운영체제의 기초:
Demand Paging

2023년 5월 18, 23, 25, 30일

홍 성 수
sshong@redwood.snu.ac.kr
SNU RTOSLab 지도교수
서울대학교 전기정보공학부 교수

Agenda

I. Background
II. Issues
III. Thrashing and Working Set
IV. Miscellaneous Issues

I. Background

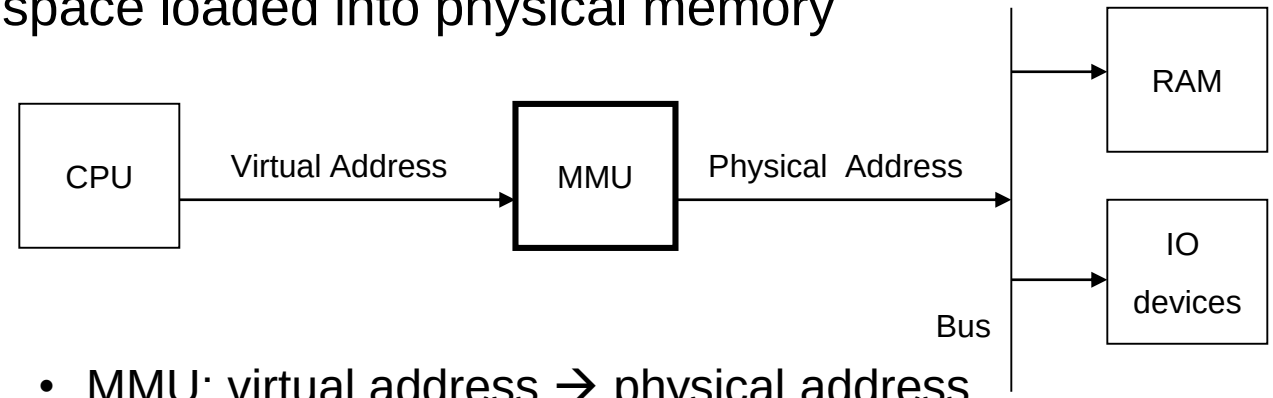
Motivation (1)

- "Address mapping mechanism" in previous lectures
  - Mechanism: Mapping hardware (MMU and mapping tables)
  - Separates programmer's view of memory from system's
    - Each user sees its own memory organization
    - Allows OS to shuffle users around and simplifies memory sharing between users

- So far, we have assumed that user processws are completely loaded into memory
  - Wasteful because of locality of reference
  - Process only needs a small amount of its total memory at any given time

Motivation (2)

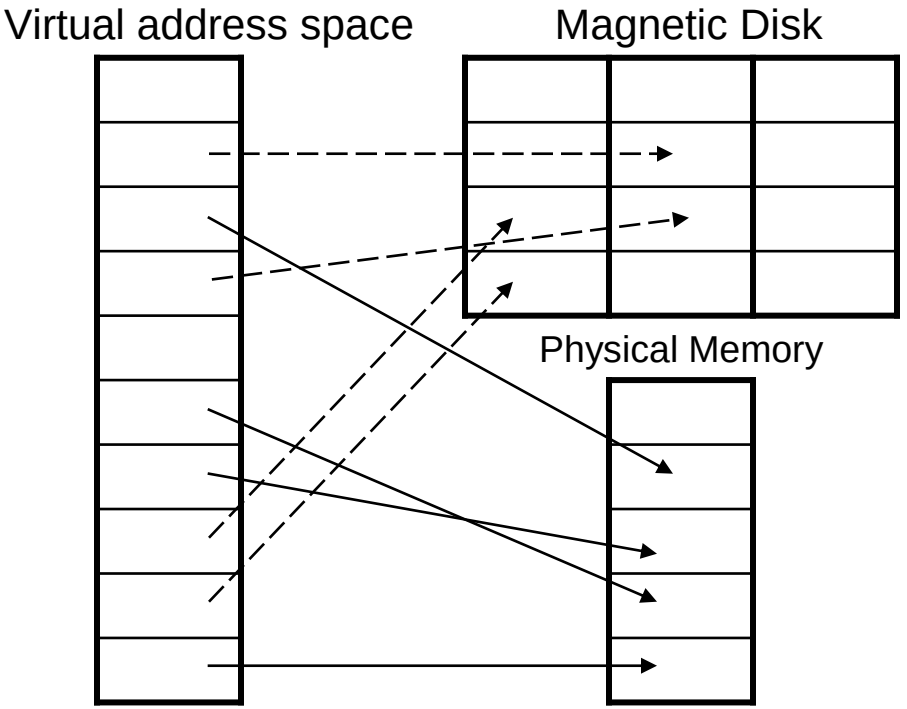
- Solution: Virtual memory
  - Allows process to run with only some of its virtual address space loaded into physical memory



  - MMU: virtual address → physical address
  - Virtual address
    - Generated by CPU core and used by programmer
    - Each process has its own virtual address space
  - Physical Address
    - Address on bus
    - Only one physical address space exists in a system

Basic Workings

- Virtual-to-physical address layout



Why Demand Paging? (1)

- Virtual address translates to either
  - Physical memory
    - Costly, small, fast
  - Disk (backing store)
    - Cheap, large, slow
  - Error — Not valid
    - Free

Why Demand Paging? (2)

- Idea
  - Produce the illusion of a disk as fast as main memory
    - Works because programs spend their time in only a small piece of the code
    - Knuth's estimate: 90% of the time in 10% of the code
- Key principle
  - Locality
    - Spatial locality
    - Temporal locality

II. Memory Management Mechanism

1. Page Fault Handling (1)

- Why page faults?
  - If not all of process is loaded when it is running, what happens when it references a word that is only in the backing store?
  - Hardware and software cooperate to make things work anyway

① Extend the page tables with an extra bit "*valid* (*present*)"
- If the valid bit isn't set, a reference to the page results in trap
- This trap is given a special name, page fault

② Any page not in main memory right now has the "valid" bit cleared in its page table entry

1. Page Fault Handling (2)

- Why page faults? (cont'd)
  - ③ When page fault occurs
    - Operating system brings the page into memory
    - The page table is updated: "present" bit is set
    - The process continues execution

1. Page Fault Handling (3)

- Problem in page fault handling
  - Continuing process is very tricky since page fault may have occurred in the middle of an instruction
    - Don’t want user process to be aware that the page fault even happened
    - Can the instruction just be skipped?
      - No: Wouldn’t be transparent to the process
    - The instruction has to be restarted from the beginning
      - What about instruction like:

        MOVE (SP)+, -(R2) or Block transfer
    - Requires hardware support to restart instructions

