Memory Management

1. Goal: CPU gains related to scheduling require that many processes be in memory; so, memory must be shared. The selection of which memory management algorithm(s) to use depends especially on the hardware available.
2. How Programs Interact With Memory
   a. compile, assemble, link, load
   b. absolute (physical) addresses
   c. bare machine, resident monitor, fence addresses, and fence registers
3. Relocation
   a. binding program addresses to absolute addresses
   b. transient monitor code
   c. monitor and user stacks grow towards each other
   d. dynamic relocation
   e. swapping: swap device (backing store), swapped processes, effect of swap time on switching contexts
   f. optimizations: swapping part of the job, speeding up the backing store's performance, overlap with process execution
   g. when you can and cannot swap
4. Simple Memory Management Schemes
   a. multiple partitions
   b. bounds registers
   c. base and limit registers
5. Fixed Regions (MFT)
   a. job scheduling
   b. memory allocation
   c. roll in-roll out
6. Variable Regions (MVT)
   a. allocation schemes: best fit, worst fit, first fit, next fit, buddy
   b. job scheduling
   c. internal vs. external fragmentation
   d. compaction
   e. swapping
   f. reducing external fragmentation
7. Paging
   a. page numbers and offsets, page frames
   b. job scheduling effects
   c. page table implementation
8. Optimizations
   a. cache, hit ratio
   b. effective memory access time
   c. sharing, protection, illegal address handling
9. Views of Memory
   a. program *vs*. operating system
   b. address translation
   c. global frame table
10. Segmentation
    a. segments: segment name, offset, segment table
    b. implementation: registers *vs*. table
    c. sharing, protection, fragmentation
11. Combining Segmentation and Paging
    a. segmented paging (segment the page table)
    b. paged segmentation (page the segments)
12. What Is Virtual Memory

   a. executing processes not completely in memory
   b. overlays
   c. demand paging, page faults, pure demand paging
   d. performance issues
   e. page replacement and victims and dirty bits
13. page replacement algorithms
   a. first in first out
   b. optimal
   c. least recently used
      i. stack algorithms, inclusion property, used (reference) bit
   d. clock
   e. second chance
   f. least and most frequently used
   g. not used recently
   h. second-chance cyclic: (1,1)→(0,1); (1,0)→(0,0); (0,1)→(0,0)*; (0,0)→*victim*
14. Ad Hoc Techniques for Improving Performance
   a. frame pool
15. Page Allocation Algorithms
   a. reserve some free frames
   b. use all before replacing
   c. minimum number of pages per process
   d. global *vs*. local allocation
   e. equal, proportional allocation
16. Working Set Policies
   a. thrashing, principle of locality
   b. working set model: working set, window size, working set principle
   c. approximations: WSCLOCK, Working Set Size, Page Fault Frequency
17. Other Considerations for Paging
   a. prepaging
   b. I/O interlock
   c. choosing page size
   d. program restructuring
   e. data structures and paging
   f. arrangement of routines when loading

