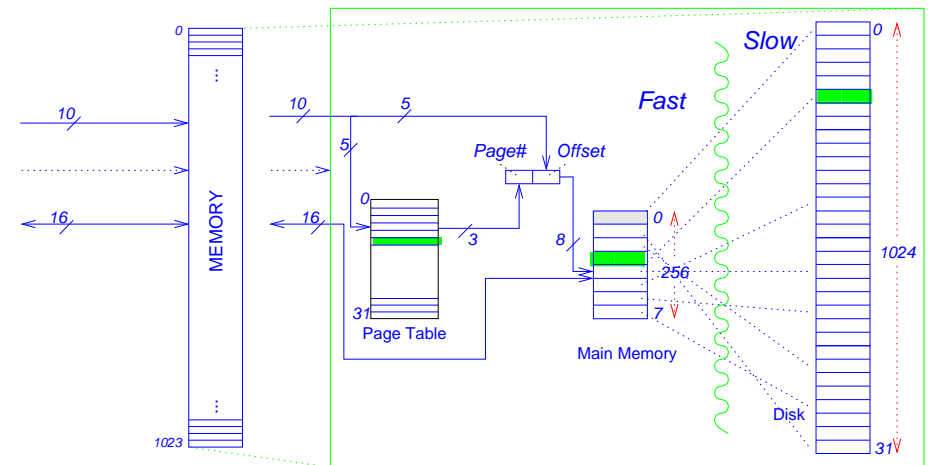


## Virtual Memory in PeANUt

- ref: [PeANUt Spec, sect 3]; additionally [O'H&Bryant, sect 10.1–10.7] or [Null&Lobur, sect 6.5]
- virtual memory implementation
  - page tables
- virtual memory in PeANUt
  - page replacement
  - paging behaviour
  - working sets
- other issues:
  - calling procedures inside procedures: example in appendix to lect P8
  - reminder: no labs next week – work on Assignment 2!

## The PeANUt Virtual Memory System



## Virtual Memory in PeANUt

- our previous model of the PeANUt considered memory to be a *black box*
  - data was written or read by the memory automatically
  - suppose paging is implemented: how would you know if the data is in main memory or on the disk?
- a closer look at PeANUt memory:
  - main memory (MM): 256 cells, in  $8 \times 32$ -cell page frames
  - secondary memory (SM): 1024 cells, in  $32 \times 32$ -cell pages
- VM is controlled by a Memory Management Unit (MMU)
  - not explicitly shown in the PeANUt architecture
  - uses a page table with 32 entries (one per page)
  - translates virtual addresses into physical addresses
  - (supports OS to) move pages between main memory and disk whenever necessary

## Page Tables

- purpose: give information on the status of each of the pages of the virtual memory
    - is it present in main memory? Present bit
    - if so, in which page frame is it? Frame number field (3 bits)
    - if so, has data been written to it? Dirty bit
- |                   |   |   |       |
|-------------------|---|---|-------|
| Other information | D | P | Frame |
|-------------------|---|---|-------|
- where does this page table reside?
    - possibly in special hardware
    - in PeANUt: part of main memory, at VM addresses 0–31 (large: 1/8 of main memory!)

## How Does PeANUt Virtual Memory Work?

- to read some data (at say address 00101 01010)
  1. the address is split into two parts: a page number (00101) and a page offset (01010)
  2. the status of the page is checked with the page table (entry 5):
 

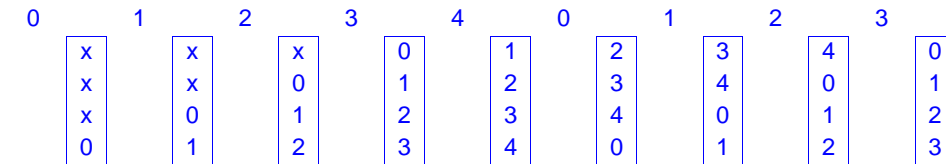
Other information	D	P	Frame
-------------------	---	---	-------
  3. A) if page is in main memory (P is set):
    - lookup its page frame number (say page frame 3 (011))
    - B) otherwise, a page fault occurs:
      - select a page frame number (say 011) to put the page into
      - may require the removal of another page (the 'victim'), if present in frame 3
        - ◆ copying back to secondary memory if 'dirty'
      - fetch (5th) page from secondary memory into main memory
  4. combine page frame number with the page offset to produce an 8 bit address (011 01010), giving the location of the cell in main memory
- to write some data: almost the same procedure as above  
(must also set the dirty bit (D) in the page table)

