Static and Dynamic Relocation

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 += 1;
    switch(oc){
    case 1:  reg += M[adr];      /* add */
    case 2:  M[adr] = reg;       /* store */
    case 3:  instr_ctr = adr;    /* branch */
    ...
    }
}
```

Dynamic Relocation

Dynamic relocation refers to address transformations being done during execution of a program. In what follows, the function *NL_map* (for *N*ame *L*ocation *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 += 1;
    switch(oc){
    case 1:  reg += M[NL_map(adr)];      /* add */
    case 2:  M[NL_map(adr)] = reg;       /* store */
    case 3:  instr_ctr = NL_map(adr);    /* branch */
    ...
    }
}
```

Paging and Address Translation

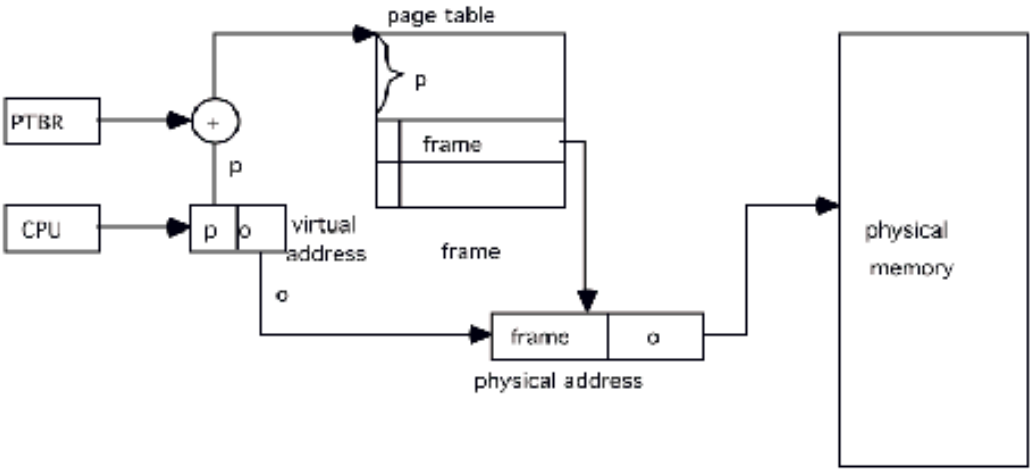
This shows the function used to map a logical address to a physical address for some paging schemes. Throughout this handout, an address in virtual memory is a pair (`logical_page`, `offset`) where `logical_page` is the page number within the logical address space and `offset` the offset into that page. 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:

```
physical_address NL_map((logical_page, offset))
{
        return(memory[pt_base_register + logical_page] * page_size + offset);
}
```

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 "lookaside memory" or "translation lookaside buffer") that 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:

```
physical_address NL_map((logical_page, offset))
{
        return(assoc_page_table(logical_page) * page_size + offset);
}
```

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:

```
physical_address NL_map((logical_page, offset))
{
        int frame_number;

        frame_number = assoc_page_table(logical_page);
        if (frame_number== -1){        /* not in associative memory */
                return(memory[pt_base_register + logical_page] * page_size + offset);
        else
                return(frame_number * page_size + offset);
}
```