The Stages of Compilation

Introduction

This describes the basic stages of compilation in a very non-traditional way …

```
From: Mateo.Burtch@Eng.Sun.COM (Room 101)
Subject: YADFH (Yet Another D*** Friday Hack)

Hi.  This isn't exactly a hack, and thus probably doesn't deserve to be called
a "Friday Hack," but SOOOOO many people have been coming up to me in the
shower, saying, Please, tell us the secrets of how the compilation process
works, that I finally decided to share with you the old family recipe for:

THE STAGES OF COMPILATION

SOURCE FILE:  This is the basic "code" that the engineer writes.  The compiler
will take code such as

            dweeble(flab, krimjaw)
            sneet flab *(**smicknat[])(blugnut);
            blook krimjaw;
            {
                  blark snapdaddle liederhosen ;

                  if (fleb <= OAK_TREE)
                        while (trousers(liederhosen))
                              thud;
                  else
                        brick(flab);
            }

and turn it into output that means, roughly, "point the hose away from yourself
while watering."

Obviously, the compiler is a lot smarter than we are.

PREPROCESSOR:  The preprocessor takes all sorts of special directives, like
#ifdefs, and converts them into conditional statements (known as "#ifdefs")
that the compiler uses to prepare the file (or "#ifdef") for processing.  (This
is a technical simplification that is in most, if not all, aspects wrong.)

The preprocessor also strips the source file of comments (not to be confused
with #ifdefs), leaving it a shaken husk of its former self. These comments are
then pieced together by a separate function and used for insulation in Building
12.

COMPILER:  The heart of the whole process.  The compiler takes the language
words, such as "if," "for," and "help!" and turns it into the bits, bytes, and
subroutines that give employment to a whole host of socially challenged people.

OPTIMIZATION:  This stage makes the generated code as efficient as possible.
The optimizer does this by carefully trimming the odd and dangly parts off of
```

numerals like "5" and "9" until all the numbers are either "1"s or "0"s.

ASSEMBLER: The assembler takes "machine code" (low-level instructions done by poorly-paid workers in windowless sweatshops called "assembler lines") and does something with it.

LINKER: Before we're done, we must link together all the various object files with libraries containing macros and canned routines. Don't ask why--it's like salmon swimming upstream to spawn. You just do it.

These libraries perform a special job in the grand pageant known as programming. The pre-written macros and functions they contain allow the programmer to save valuable time when writing a program, while the arcane linker options and obscure environment variables slow the programmer down just as much. This is known as "run-time equilibrium" and is rather similar to horizontal bungee-jumping.

EXECUTABLE: This is the final product, the finished masterpiece, the piece de resistance of the whole process. This file is what the engineer had in mind when he or she started out.

It's called "core."

--Mateo

Static and Dynamic Relocation

Introduction

This shows the basic hardware instruction cycle for a machine that uses static relocation and for one that uses dynamic relocation.

Static Relocation

Static relocation refers to address transformations being done before execution of a program begins. A typical hardware instruction cycle looks like this:

```
loop
      w := M[instr_ctr];            (* fetch instruction *)
      oc := Opcode(w);
      adr := Address(w);
      instr_ctr := instr_ctr + 1;
      case oc of
      1:    reg := reg+M[adr];      (* add *)
      2:    M[adr] := reg;          (* store *)
      3:    instr_ctr := adr;       (* branch *)
      …
      end
end (* loop *)
```

Dynamic Relocation

Dynamic relocation refers to address transformations being done during execution of a program. In what follows, the function *NL_map* (for Name Location map) maps the relocatable (virtual) address va given in the program into the real (physical) storage address *pa*:

$$pa := NL\_map(va)$$

So, a typical hardware instruction cycle looks like this:

```
loop
      w := M[NL_map(instr_ctr)];    (* fetch instruction *)
      oc := Opcode(w);
      adr := Address(w);
      instr_ctr := instr_ctr + 1;
      case oc of
      1:    reg := reg+M[NL_map(adr)];    (* add *)
      2:    M[NL_map(adr)] := reg;        (* store *)
      3:    instr_ctr := NL_map(adr);     (* branch *)
      …
      end
end (* loop *)
```