1. Page Fault Handling (4)

- Two kinds of memory-related faults
  - TLB fault
    - Required virtual to physical address translation is not in TLB
  - Page fault
    - Contents of a virtual page are either not initialized or not in memory

1. Page Fault Handling (5)

- Important facts regarding page/TLB fault
  - *Every* page fault is preceded by a TLB fault
    - If the contents of the virtual page are not in memory, a translation cannot exist for it
  - *Not every* TLB fault generates a page fault
    - If a page is in memory and the translation is the page table, the TLB fault can be handled without generating a page fault

1. Page Fault Handling (6)

- On each page fault
  - ***Stop*** the instruction that is trying to translate the address until we can retrieve the contents
  - ***Allocate*** a page in memory to hold the new page contents
  - ***Locate*** the page on disk using the page table entry
  - ***Copy*** the contents of the page from disk
  - ***Update*** the page table entry to indicate that the page is in memory
  - ***Load*** the TLB
  - ***Restart*** the instruction that was addressing the virtual address we retrieved

1. Page Fault Handling (7)

- To swap out a page to disk
  - ***Remove*** the translation from the TLB, if it exists
  - ***Copy*** the contents of the page to disk
  - ***Update*** the page table entry to indicate that the page is on disk

III. Memory Management Policy

Key Issues in Demand Paging

1. Page selection
   - When to bring pages into memory and which?
2. Page replacement
   - Which page(s) should be thrown out and when?
3. Page frame allocation
   - Global vs. local
4. In local page frame allocation
   - Static
     - # of page frames are fixed (LRU algorithm)
   - Dynamic
     - # of page frames are varying (working set algorithm)

1. Page Selection (1)

- Page selection policies
  1. Demand paging
     - Start up process with no pages loaded
     - Load a page when a page fault for it occurs
       - Wait until the page is in memory
     - Almost all paging systems in the past were like this
  2. Prepaging
     - Bring a page into memory before it is referenced
     - When a page is referenced, bring in the next one, just in case
     - Hard to do effectively without a prophet
     - Sometimes works: sequential read-ahead

1. Page Selection (2)

- Page selection policies (cont'd)
  3. Request paging
     - Let user say which pages are needed
     - What's wrong with this?
       - Users don't always know best
       - Users aren't always impartial (They overestimate needs)
     - Example: Overlay

2. Page Replacement (1)

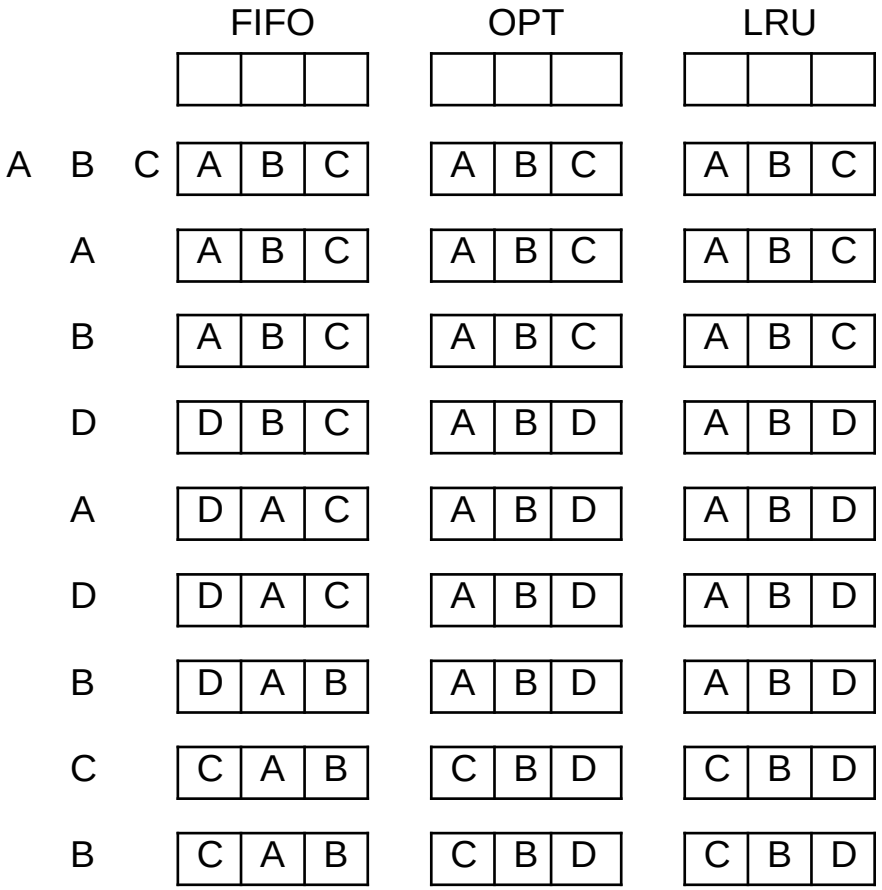
- Page replacement algorithms
  1. OPT
     - Throw out the page that won't be used for the longest time into the future (by Belady)
     - As always, the best algorithm arises if we can predict the future
     - It isn't practical, but it is good for comparison
  2. Random
     - Pick any page at random
     - Works surprisingly well!

2. Page Replacement (2)

- Page replacement algorithms (cont'd)
  3. LRU
     - Throw out the page that hasn't been used in the longest time
     - Use the past to predict the future
       - With locality, LRU approximates OPT
  4. FIFO
     - Throw out the page that has been in memory the longest
     - The idea is to be fair
     - Give all pages equal residency

2. Page Replacement (3)

- Example reference string: A B C A B D A D B C B



2. Page Replacement (4)

- Stack algorithms
  - Belady's anomaly: Page fault rate may increase as the number of allocated frames increases
    - Reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 with FIFO algorithms with frames 3 and 4, respectively
  - Stack algorithms never exhibit Belady's anomaly
    - A set of pages in memory for *n* frames is always a subset of pages in memory for *n*+1 frames

2. Page Replacement (5)

- Stack algorithms (cont'd)
  - LRU is a stack algorithm
    - Proof sketch:
      - The set pages in memory with *n* page frames always include *n* most recently referenced pages
        - *M*(*m*, *r*) = the set of m pages of the least page age
        - Clearly, *M*(*m*+1, *r*) = the set of *m*+1 pages of the least page age
        - *M*(*m*+1, *r*) contains M(*m*, *r*)

