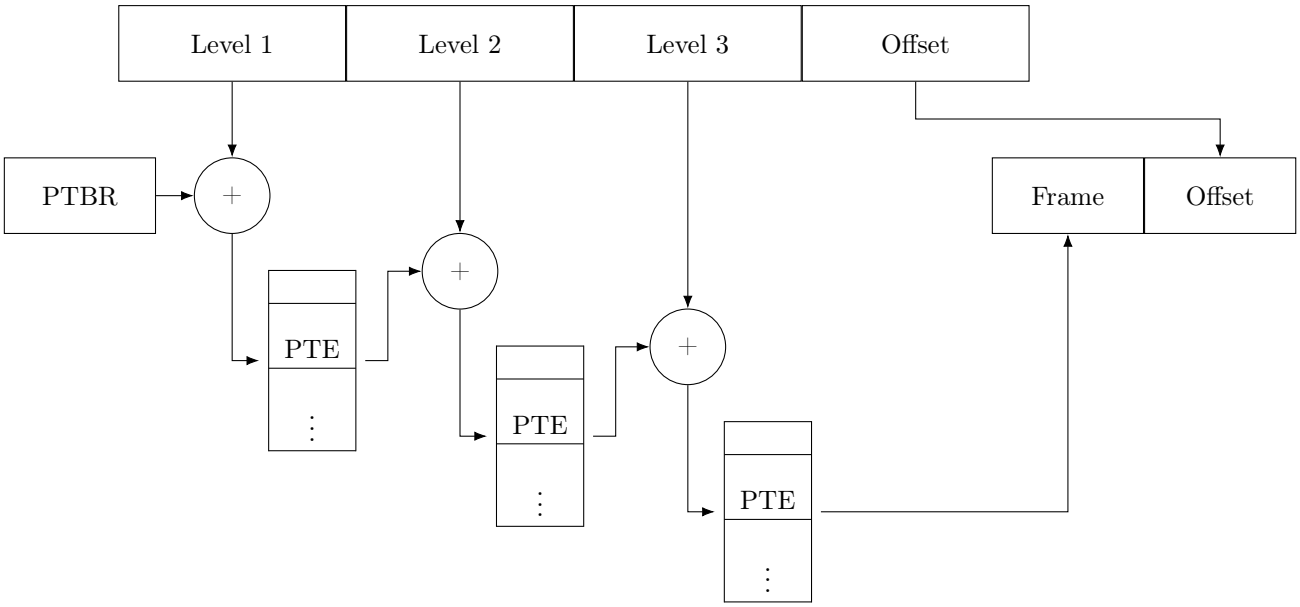
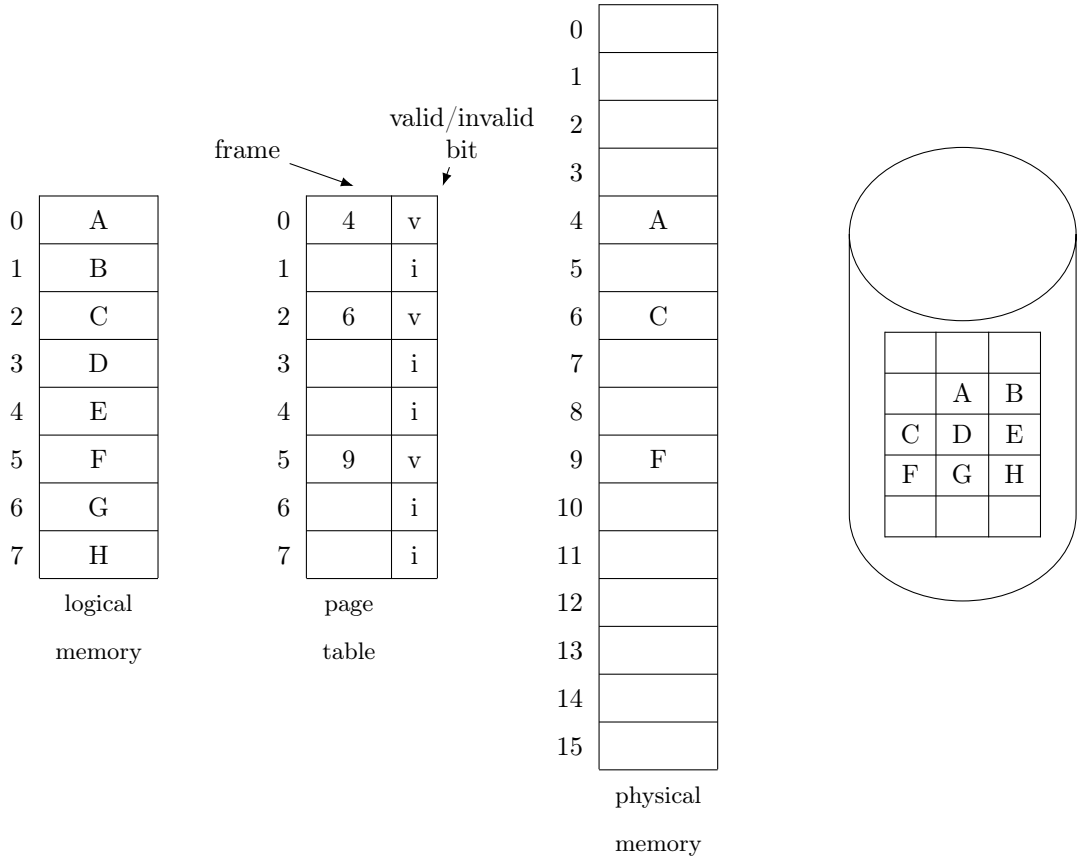
When utilizing virtual memory, we often have to deal with *page faults* := the failure of the hardware's rule for page lookup because:

(a) the user lacks the permissions to access the page
(b) the page is invalid
(c) the page does not exist

The hardware attempts to look the page up in the page table using the index of the form

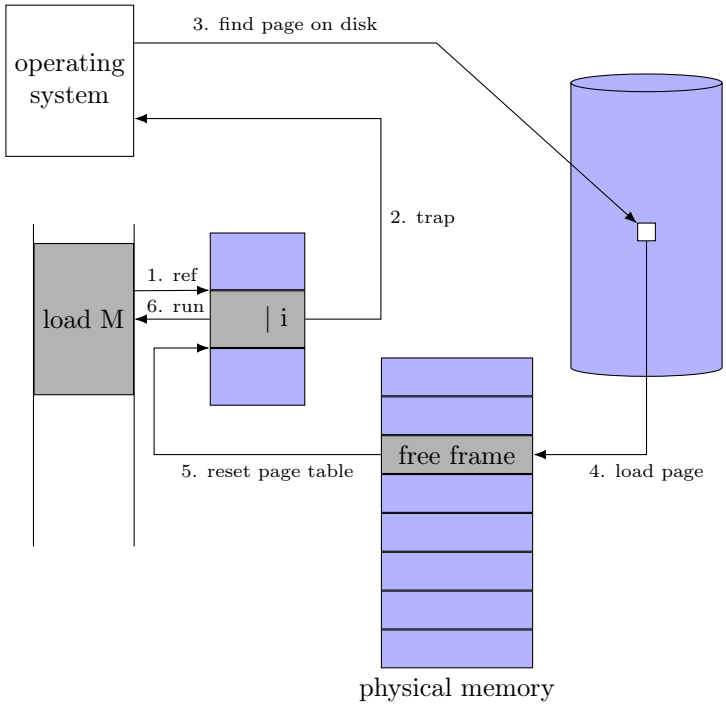


Which then finds the page like so



But if the page is not in memory, we have a page fault! The page fault is treated like any other fault:

- the OS looks in the interrupt source vector (aka trap table) and jumps to indicated page; note that we could end up trapped if this page faults as well
- The kernel has a few options on a page fault
  1. the kernel level fix: assume the program is buggy and kill it
  2. the program level fix: arrange the stack as if there were no fault and send signal
  3. the data level fix: change the page table entry and jump before the fault (the slowest)



The third option is called *paging*; we use it to simulate a big machine on a small one (for instance).

The kernel memory where the physical pages are cached from the disk is the *swap space*.

