Coreadd(PDrec)) := false;   Update the status of the core frame.
               Rws_in_progress(PDrec):= false;
               Number_of_free_pd_records := Number_of_free_pd_records - 1;   Reverse decision made by start_rws (+).
          end;                                          Return to page_fault with Rws_in_progress off.
          else;                                         Case 2. Abort already started. Wait for it.
       else  Abort_flag (PDrec) := true;                Case 1. Abort the rws.
     return;
end rws_abort;
```

```
rws_done:procedure(Coreadd);

    local PDrec;
    PDrec :=  PDrec(Coreadd);

    if Abort_flag (PDrec)
      then do;
        if Io_read_or_write(Coreadd) =  "read"
          then Modified_from_disk(PDrec) := true;
          else Writes_outstanding := Writes_outstanding - 1;
        Abort_Complete(PDrec) := true;


        call notify(Event(PDrec));


     end;
     else do;
        if Io_read_or_write(Coreadd) = "read"
          then do;
            Io_read_or_write(Coreadd) := "write";
            call device_write(Diskaddr(PDrec),Phys_Coreadd(Coreadd));
          end;
          else do;

            Rws_active_count := Rws_active_count - 1;
            Writes_outstanding := Writes_outstanding - 1;
            Rws_in_frame (Coreadd) := false;
            Rws_in_progress(PDrec) := false;
            In_use(PDrec) := false;
            call pd_thread_to_top(PDrec);
            Devadd(Page(PDrec)) := Diskaddr(PDrec);
            On_pd(Page(PDrec)) := false;
            Page(Coreadd) := "null";
            call thread_to_top(Coreadd);
          end;
     end;
     return;
end rws_done;
```

This procedure handles the completion of i/o which is on behalf of read-write sequences. Aborts are noticed here, as well.

Identify the P^{D}rec involved from the field specifically reserved for this line.

If an abort was requested, abort the rws.

If a read was aborted, we will not write, and must re-indicate that record differs from disk.

Otherwise, maintain write count.

Indicate that we have aborted the rws. We cannot make the page addressible because (point of fact!) we have no way to locate Page(PDrec) in the actual implementation. This is due to not having enough space to save the required pointers.

Thus, we cause all processes who faulted on this page during the rws to resume. They will all re-take the page faults which made them first see the rws, and the first one to lock the page table lock will make the Page addressable. The others will find that there is no more page fault (see page_fault) and simply return.

Normal rws i/o completion.

If a read finished, start the write half o f the rws.

Let the next pass through here know the story.

Start the write.

The write, and thus the whole rws, finished successfully, i.e., without an abort.

Decrement heuristic used by start_rws (+);

Maintain find_core (+) heuristic.

No more rws here,....

Or here.

This pd rec is now free.

Thus, move pd record to claimable position.

The page that was on this pd record is now only on disk.

That page is no longer on the paging device.

The core block used as an rws buffer is now free.

This page frame should be the next one claimed.

Small Auxiliary Routines

Although some of these short routines might better be expressed in-line, they are conceptually modules in their own right, and may be called from other points in the system.

```
device_read:procedure(    Devadd,Phys_Coreadd);        Called to initiate a read - dispatches call to correct
                                                         i/o routine;
    declare Phys_Coreadd arithmetic;
    if Device (Devadd) ≠ "drum"                        Meter all disk reads for the experiment
+       then call meter_disk (Devadd, "read");
    call select_io_routine_entry
        (Device(Devadd),"read")(Phys_devadd(Devadd),Phys_Coreadd);   Call the righ io routine's read entry.
end device read;

device_write:procedure(Devadd,Phys_Coreadd);          Called to initiate a write - dispatches call.
     Writes_outstanding := Writes_outstanding + 1;    Maintain find_core heuristic;
     call select_io_routine_entry
        (Device(Devadd),"write")(Phys_devadd(Devadd),Phys_Coreadd);
end device_write;

make_accessible:procedure(Page);                       Called to allow access to a page;
     Phys_Coreadd(Descriptor(Page)):=Phys_Coreadd(Coreadd(Page));   Place physical core address  in descriptor.
     Addressable(Descriptor(Page)) := true;            Make page addressable.
end make_accessible;

make_nonaccessible:procedure(Page);                    Used to deny system access to a page.
     Addressable(Descriptor(Page)) := false;           Turn off access. Future references will cause page faults.
     call clear_associative_memory;                    Flush descriptor from processor associative memories.
                                                       Not until all processors have indicated that they have done
end make_nonaccessible;                                this can page be considered inaccessible.

meter_disk:procedure(Devadd,Type);                     Procedure to accumulate data for thesis experiment.
     declare Type literal;
     if Experiment_active then                         Buffer for Trace_queue is not wired unless this  is set.
        enqueue(construct Trace_datum
              (Devadd:Devadd,Type:Type),Trace_queue));    Enqueue the trace datum in the buffer.
end;

post_any_io:procedure ;                                This routine is called in any situation where page control
     declare Device literal;                             discovers some i/o bottleneck. It polls i/o routines
     do  Device:= range(i/o devices as  a set);          for completed operations. They will call post_page(+)
        call select_io_routine_entry (Device,"post")();  if they have any.
    end;
end  post_any_io;
```

A Typical Paging i/o Control Routine

This routine is the i/o control routine for the fixed-head disk. There exist routines almost identical routines for the moving-head disk and drum. The routine select_io_routine entry (not given here) is used to select appropriate routines given a device identification.