2. Page Replacement (6)

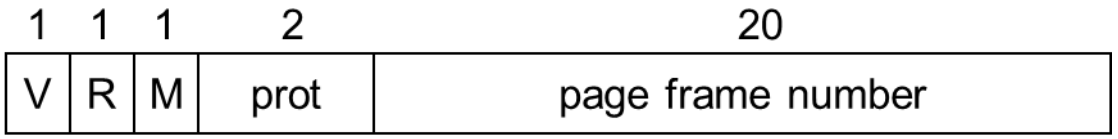
3. LRU algorithm
   - Naïve implementation

```
/* on each memory reference:*/
long timeStamp = System.currentTimeMillis();
sortedList.insert(pageFrameNumber,timeStamp);
```

   - Too inefficient
     - Time stamp + data structure manipulation on each memory operation
   - Too complex for hardware

2. Page Replacement (7)

4. LRU approximations
   - Make use of hardware support: "*reference bit*"
     - Reference bit is set when pages are accessed
     - Can be cleared by the OS



   - Trade off accuracy for speed
     - It suffices to find a "pretty old" page

2. Page Replacement (8)

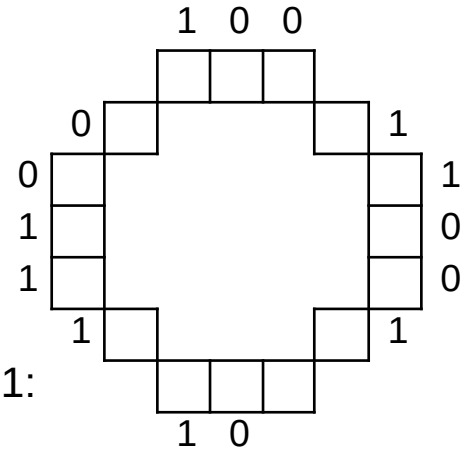
4. LRU approximations (cont'd)
   1) Additional reference bits algorithm
      - Each page has a reference bit and an 8-bit register
        - "*Reference byte*"
      - At a regular interval, (R-bit, register) is shift to right
      - A page with the smallest register value is the LRU page
      - Pro:
        - Does not impose overhead on every memory reference
        - Interval rate can be configured
      - Con:
        - Scanning all page frames can still be inefficient
        - E.g., 4 GB of memory, 4KB pages → 1 million page frames

2. Page Replacement (9)

4. LRU approximations (cont'd)
   2) Clock algorithm
      - Background: general idea of the reference bit
        - Keep a reference bit for each page frame
        - Hardware sets the appropriate bit on every memory reference
        - OS clears the bits from time to time
        - OS uses the bit to figure out how often pages are being referenced
      - Aka "*second chance algorithm*"
      - Used in Unix for VAX-11
        - Why didn't they use hybrid FIFO-LRU?
        - It's too difficult to find the right residence set size

2. Page Replacement (10)

4. LRU approximations (cont'd)
   2) Clock algorithm (cont'd)
      - On page eviction:
        1. Circulate through the list of reference bits
        2. If the value is zero, replace this page
        3. If the value is one, set the value to zero; go to 1:
      - Pro: Very low overhead
        – Only runs when a page needs evicted
        – Takes the first page that hasn't been referenced
      - Con: Isn't very accurate (one measly bit!)
        – Degenerates into FIFO if all reference bits are set
      - Pro: But, the algorithm is self-regulating
        – If there is a lot of memory pressure, the clock runs more often (and is more up-to-date)



2. Page Replacement (11)

4. LRU approximations (cont'd)
   3) Enhanced clock algorithm
      - Some systems use a "*dirty bit*" to give preference to dirty pages
        - Clean ones need not be written to disk
        - More expensive to throw out dirty pages
      - Used in Macintosh OS
      - Problem
        - What if clean pages are frequently accessed?
        - Example: code pages

2. Page Replacement (12)

4. LRU approximations (cont'd)

- Performance of the clock algorithm
  - What does it mean if the clock hand is sweeping very fast?
    - Not enough memory
    - BSD Unix uses the Clock Algorithm: Sun OS
    - "`vmstat`" command gives info
    - "`sr`" — pages scanned by clock algorithm, per-second
    - `vmstat -s`:
      - 292853 pages examined by the clock daemon
      - 6 revolutions of the clock hand
      - 127878 pages freed by the clock daemon
    - `vmstat 5`:
      - VM activity every 5 seconds

2. Page Replacement (13)

4. LRU approximations (cont'd)
   - Needs "*reference bit*" (AKA "*use*" bit)
     - Most CPUs don't support it in hardware
     - So software emulation is performed

2. Page Replacement (14)

- Software emulation of a reference bit
  - Basic operations
    - Set the bit: done by hardware when a page is referenced
    - Clear the bit: done by the kernel, usually periodically
  - Set the bit
    - How does the kernel know of a reference to a page?
      - Through a page fault
    - Upon a page fault, the kernel checks if it is a genuine fault
    - If not, the kernel sets both the reference and valid bit to one
    - And then flushes TLB
  - Clear the bit
    - The kernel clears both the reference and valid bit
    - And then flushes TLB

2. Page Replacement (15)

5. FIFO-based algorithm
   - Used in VAX/VMS operating system
   - Data structures
     - Each process has a "*resident set list*" ordered in FIFO
       - For a process, the resident set is the set of pages in memory
       - In the initial VMS, the resident set size is fixed to all processes
       - In the later VMS, processes may have different sizes
     - Keeps a system-wide "*free page list*"
       - Used alongside to offset the weakness of FIFO
       - Size of the free page list is maintained at run-time

2. Page Replacement (16)

5. FIFO-based algorithm (cont'd)
   - Operation
     - On page fault, frames are taken from the head of the free list
       - The desired pages are written into the selected page frames
     - Old pages are freed from the resident set list to the free list
       - But they may be reused when requested
     - Maintains a system-wide "*modified page list*"
       - When the swap device is idle, they are written out and become clean

2. Page Replacement (17)

5. FIFO-based algorithm (cont'd)
   - Also called "*hybrid FIFO-LRU*" algorithm
     - The resident set is managed according to FIFO
     - The global free page list behaves similarly to LRU
     - There exists *a resident set size* for which FIFO-LRU achieves a page fault rate close to the pure LRU while incurring a cost comparable to that of the FIFO
       - Important to dynamically find the right resident set size

3. Page Frame Allocation (1)

- Three different styles for page frame allocation
  1. Global allocation
     - All pages from all processes are lumped into a single allocation pool
     - Each process competes with all the other processes for page frames
  2. Per-process allocation
     - Each process has a separate pool of pages
     - A page fault in one process can only replace one of the process's frames
     - This relieves interference from other processes
  3. Per-job allocation
     - Lump all processes for a given user into a single allocation pool