Paging and Address Translation

Introduction

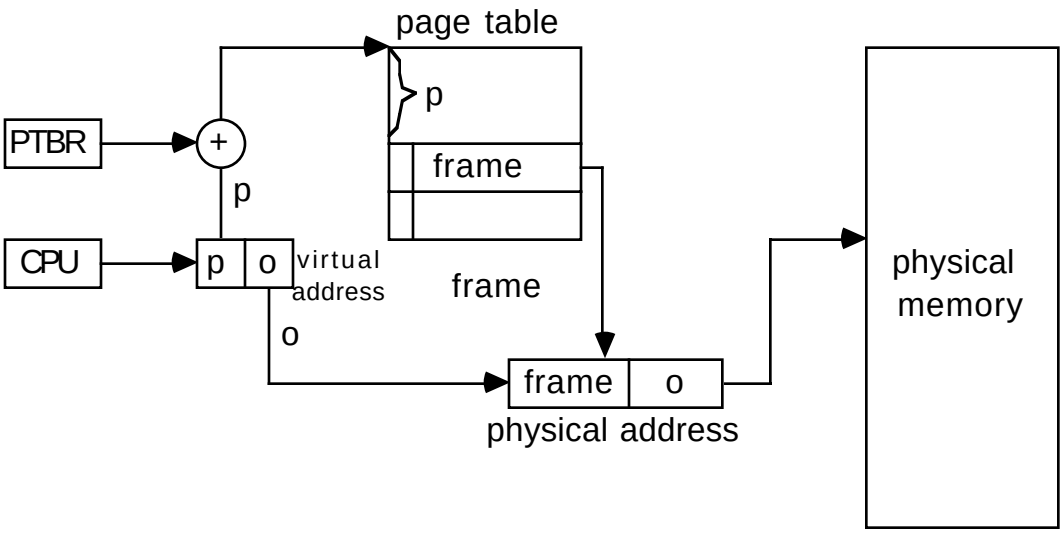
This shows the function used to map a logical address to a physical address for some paging schemes. Throughout this handout, an address in virtual memory is a pair (*logical_page*, *offset*) where *logical_page* is the page number within the logical address space and *offset* the offset into that page. Also, *page_size* is the size of the page (which is a multiple of 2). We will assume the entire program is in memory, so no error handling is given; were this assumption false, the situation where the requested address were not in memory would need to be handled (by generating a page fault and loading the necessary page):

Paging Address Translation by Direct Mapping

This method stores the page table in main memory and the address of this table in the process control block, in a register called the page table base register. Let the page table base register be called *pt_base_register*, and let memory represent the main store of the computer. Then:

```
function NL_map((logical_page, offset)): physical_address;
begin
      NL_map := memory[pt_base_register + logical_page] * page_size + offset;
end (* NL_map *)
```

In pictures, here is what is going on:



Paging Address Translation by Associative Mapping

In this algorithm, *assoc_page_table* represents an associative memory. This function can check a type of memory called "associative memory" (or "cache" or "lookaside memory") which stores both a frame number and a page number. The search is done in parallel, and is much faster than a linear (or binary) search. The function returns the frame number associated with its argument:

```
function NL_map((logical_page, offset)): physical_address;
begin
      NL_map := assoc_page_table(logical_page) * page_size + offset;
end (* NL_map *)
```

Paging Address Translation with Combined Associative and Direct Mapping

This combines the above two methods. The array *page_table* is a small associative store that can hold only a few

page numbers; there is also a page table kept in memory. For this method, we shall assume that if there is no entry for *logical_page* in the associative memory, *assoc_page_table* returns –1. Taking everything else as in the previous two sections:

```
function NL_map((logical_page, offset)): physical_address;
var   frame_number: integer;
begin
      frame_number := assoc_page_table(logical_page);
      if frame_number = -1 then           (* not in associative memory *)
            NL_map := memory[pt_base_register + logical_page]
                                          * page_size + offset;
      else
            NL_map := frame_number * page_size + offset;
end (* NL_map *)
```

This is the most common method, and is used in modern computers with paging.