## Page Replacement

- how do we decide which page to throw out?
- Random: cannot get any performance advantages from locality of reference
- First In First Out (FIFO):
  - hardware/OS keeps track of age of each page
  - replace the oldest page
  - problem: any page continuously required must eventually be thrown out!
- Least Recently Used (LRU):
  - hardware keeps track of access times to each page
  - replace the least recently used page
  - generally performs quite well

## Paging Policy Failure

- example (Belady's anomaly): access pages 0, 1, 2, 3, 4 cyclically

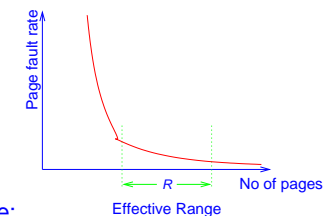


(resident pages, oldest at top)

- here, five pages are needed continuously, but there are only four page frames
- under both LRU and FIFO policies, the page required next is removed!
  - this problem can only be solved if future page demand can be predicted!
  - problems are inevitable when the demand for pages exceeds the number of page frames available
- what would be a better replacement policy? what would be an optimal policy?

## Paging Behaviour

- how many pages? (or, how big are the pages?)
  - too few (too big) can result in un-needed data put in main memory, resulting in fragmentation, in turn causing page faults (when in 'steady-state')
  - too many results in huge page tables (and other large per-page overheads, including a large number of page faults initially, with a large number of disk accesses)

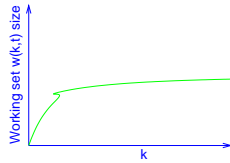


when program is in steady-state:

Assumption: page demand < available pages in main memory

## Working Set

- the working set function  $w(k,t)$  is defined as the set of pages accessed during the  $k$  most recent memory accesses, up until time  $t$



- as  $k$  increases, the size of the  $w(k,t)$  approaches a (constant) limit
    - this limit is the working set size at time  $t$
    - it is roughly independent of  $t$
  - the working set size concept can be used to describe the memory needs of a program
    - if the main memory is not large enough to hold the working set, there will be excessive page faults (such behaviour is called thrashing)
- is also an important concept in the memory hierarchy

## The Page Table in PeANUt

- is permanently located in page 0 (addresses a0 to a37)
- it has 32 entries, each relating to one of the 32 VM pages
- 5 bits of a memory (VM) address are used to determine the page frame number, 5 bits for the offset within the page

- format of a page table entry:

Unused	Last used	Swap count	D	P	Frame number
15	11 10 8 7	5	4	3	2 0

- page table fields:
  - Last used: reflects how recently the page has been accessed (a value of 0 indicates that this is the most recently accessed page)
  - Swap count: indicates how many page faults have occurred since this page was loaded into memory (indicating its age)
  - Dirty flag: a value of 1 indicates that data has been written to this page since it was swapped into memory
  - Present flag: a value of 1 indicates the page is present in main memory
  - Frame number: states which page frame (slot in main memory) a page is in (if the page is present)

## PeANUt Virtual Memory Traps

- trap #11: page fault (exception)
  - Is not user-initiated or user-definable. Can occur in the fetch of an instruction or the evaluation of its operand. Note: it causes *re-execution* of the faulting instruction.
- trap #12: swap page in
  - Move page number AC from secondary memory to main memory. The O/S will use page table entry AC (at Mem[AC]) to determine the destination page frame
- trap #13: swap page out
  - The O/S will move page number AC from main memory to secondary memory
- the handler routine for trap #11 will use trap #12 and trap #13, in accordance to which paging policy it uses
- example using the FIFO policy: vmfifo.ass
- Q: what if the handler routine for trap #11 was swapped out? Similarly for the page tables? Can the routine use the stack (safely)?