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

Segmentation and Address Translation

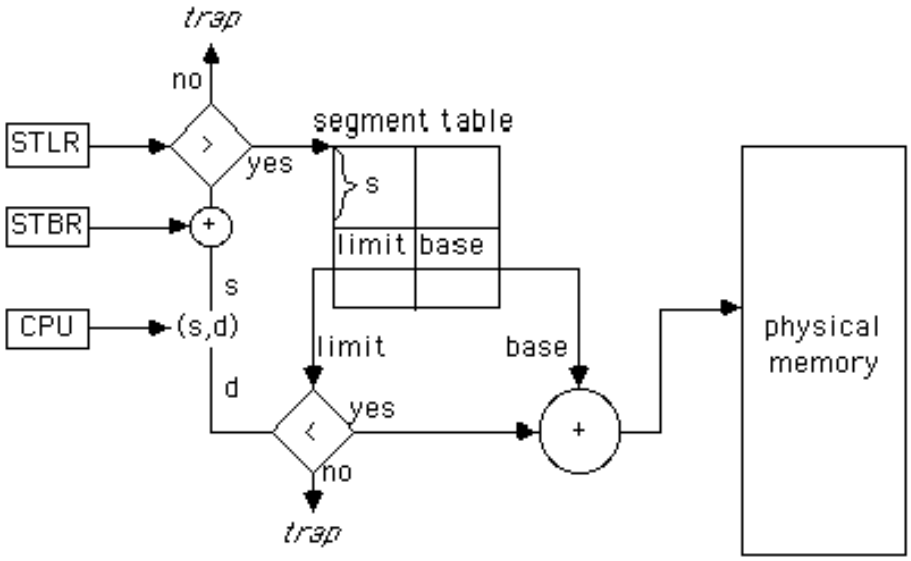
This shows the function used to map a logical address to a physical address for some segmentation schemes. Throughout this handout, an address in virtual memory is a pair (`segment`, `offset`) where `segment` is the segment number within the logical address space and `offset` the offset into that segment. 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:

```
physical_address NL_map((logical_page, offset))
{
        return(memory[st_base_register + segment] + offset);
}
```

In pictures, here is what is going on:



Segmentation and Paging Combined

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 function *`assoc_page_table`*, which returns the frame number if that is in the table, and -1 if not:

```
physical_address NL_map((logical_page, offset))
{
    int frame_no: integer;          (* number of frame *)
    int pg_tbl_base: integer;       (* addr. of page table segment *)

    frame_no = assoc_page_table(logical_page);
    if (frame_no == -1){
        pg_tbl_base = memory[seg_tbl_base_reg + seg_number];
        frame_no = memory[pg_tbl_base + seg_offset];
    }
    return(frame_no * page_size + offset);
}
```

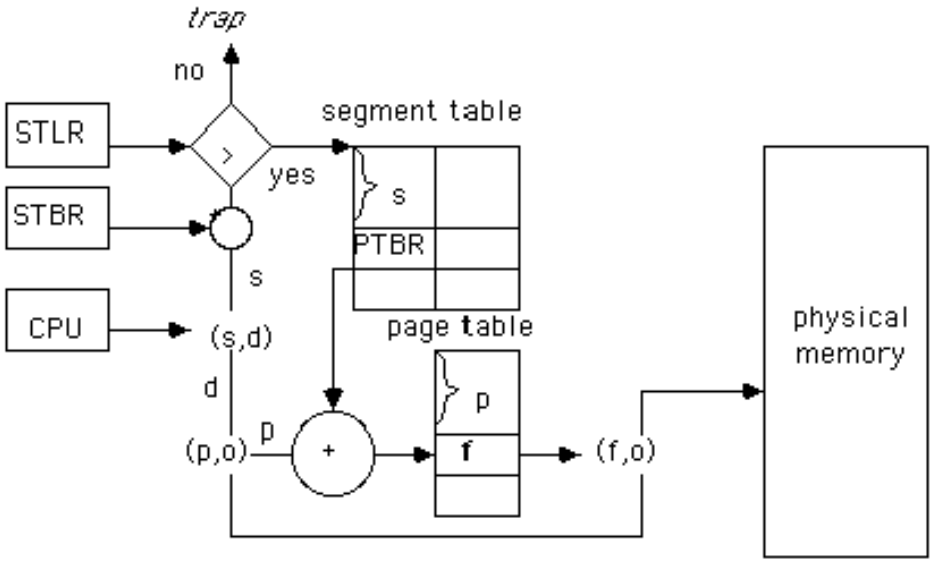
Paged Segmentation

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 function *`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:

```
physical_address NL_map((seg_number, offset))
{
    int frame_no: integer;          (* number of frame *)
    int pg_tbl_base: integer;       (* addr. of page table segment *)

    frame_no = assoc_page_table(seg_number, page_number);
    if (frame_no == -1){
        pg_tbl_base = memory[seg_tbl_base_reg + seg_number];
        frame_no = memory[pg_tbl_base + page_number];
    }
   return(frame_no * page_size + page_offset);
}
```