```
declare Phys_devadd arithmetic, Phys_Coreadd arithmetic;

fixed_head_read:procedure(Phys_devadd,Phys_Coreadd);                    Read entry;
   call fixed_head_start(Phys_devadd,Phys_Coreadd,"read");
end;

fixed_head_write:procedure(Phys_devadd,Phys_Coreadd);                   Write entry;
   call fixed_head_start(Phys_devadd,Phys_Coreadd,"write");
end;

fixed_head_start:procedure(Phys_devadd,Phys_Coreadd,Direction);         Entry to start an i/o operation.
   declare Direction literal;
   call fixed_head_post;                                                See if any operations have completed.
   enqueue (construct Io_program(Phys_Devadd:Phys_devadd,               Construct a channel program and enqueue it.
                                 Phys_Coreadd:Phys_Coreadd,
                                 "null"),Fixed_head_Channel_Queue);
   if (fixed head disk is not busy) then call start_io (Fixed_head_Channel_Queue);  There is now work for fixed-head
                                                                        disk. Start it if itis idle.
end fixed_head_start;

fixed_head_post:procedure;                                              Used to post complete status.
   do Io_status := range (any complete i/o status);                     Look at all new status.
      remove Io_status from (set of complete status);                   Take it out of hardware queue.
      call post_page(Coreadd(Io_status));                               Inform page control. See the declaration of
      remove Io_program(Io_status) from Fixed_head_Channel_Queue;          Io_status.
   end;
end;

fixed_head_interrupt:procedure masked;                                  Entered masked against all i/o interrupts from
                                                                           system interrupt manager.
      set-lock (Page_table_lock);                                       Lock the page tables. Remeber that page_fault
                                                                           masks against interrupts before locking.
      call fixed_head_post;                                             Post the completed i/o.
      if (fixed head disk is not busy)                                      If the disk has stopped doing work,
         then if not void Fixed_head_Channel_Queue                      but there is more useful work, start it.
            then call start_io (Fixed_head_Channel_Queue);
      unlock (Page_table_lock);                                         Unlock.
end fixed_head_interrupt;
```

To those reading this to learn about Multics:

My name for an object	Real name of object
page_fault	masking is in wired_fim, all else page_fault
Number_of_free_pd_records	sst.pd_free
Addressable(Descriptor(Page))	ptw.df
On_pd	devadd.did = sst.pd_id
Rws_in_progress	pdme.rws
Io_in_progress	ptw.os
Event	fabricated entity for clarity
rws_abort	fabricated, part of page_fault
Page	may mean ptw, or cme, depending upon context
Coreframe, Coreadd	cme
Io_read_or_write	cme.io
Incore	pdme.incore
In_use	pdme.used
Page(Coreadd)	cme.ptwp
Writes_outstanding	sst.wtct
Rws_active_count	sst.pd_wtct
Gtpd	aste.gtpd
clear_associative_memory	mmster_pxss_page$cam
post_any_io	device_control$run
Loop_counter	total_steps (stack variable)
Io_skip_counter	count (stack variable)
Usage	ptw.phu
Modified	ptw.phm
make_accessible, make_nonaccessible	fabricated entities, done in line
thread_to_bottom, thread_to_top	done by unthread subroutine, but see note in find_core.
relinquish_disk_space	free_store$deposit
allocate_disk_record	free_store$withdraw
No_pd_flag	no_pd
Modified_from_disk	pdme.mod
pd_thread_to_top, pd_thread_to_bottom, pd_thread_out	rethread [illegible] subroutine in pd_util, and done in line
Rws_ctr	pd_count (stack variable)

start_io	mini_gim, iom_manager
iocr_post_any_entry	"$run" entries
post_page	done_
page_wait	pxss$page_wait
notify	pxss$notify
Abort_flag	pdme.rws_abort
Rws_in_frame	cme.rws
PDrec	usually pdme
Devadd (Page)	cme.devadd
Diskaddr	pdme.devadd
Device Descriptor	did ptw
"The experiment"	the "disk metering" installed in 18.5 for Bernard Greenberg's Thesis
rws_done	fabricated, part of page$done_
try_to_write_page, write_page, page_is_zero, allocate_pd	write_page
select_io_routine	device_control
CoreTop	sst.usedp
Top_of_pd_used_list	sst,pdusedp

Anybody else plays himself.