Segmentation and Address Translation

Introduction

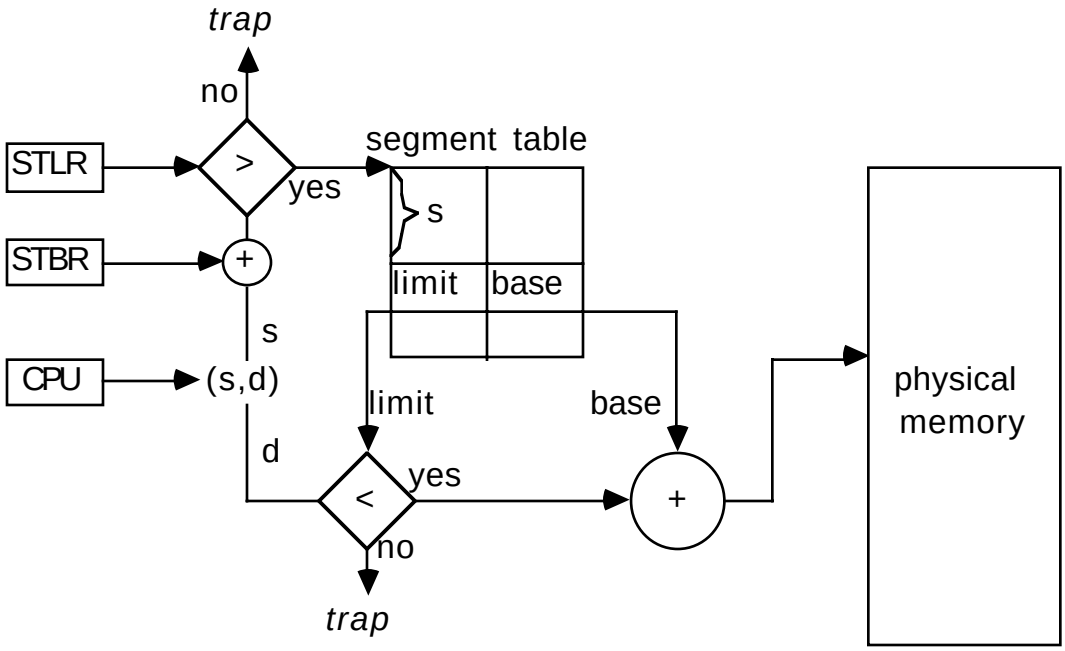
This shows the function used to map a logical address to a physical address for some segmentation schemes. Throughout this handout, an address in virtual memory is a pair (*segment*, *offset*) where *segment* is the segment number within the logical address space and *offset* the offset into that segment. We will assume the entire program is in memory, so no error handling is given; were this assumption false, the situation where the requested address were not in memory would need to be handled (by generating a segment fault and loading the necessary segment):

Segmentation

As with paging address translation with direct mapping, the segment table is stored in memory, and a pointer to its base in a register called the segment table base register. Let the segment table base register be called *st_base_register*, and let memory represent the main store of the computer. Then:

```
function NL_map((segment, offset)): physical_address;
begin
      NL_map := memory[st_base_register + segment] + offset;
end (* NL_map *)
```

In pictures:



Segmentation and Paging Combined

Introduction

This shows the function used to map a logical address to a physical address for schemes combining paging and segmentation. Throughout this handout, *page_size* is the size of the page (which is a multiple of 2), *seg_tbl_base_reg* contains the address of the base of the segment table, and *memory* is the main store of the computer. We will assume the entire program is in memory, so no error handling is given; were this assumption false, the situation where the requested address were not in memory would need to be handled (by generating a fault and loading the appropriate data structure).

Segmented Paging

In this algorithm, the page tables are segmented. The virtual address is represented as a pair (*logical_page*, *offset*), but the *logical_page* consists of a pair (*seg_number*, *seg_offset*) indicating which segment number *seg_number* of the page table the frame number *frame_no* is stored in, and the offset *seg_offset* from the base of that segment table. As usual, an associative memory is first checked; this will be represented by the funcetion *assoc_page_table*, which returns the frame number if that is in the table, and –1 if not:

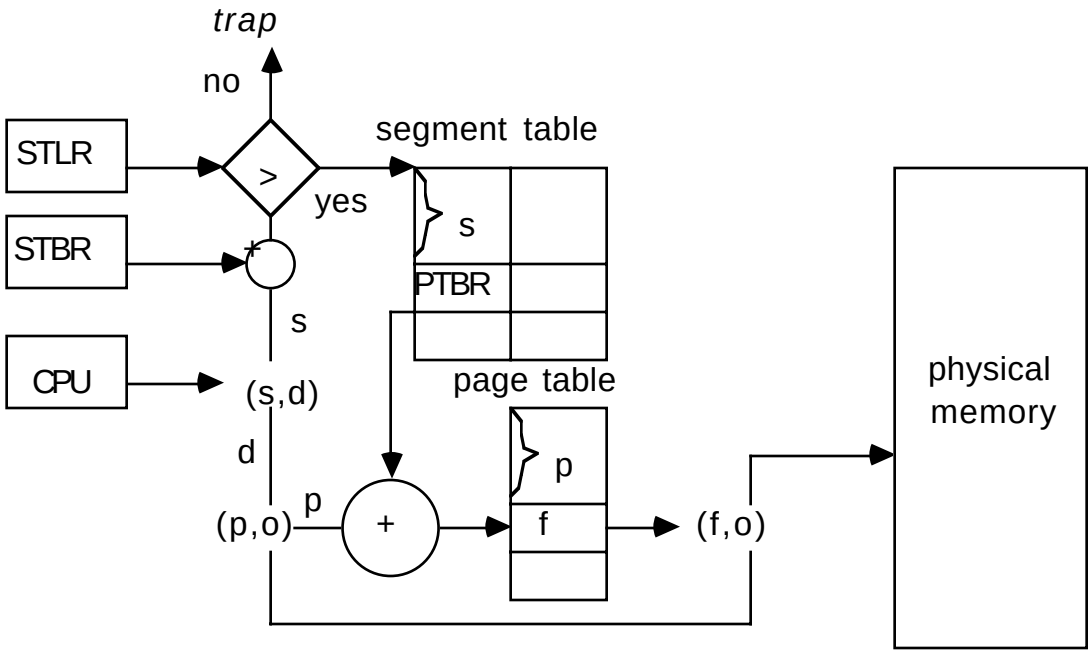
```
function NL_map((logical_page, offset)): physical_address;
var   frame_no: integer;            (* number of frame *)
      pg_tbl_base: integer;         (* addr. of page table segment *)
begin
      frame_no := assoc_page_table(logical_page);
      if frame_no = -1 then begin
            pg_tbl_base := memory[seg_tbl_base_reg + seg_number];
            frame_no := memory[pg_tbl_base + seg_offset];
      end;
      NL_map := frame_no * page_size * offset;
end (* NL_map *)
```

Paged Segmentation

In this algorithm, the segments are paged. The virtual address is represented as a pair (*seg_number*, *offset*), but the offset consists of a pair (*page_number*, *page_offset*), indicating which page number *page_number* of the segment *seg_number* the frame number *frame_no* is stored in, and the offset *page_offset* from the base of that page. As usual, an associative memory is first checked; this will be represented by the funcetion *assoc_page_table*, which returns the frame number if that is in the table, and –1 if not. Note it takes the segment number as an argument as well:

```
function NL_map((seg_number, offset)): physical_address;
var   frame_no: integer;            (* number of frame *)
      pg_tbl_base: integer;         (* addr. of page table segment *)
begin
      frame_no := assoc_page_table(seg_number, page_number);
      if frame_no = -1 then begin
            pg_tbl_base := memory[seg_tbl_base_reg + seg_number];
            frame_no := memory[pg_tbl_base + page_number];
      end;
      NL_map := frame_no * page_size * page_offset;
end (* NL_map *)
```

In pictures:


trap
no
STLR
>
yes
segment table
STBR
+
s
PTBR
s
CPU
(s,d)
page table
d
p
p
(p,o)
+
f
(f,o)
physical
memory

Page Replacement Algorithms

Introduction

This handout shows how the various page replacement algorithms work. We shall call the pages of the program a, b, c, … to distinguish them from the time (1, 2, 3, …).