In pictures, here is what is going on:


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

Page Replacement Algorithms

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 $\omega = cadbebabcd$ and assume that, initialy, pages *a*, *b*, *c*, and *d*occupy frames 0, 1, 2, and 3 respectively. When appropriate, the little arrow $\rightarrow$ indicates the location of the "pointer" that 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	$\rightarrow a$	$\rightarrow a$	$\rightarrow a$	$\rightarrow a$	$\rightarrow a$	*e*	*e*	*e*	*e*	$\rightarrow e$	*d*
frame 1	*b*	*b*	*b*	*b*	*b*	$\rightarrow b$	$\rightarrow b$	*a*	*a*	*a*	$\rightarrow a$
frame 2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	$\rightarrow c$	*b*	*b*	*b*
frame 3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	$\rightarrow d$	*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 time 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 at times 2, 4, and 7 are writes (represented by the bold page references). 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*/00	*a*/00	*a*/11	*a*/11	*a*/11	*a*/01	*a*/01	*a*/11	*a*/11	*a*/01	*a*/01
frame 1	*b*/00	*b*/00	*b*/00	*b*/00	*b*/11	*b*/01	*b*/11	*b*/11	*b*/11	*b*/01	*b*/01
frame 2	*c*/00	*c*/10	*c*/10	*c*/10	*c*/10	*e*/10	*e*/10	*e*/10	*e*/10	*e*/00	*d*/10
frame 3	*d*/00	*d*/00	*d*/00	*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*	*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*/0	→*a*/0	→*a*/1	→*a*/1	→*a*/1	*e*/1	*e*/1	*e*/1	*e*/1	→*e*/1	*d*/1
frame 1	*b*/0	*b*/0	*b*/0	*b*/0	*b*/1	→*b*/0	→*b*/1	*b*/0	*b*/1	*b*/1	*b*/0
frame 2	*c*/0	*c*/1	*c*/1	*c*/1	*c*/1	*c*/0	*c*/0	*a*/1	*a*/1	*a*/1	*a*/0
frame 3	*d*/0	*d*/0	*d*/0	*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; here, the page is emboldened) 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 at times 2, 4, and 7 are writes (represented by the bold page references). 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*/00	→*a*/00	→*a*/11	→*a*/11	→*a*/11	***a***/00	***a***/00	*a*/11	*a*/11	→*a*/11	***a***/00
frame 1	*b*/00	*b*/00	*b*/00	*b*/00	*b*/11	***b***/00	***b***/10	***b***/10	***b***/10	***b***/10	*d*/10
frame 2	*c*/00	*c*/10	*c*/10	*c*/10	*c*/10	*e*/10	*e*/10	*e*/10	*e*/10	*e*/10	→*e*/00
frame 3	*d*/00	*d*/00	*d*/00	*d*/10	*d*/10	→*d*/00	→*d*/00	→*d*/00	→*d*/00	*c*/10	*c*/00
fault						1				2	3
loaded						*e*				*c*	*d*
removed						*c*				*d*	*e*

Variable Number of Frames

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 $\tau = 4$ units. We also assume that a was referenced at time 0, d at time -1, and e at time -2. The window begins with the current reference.

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

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 p, then all pages not referenced between those page faults are removed. This table shows the pages resident at each reference. Here, we take $p = 2$ units and assume that initially, a, d, and e are resident. This example assumes the interval between page faults does *not* include the reference that caused the previous page fault.

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