3. Page Frame Allocation (2)

- Pros and cons
  - In per-process and per-job allocation, OS must have a mechanism for (slowly) changing the allocations to each pool
  - Otherwise, can end up with very inefficient memory usage
  - Global allocation provides most flexibility but the least “hog protection”

IV. More on Page Selection: Thrashing and Working Set

1. Thrashing (1)

1. Rationale
   - What happens when memory gets overcommitted?
     - Suppose that there are many users and that between them their processes are making frequent references to 50 pages, but memory has only 49 pages
     - Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out
     - System will spend all of its time reading and writing pages
       - It will be working very hard but not getting anything done
     - Average memory access time equals disk access time
     - Illusion breaks: Memory access will look as slow as a disk rather than disks being as fast as memory
       - Effective memory access time: $t_{eff\_acc} = pt_{mem\_acc} + (1-p)t_{fault}$
       - Thrashing was a severe problem in early demand paging systems

1. Thrashing (2)

2. Why does the system incur thrashing?
   - *Because* it doesn't know when it has taken on more work than it can handle
   - LRU mechanisms order pages in terms of last access but don't give absolute numbers indicating pages that mustn't be thrown out
   - What's the human analogy to thrashing?
     - Too many courses? – What's the solution? Drop one!

1. Thrashing (3)

3. What can be done about thrashing in OS?
   - If a single process is too large for memory, there is nothing the OS can do
     - That process will simply thrash
   - If the problem arises because of the sum of several processes
     - *Figure out how much memory each process needs*
     - Change scheduling priorities to *run processes in groups* whose memory needs can be satisfied
     - Shed load

1. Thrashing (4)

4. Impact
   - The issue of thrashing may be less critical for PCs than for time-shared machines
     - With just one user, one can kill jobs when the response gets bad
     - With many users, OS must arbitrate between them

2. Working Set (1)

1. Key ideas
   - A solution proposed by Peter Denning in 1968
   - An informal definition
     - The collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing
   - The idea behind the working set is to utilize the recent needs of a process to predict its future needs
     - Choose t, the working set parameter
       - At any given time, all pages referenced by a process in its last t seconds of execution are considered to comprise its working set
     - A process will never be executed unless its working set is resident in the main memory
     - Pages outside the working set may be discarded at any time

2. Working Set (2)

2. Rationale
   - Effect of choice of what pages to be kept in main memory
     - If too many pages of a process are kept in main memory
       - Fewer other processes can be ready at any one time
     - If too few pages of a process are kept in main memory
       - Page fault frequency is greatly increased
       - The number of active processes executing approaches zero

2. Working Set (3)

3. "*Working set strategy*"
   - A process can be in memory *iff* all of the pages that it is currently using can be in memory
   - *All or nothing model*
     - If the pages a process needs to use increase and there is no room in memory, the process is swapped out of memory to free the memory for other processes to use

2. Working Set (4)

4. Benefits
   - By swapping some processes from memory, processes finish much sooner than they would if the computer attempted to run them all at once
   - The processes also finish much sooner than they would if the computer only ran one process at a time to completion, since it allows other processes to run and make progress during times that one process is waiting on the hard drive or some other global resource
   - The working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible
     - Thus it optimizes CPU utilization and throughput

2. Working Set (5)

5. Problem with the working set model
   - Working set window size varies with processes and time
   - OS must constantly update working set information
     - What pages have been accessed in the last t seconds?

2. Working Set (6)

6. One solution: Take advantage of reference bits
   - OS maintains "*idle time*" value of each page
     - *Idle time*
       - Amount of *CPU time* received by process since last access to page
   - Algorithm: Every once in a while, scan all pages of a process
     - For each reference bit on, clear page's idle time
     - For reference bit off, add process CPU time (since last scan) to idle time
     - Turn all reference bits off during scan
   - Scans happen on order of every few seconds

❖ In Unix, t is on the order of a minute or more

2. Working Set (7)

7. Other questions about the working set model
   - What should t be?
     - What if it's too large?
     - What if it's too small?
   - How do we compute working sets if pages are shared?
   - How much memory is needed in order to keep the CPU busy?
     - Note that under working set methods, the CPU may occasionally sit idle even though there are runnable processes
       – Ends up being "*non-work-conserving*" scheduling

3. Resident Set (1)

- Approximation of working set in VAX/VMS
  - "*Resident set*" of a process
    - A set of its pages resident in physical memory
  - Each process has a resident set limit
    - Max # of page frames that can be assigned to that process
  - Resident set limit is dynamically adjusted based on page fault rate
    - No need for costly computation of working set information

3. Resident Set (2)

- How can we use the notion of resident set
  - Thrashing avoidance with resident set
    - When a process is made inactive,
      all of its pages in its resident set are swapped out
      to the swap space
    - When a process is made active,
      all pages in its resident set are loaded
      - Each process has a resident set list in its context

4. Balance Set (1)

- Definition
  - A collection of *processes* whose working sets are resident in physical memory
- Motivation
  - Working sets are not enough by themselves to make sure memory doesn't get overcommitted
  - We must also introduce the idea of a "*balance set*"

4. Balance Set (2)

- How it works
  - Key idea
    - If the sum of the working sets of all runnable processes is greater than the size of memory,
      then refuse to run some of the processes (for a while)
  - Thus, we divide runnable processes into two groups
    - *Active* and *inactive*
    - The "*collection of active processes*" is the "balance set"
    - Only when its working set can be loaded,
      a process is made active
    - When a process is made inactive,
      its working set is migrated back to the swap device

4. Balance Set (3)

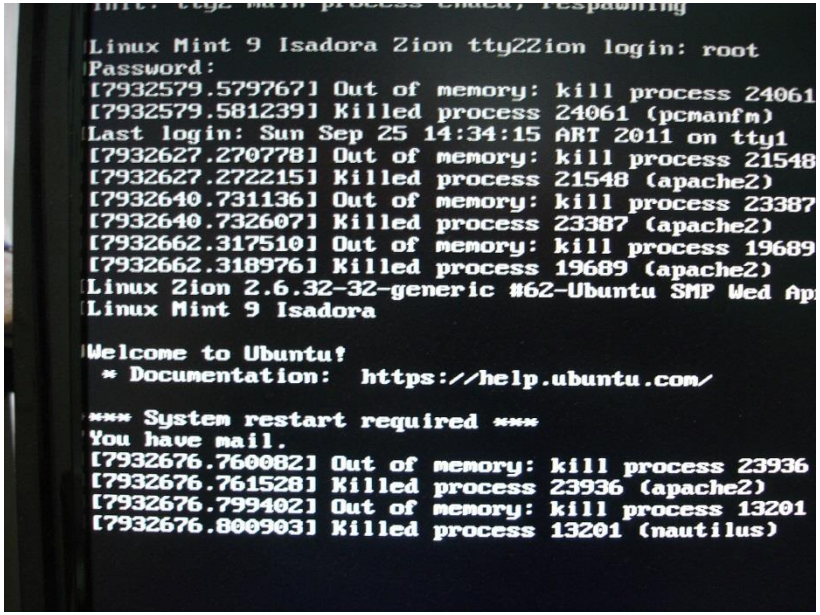
- Balance set management mechanism
  - Some algorithm must be provided for moving processes into and out of the balance set
  - The "*long-term scheduler*" does this in Unix
    - When the number of free pages goes low, it picks idle processes or less important processes and swaps out their pages back to the swap device
  - What happens if the balance set changes too frequently?
    - Thrashing