Fixed Number of Frames

We shall demonstrate these algorithms by running them on the reference string ω = cadbebabcd and assume that, initialy, pages a, b, c, and d occupy frames 0, 1, 2, and 3 respectively. When appropriate, the little arrow → indicates the location of the "pointer" which indicates where the search for the next victim will begin.

First In/First Out (FIFO)

This policy replaces pages in the order of arrival in memory.

time	0	1	2	3	4	5	6	7	8	9	10
ω		c	a	d	b	e	b	a	b	c	d
frame 0	→a	→a	→a	→a	→a	e	e	e	e	→e	d
frame 1	b	b	b	b	b	→b	→b	a	a	a	→a
frame 2	c	c	c	c	c	c	c	→c	b	b	b
frame 3	d	d	d	d	d	d	d	d	→d	c	c
page fault						1		2	3	4	5
page(s) loaded						e		a	b	c	d
page(s) removed						a		b	c	d	e

Optimal (OPT, MIN)

This policy selects for replacement the page that will not be referenced for the longest tile in the future.

time	0	1	2	3	4	5	6	7	8	9	10
ω		c	a	d	b	e	b	a	b	c	d
frame 0	a	a	a	a	a	a	a	a	a	a	d
frame 1	b	b	b	b	b	b	b	b	b	b	b
frame 2	c	c	c	c	c	c	c	c	c	c	c
frame 3	d	d	d	d	d	e	e	e	e	e	e
page fault						1					2
page(s) loaded						e					d
page(s) removed						d					a

Least Recently Used (LRU)

This policy selects for replacement the page that has not been used for the longest period of time.

time	0	1	2	3	4	5	6	7	8	9	10
ω		c	a	d	b	e	b	a	b	c	d
frame 0	a	a	a	a	a	a	a	a	a	a	a
frame 1	b	b	b	b	b	b	b	b	b	b	b
frame 2	c	c	c	c	c	e	e	e	e	e	d
frame 3	d	d	d	d	d	d	d	d	d	c	c
page fault						1				2	3
page(s) loaded					e				c	d	
page(s) removed					c				d	e	
stack (top)		c	a	d	b	e	b	a	b	c	d
		—	c	a	d	b	e	b	a	b	c
		—	—	c	a	d	d	e	e	a	b
stack (bottom)		—	—	—	c	a	a	d	d	e	a

Not-Recently-Used or Not Used Recently (NRU, NUR)

This policy selects for replacement a random page from the following classes (in the order given): not used or modified, not used but modified, used and not modified, used and modified. In the following, assume references 2, 4, and 7 are writes. The two numbers written after each page are the use and modified bits, respectively.)

time	0	1	2	3	4	5	6	7	8	9	10
ω		c	a*	d	b*	e	b	a*	b	c	d
frame 0	a	a/00	a/11	a/11	a/11	a/01	a/01	a/11	a/11	a/01	a/01
frame 1	b	b/00	b/00	b/00	b/11	b/01	b/11	b/11	b/11	b/01	b/01
frame 2	c	c/10	c/10	c/10	c/10	e/10	e/10	e/10	e/10	e/00	d/10
frame 3	d	d/00	d/00	d/10	d/10	d/00	d/00	d/00	d/00	c/10	c/10
page fault						1				2	3
page(s) loaded						e				c	d
page(s) removed						c				d	e

Clock

This policy is similar to LRU and FIFO. Whenever a page is referenced, the use bit is set. When a page must be replaced, the algorithm begins with the page frame pointed to. If the frame's use bit is set, it is cleared and the pointer advanced. If not, the page in that frame is replaced. Here the number after the page is the use bit; we'll assume all pages have been referenced initially.