0.1 Page Replacement

```
// we assume that we have implemented the following functions:
int swapmap(int process, int virtual_address);
// this returns the disk address in the swap space or FAIL
tuple (process, va) removal_policy()
// this returns the process and virtual address of the next victim page
int pmap(int va);
// refers to the physical address mapped to by a given virtual

void pfault(int va, int proc, int access_type...) {
  if (swapmap(proc, va) == FALL) kill(proc);
  else {
    (vic_proc, vic_va) = removal_policy();
    int vic_pa = vic_proc->pmap(vic_va);
    vic_proc->pmap(vic_va) = FAIL;
    /// write vic_pa to flash at location swapmap(vic_proc, vic_va)
    // read vic_pa from flash at location swapmap(proc, va)
        proc->pmap(va) = pa;
      }
    }
```

How do we decide the removal policy? We need some heuristic for when to swap a page:

(a) Nobody needs the page (and there is no dynamic linkage to it)
(b) Page has not changed since load (and therefore RAM data is valid)
(c) Page is not needed for a while (based on an approximation

Here follows a few simple page replacement policies:

ORACLE

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	

page frames

Figure 1: 9 replacements

FIRST IN, FIRST OUT

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	7	7	7	
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	0	0	
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	1	

page frames

Figure 2: 15 replacements

Note that with FIFO, increasing the number of pages increases the number of replacements; this is called *Beladay's Anomaly*, and it occurs because FIFO is not a *stack algorithm* (ie the state of a smaller cache at any point is not a subset of the larger cache),

FIFO is easy to implement, since the table is in the kernel and the kernel retrieves pages; we just keep a table of when each page was brought into RAM.

LEAST RECENTLY USED

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	

page frames

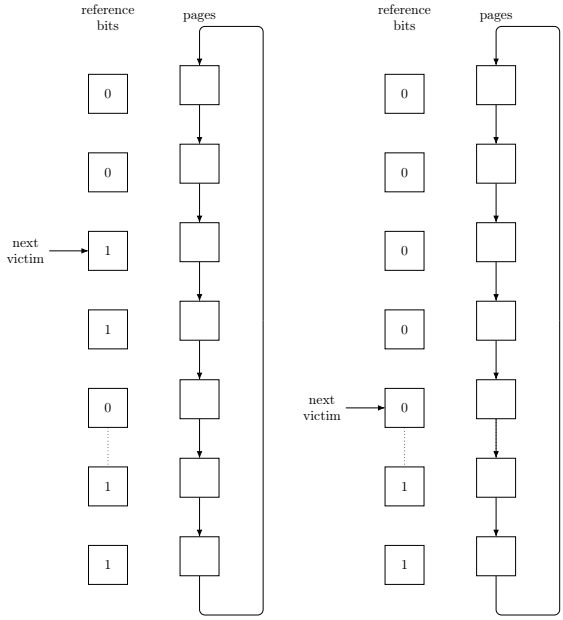
Figure 3: 12 replacements

LRU is more difficult to implement, since the kernel does not take over on each access and therefore has no chance to edit the last accessed time. We can address this in one of a few ways:

1. get hardware support
    - this is slow and rarely used
2. periodic invalidation
    - Procedure:
        (a) the kernel periodically invokes a timer fault to mark all pages as invalid
        (b) any succeeding access will then cause a fault which allows the kernel to update times
    - for clock precision, this is done in hardware after about a second passes. The time choice is important:
        - a long gap means less interruption
        - a short gap means better approximation

Possible Optimizations:

1. Demand Paging
    - Procedure:
        1. start with no pages in RAM
        2. c startup routine (cstrt()) causes initial fault
        3. call to main causes a second fault
    - This strives to lessen wait time by skipping unnecessary copies
        - best case: we only need the two pages
        - worst case: we need all the pages. This performs worse than the standard approach since it prevents us from batch copying.
2. Copy on Write
    - only give processes read permission and do not copy entries write is attempted
    - this is done by vfork() to get the benefit of threading without race conditions
    - vfork() is equivalent to fork, except:
        (i) Parent and child can share memory.
        (ii) Parent is frozen and cannot run until child either exits or execs
3. Dirty Bit
    - a special bit in a page table entry indicates whether the page has been modified.
    - The bit is set to 1 if it has been modified since last load, 0 if not
    - we only need write a victim to memory if the dirty bit is 1
    - often implemented by making pages O_RDONLY so that the first access is a fault
    - sometimes implemented using the clock algorithm, which works like the elevator


reference bits
pages
0
0
next victim
1
1
0
1
1
reference bits
pages
0
0
0
0
next victim
0
1
1