4. Balance Set (4)

- In Unix, **`swapper`** (process 0)manages the balance set and performs swapping
  - The first process created by OS
  - System process with no user context (daemon process)
  - It executes a routine called `sched()`
  - It is normally asleep and awakened once every second
  - It swaps out a process, if free mem is less than `t_gpgslo`
  - It calls `as_swapout()`, which cycles through each segment
  - To swap in a process, the swapper swaps in its `u_area`
    - When the process eventually runs, it will fault in other pages as needed

Out-of-Memory Killer in Linux (1)

- Out-of-memory (OOM)
  - Undesired state of computer operation where no additional memory can be allocated for use by OS
  - Occurs because all available memory, including disk swap space, has been allocated



Out-of-Memory Killer in Linux (2)

- Out-of-memory (OOM) killer
  - Typical OOM case happens when OS is unable to create any more virtual memory because all of its potential backing devices have been filled
  - Linux kernel mechanism which attempts to recover from OOM condition by terminating a low-priority process

V. Trends in Memory Management

Trends in Memory Management (1)

1. Larger physical memory
   - Page replacement algorithm is less important
     - Hopefully, it will rarely get invoked
   - Less hardware support for replacement policies
     - Software emulation of use and dirty bits
   - Larger page sizes
     - Better TLB coverage
     - Smaller page tables
     - Fewer pages to manage

Trends in Memory Management (2)

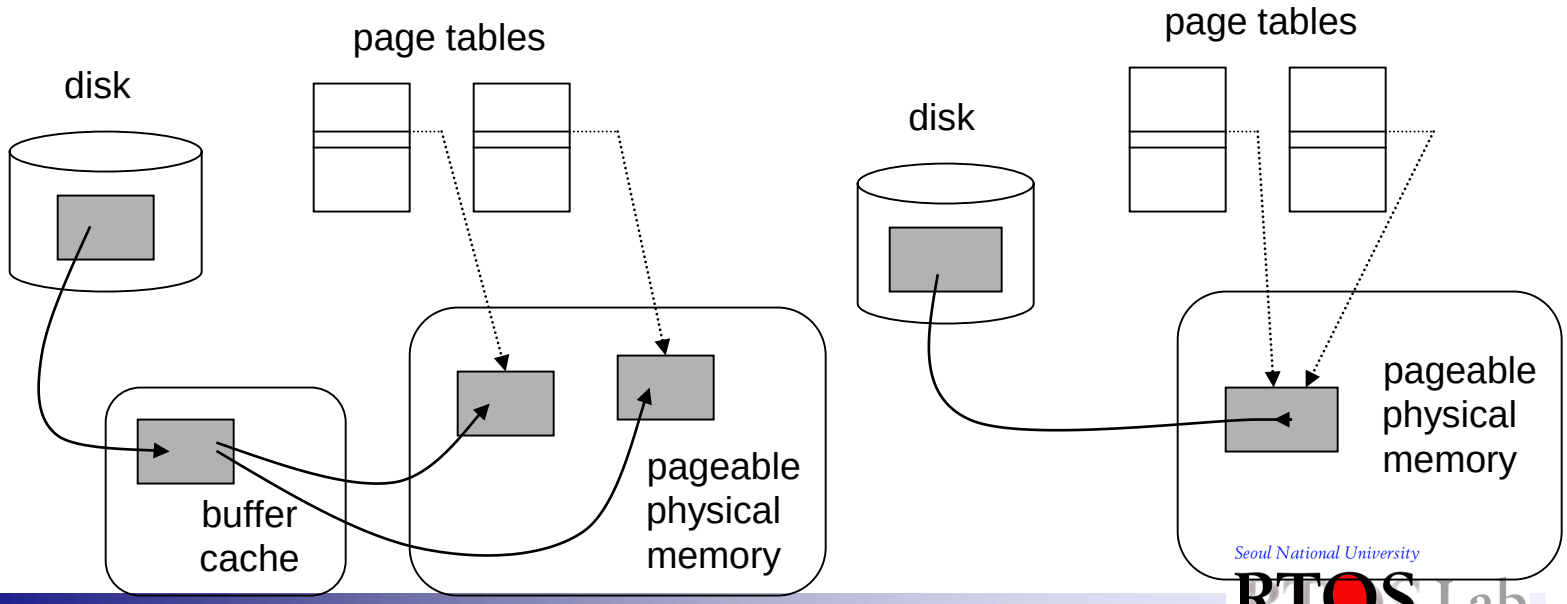
2. Larger virtual address spaces
   - 64 bits with 4K page: Max page table size 64K TB
   - Sparse address spaces
   - Inverted page tables
     - Hash table VA → PA
     - Scale with the size of physical memory

Trends in Memory Management (3)

3. File I/O using the virtual memory system
   - Memory mapped files
     - Uses VM system for file caching
     - Page fault handing for reads/writes
   - Make page replacement interesting again
   - More sequential behavior

Memory Mapped Files

- Map a disk block onto pages
  - **`paddr = mmap(addr,len,prot,flags,fd,offset)`**
    - [offset, offset+len) → [paddr, paddr+len)
    - **`fd`** is file descriptor returned by open()
  - Reduces no. of system calls and data copies for file access



`mmap()` System Call (1)