time	0	1	2	3	4	5	6	7	8	9	10
ω		c	a	d	b	e	b	a	b	c	d
frame 0	a	→a/1	→a/1	→a/1	→a/1	e/1	e/1	e/1	e/1	→e/1	d/1
frame 1	b	b/1	b/1	b/1	b/1	→b/0	→b/1	b/0	b/1	b/1	b/0
frame 2	c	c/1	c/1	c/1	c/1	c/0	c/0	a/1	a/1	a/1	a/0
frame 3	d	d/1	d/1	d/1	d/1	d/0	d/0	→d/0	→d/0	c/1	c/0
page fault						1		2		3	4
page(s) loaded						e		a		c	d
page(s) removed						a		c		d	e

Second-chance Cyclic

This policy merges the clock algorithm and the NRU algorithm. Each page frame has a use and a modified bit. Whenever a page is referenced, the use bit is set; whenever modified, the modify bit is set. When a page must be replaced, the algorithm begins with the page frame pointed to. If the frame's use bit and modify bit are set, the use bit is cleared and the pointer advanced; if the use bit is set but the modify bit is not, the use bit is cleared and the pointer advanced; if the use bit is clear but the modify bit is set, the modify bit is cleared (and the algorithm notes that the page must be copied out before being replaced) and the pointer is advanced; if both the use and modify bits are clear, the page in that frame is replaced. In the following, assume references 2, 4, and 7 are writes. The two numbers written after each page are the use and modified bits, respectively.) Initially, all pages have been used but none are modified.

time	0	1	2	3	4	5	6	7	8	9	10
ω		c	a*	d	b*	e	b	a*	b	c	d
frame 0	a	→a/10	→a/11	→a/11	→a/11	a/00*	a/00*	a/11	a/11	→a/11	a/00*
frame 1	b	b/10	b/10	b/10	b/11	b/00*	b/10*	b/10*	b/10*	b/10*	d/10
frame 2	c	c/10	c/10	c/10	c/10	e/10	e/10	e/10	e/10	e/10	→e/00
frame 3	d	d/10	d/10	d/10	d/10	→d/00	→d/00	→d/00	→d/00	c/10	c/00
page fault						1				2	3
page(s) loaded						e				c	d
page(s) removed						c				d	b

Variable Number of Frames

We shall demonstrate these algorithms by running them on the reference string ω = ccdbcecead.

Working Set (WS)

This policy tries to keep all pages in a process' working set in memory. This table shows the pages consitiuting the working set at each reference. Here, we take the working set to be that set of pages which has been referenced during the last $t = 4$ units. We also assume that a was referenced at time 0, d at time -1, and e at time -2.

time	0	1	2	3	4	5	6	7	8	9	10
ω		c	c	d	b	c	e	c	e	a	d
Page a	a	a	a	a	—	—	—	—	—	a	a
Page b	—	—	—	—	b	b	b	b	—	—	—
Page c	—	c	c	c	c	c	c	c	c	c	c
Page d	d	d	d	d	d	d	d	—	—	—	d
Page e	e	e	—	—	—	—	e	e	e	e	e
page fault		1			2		3			4	5
page(s) loaded		c			b		e			a	d
page(s) removed			e		a			d	b		

Page Fault Frequency (PFF)

This approximation to the working set policy tries to keep page faulting to some prespecified range. If the time between the current and the previous page fault exceeds some critical value t, then all pages not referenced during that interval are removed. This table shows the pages resident at each reference. Here, we take t = 2 units and assume that initially, a, d, and e are resident.

time	0	1	2	3	4	5	6	7	8	9	10
ω		c	c	d	b	c	e	c	e	a	d
Page a	a	a	a	a	—	—	—	—	—	a	a
Page b	—	—	—	—	b	b	b	b	b	—	—
Page c	—	c	c	c	c	c	c	c	c	c	c
Page d	d	d	d	d	d	d	d	d	d	—	d
Page e	e	e	e	e	—	—	e	e	e	e	e
page fault		1			2		3			4	5
page(s) loaded		c			b		e			a	d
page(s) removed		?			a,e					b,d	