- **`mmap()`** creates a VMA in the virtual memory of the calling process

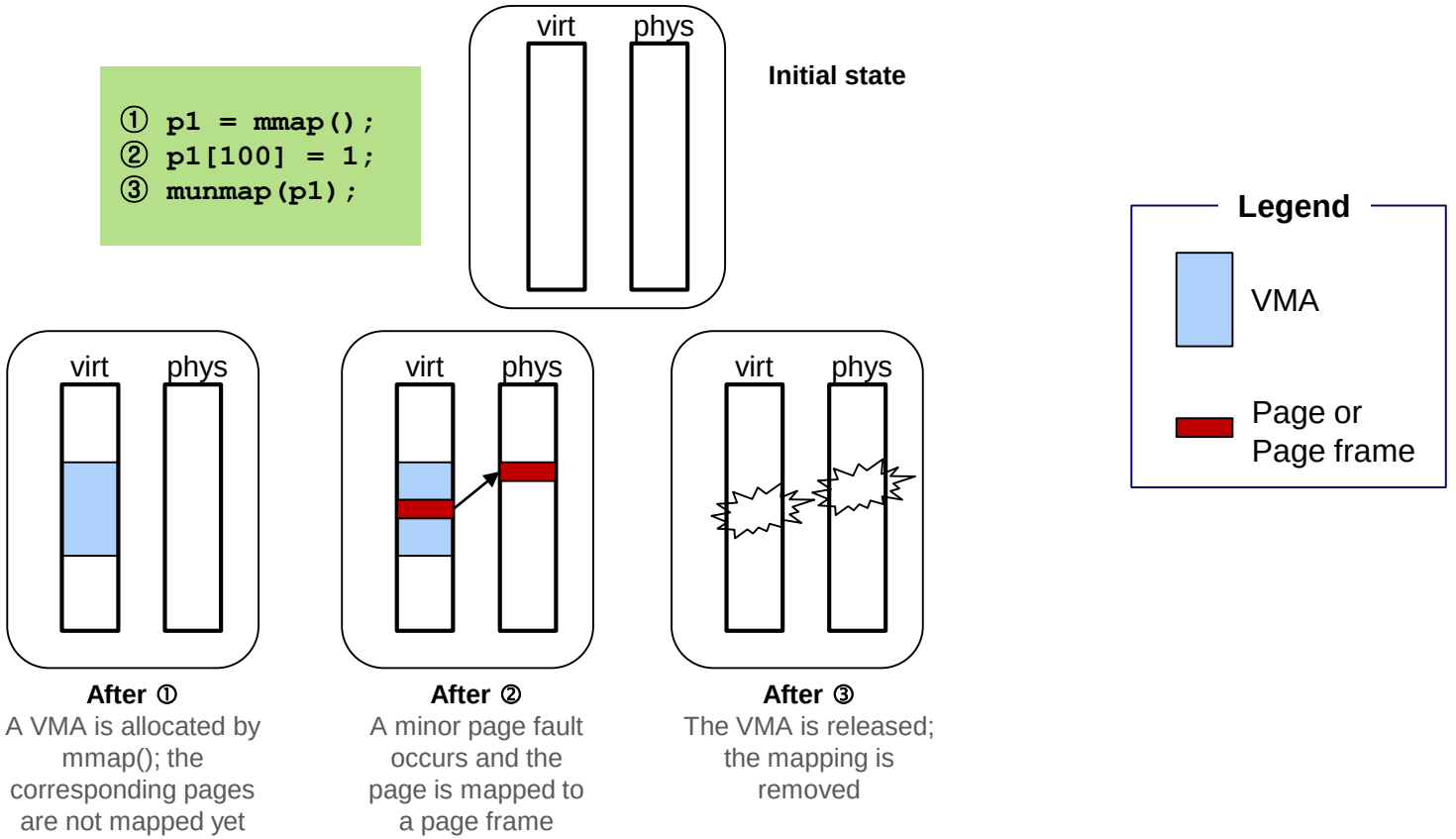
```
void *mmap(void *addr, size_t length,
int prot, int flags, int fd, off_t offset);
```

  - Case1: **`flags`** includes **`MAP_ANONYMOUS`**
    - Created VMA consists of anonymous pages
    - These are allocated to page frames later via a *minor page fault*
  - Case2: **`flags`** does **NOT** include **`MAP_ANONYMOUS`**
    - Created VMA consists of the pages which are mapped with the file specified by **`fd`**
    - These are allocated to a page frame later via a *major page fault*

mmap() System Call (2)

- Case1: `flags` includes **`MAP_ANONYMOUS`**

```
① p1 = mmap();
② p1[100] = 1;
③ munmap(p1);
```



After ①
A VMA is allocated by mmap(); the corresponding pages are not mapped yet

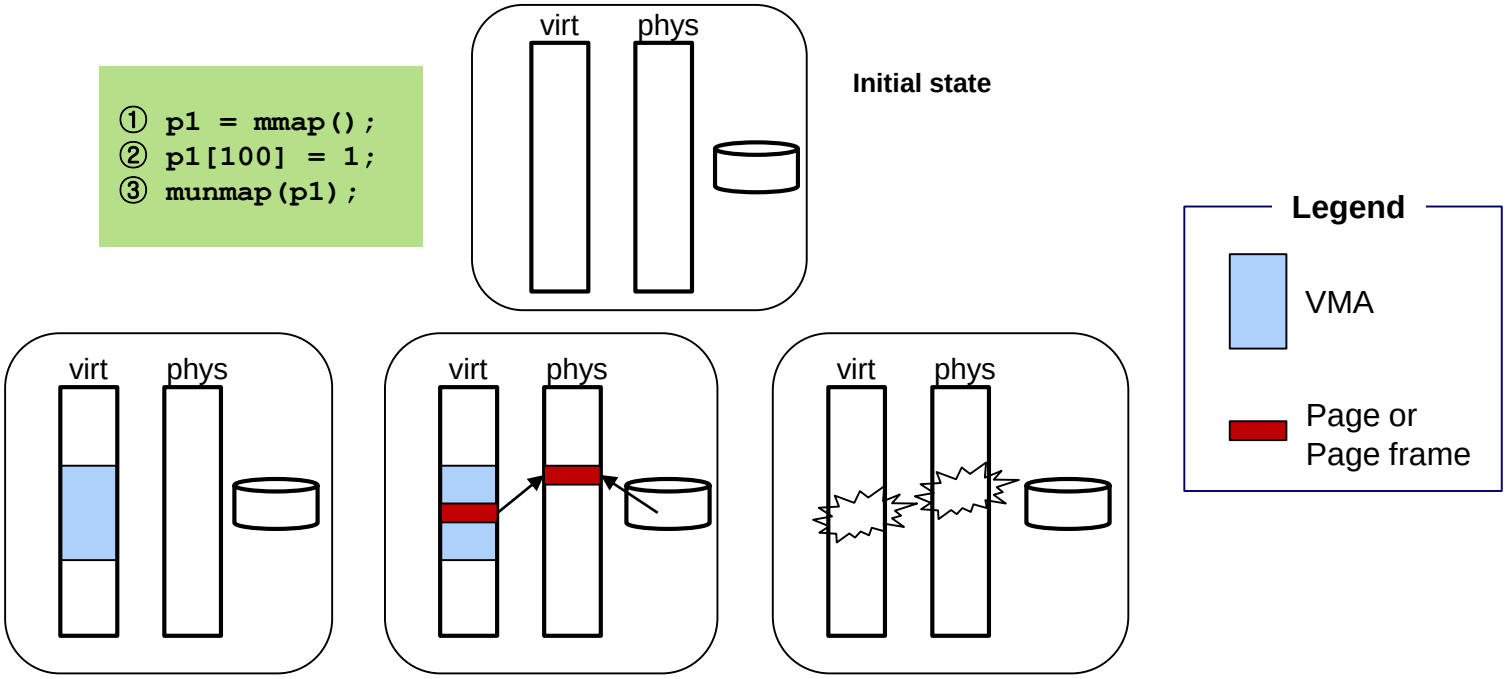
After ②
A minor page fault occurs and the page is mapped to a page frame

After ③
The VMA is released; the mapping is removed

mmap() System Call (2)

- Case2: `flags` does **NOT** include **`MAP_ANONYMOUS`**

```
① p1 = mmap();
② p1[100] = 1;
③ munmap(p1);
```



After ①
A VMA is allocated by mmap(); the corresponding pages are not mapped yet

After ②
A major page fault occurs and the page is mapped to a page frame and the handler reads data from disk

After ③
The VMA is released; the mapping is removed

Paging for Large Address Spaces

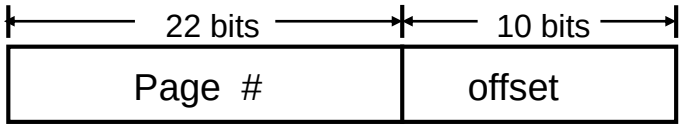
- Three approaches
  1. Hierarchical page table
  2. Hashed page table
  3. Inverted page table

1. Hierarchical Page Table (1)

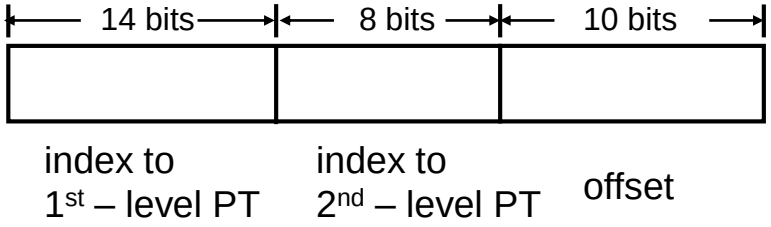
- Key ideas
  - Break up the logical address space into multiple page tables
  - A simple technique is a two-level page table
    - Logical address (on 32-bit machine with 1K page size) is divided into
      - Page number consisting of 22 bits
      - Page offset consisting of 10 bits
    - Since page table is paged, page number is further divided into
      - 14-bit first-level index
      - 8-bit second-level index

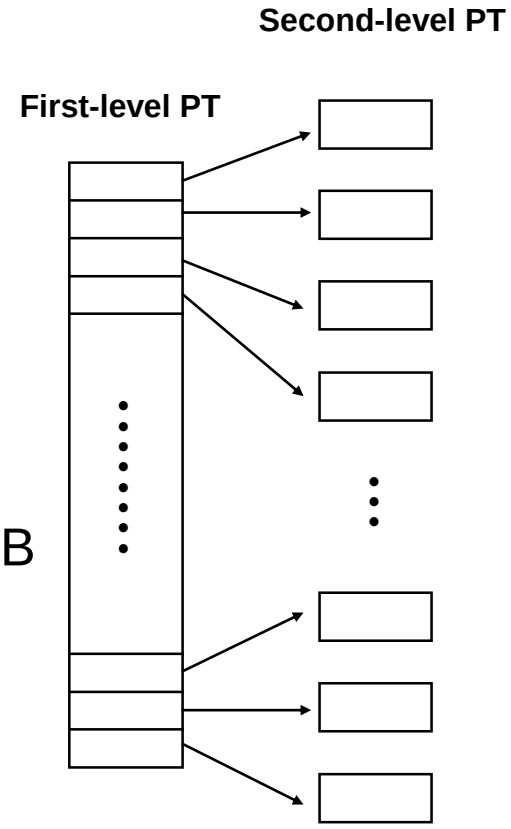
1. Hierarchical Page Table (2): Two-Level Page Table

- Address format



  - Size of PTE: 4 bytes
  - Size of PT: 2^{22} x 4 bytes = 16 MB
  - No of PTEs in a page: 256 entries (8-bit)
  - No of pages in PT: 16K (14-bit)
  - Size of first-level PT: 16K x 4 bytes = 64 KB

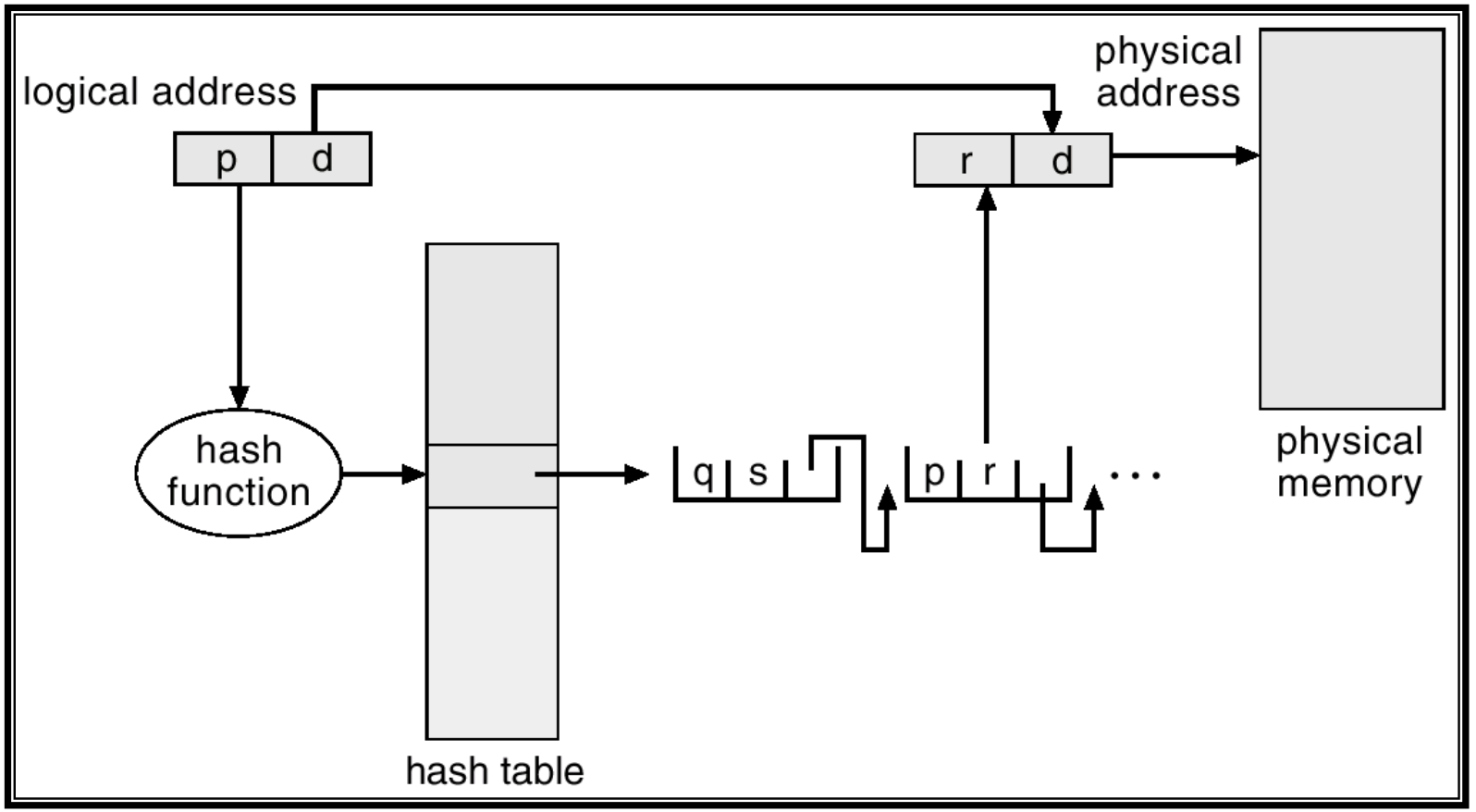




2. Hashed Page Table (1)

- Key ideas
  - The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
  - Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

2. Hashed Page Table (2)

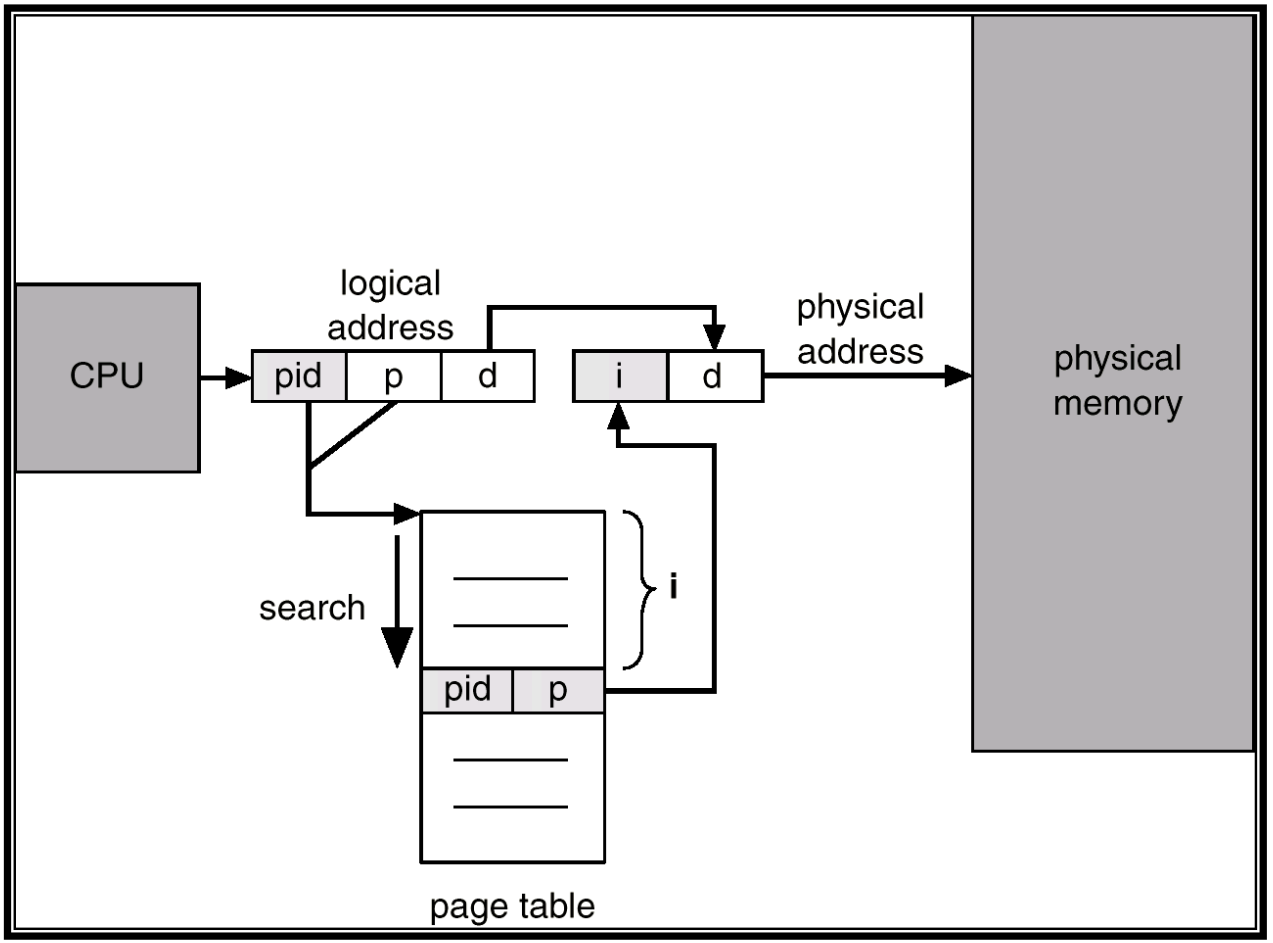


Source: Silberschatz, Galvin and Gagne, Operating System Concepts, 2002

3. Inverted Page Table (1)

- Key ideas
  - Best thought of as an off-chip extension of the TLB
  - One entry for each page frame of physical memory
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
  - Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
  - Uses hash table to limit the search to one, or at most a few page-table entries
  - Each process still keeps its page table
    - Disk address of an invalid page

3. Inverted Page Table (2)



Source: Silberschatz, Galvin and Gagne, Operating System Concepts, 2002