Translate Virtual to Physical Address

- Split virtual address (from CPU) into two pieces
  - Page number ($p$)
  - Page offset ($d$)
- Page number
  - Index into an entry of the page table, with addresses of physical pages
- Page offset
  - Position inside a page
- Page size = $2^d$ bytes: determined by offset size

An Example of Virtual/Physical Addresses

- Example:
  - 64 KB virtual memory
  - 32 KB physical memory
  - 4 KB page/frame size → 12 bits as offset ($d$)

Page Table Entry (PTE)

- Each entry in the page table contains
  - Frame number: bits depends on # of frames in physical memory
  - Valid bit: set if this logical page number has a corresponding physical frame in memory
  - Referenced bit: set if data on the page has been accessed
  - Dirty (modified) bit: set if data on the page has been modified
  - Protection information
- Size of each PTE: at least frame number plus 2/3 bits

More on Page Table

- Different processes have different page tables
  - CR3 points to the page table
  - Change CR3 registers when context switches
- Page table resides in main (physical) memory
  - Continuous memory segment
  - Why?
Outline

- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
  - Translation looksaside buffer (TLB)
- Multi-level page table
- Track free memory: bitmaps or linked list
- Page replacement algorithms and modeling
- Working set of processes
- Other implementation issues
  - Dirty page and locking page etc.

Two-Level Page Tables

- Why it is good?
- We don’t have to allocate all levels initially
- They don’t have to continuous

Page Table Size

- Modern Systems/Applications
  - 32 bits virtual address
  - System with 1GB physical memory → 30 bits physical address
  - Suppose the size of one page/frame is 4KB (12 bits)

- Page table size
  - # of virtual pages: 32 – 12 = 20 bits → 20 PTEs
  - Page table size = PTE size * 20 = 4 MB per process → 20 frames

- If there are 128 processes
  - Page tables occupy 128 * 4MB = 512 MB
  - 50% of memory will be used by page tables?

How can we get smaller page table?!
Memory Requirement of Page Tables

- Only the 1st level page table and the required 2nd level page tables need to be in memory.

- Example: a process access 32 MB (recall 32 bits virtual address, using 10-10-12), what is the minimum and maximum memory for the page table.

Example: a process access 32 MB (recall 32 bits virtual address, using 10-10-12), what is the minimum memory for the page table?

- 4KB/page process has 8K (8*2^10) virtual pages.
- One 2nd level page table maps 2^10 pages.
- Number (minimum) of 2nd level page table needed: 8 = 8*2^10/2^10 = 8
- Total (minimum) memory for page table: 1st level page table + 8; in total of 9 page tables → 9 X 4KB = 36 KB

Memory Requirement of Page Tables

- Only the 1st level page table and the required 2nd level page tables need to be in memory.

- Example: a process access 32 MB (recall 32 bits virtual address, using 10-10-12), what is the maximum memory for the page table?

- 4KB/page process has 8K (8*2^10) virtual pages.
- Thus, 8K pages could cover all of these entries.
- Total maximum memory for page table: 1st level page table + 1024 2nd level; → 1025 X 4KB

Quiz

- Why the page table has to be physically continuous?

- If we have two pages (0x00000000 and 0x00201000), what is the size of page table?

Page table size

- 32bit machine, page size 4k, each entry 4 bytes, one level page table (full 4GB linear address)
  
  Page table size = 2^20 pages = 2^22 = 4M

- 32bit machine, page size 4k, each entry 4 bytes, two level page table (two pages: 0x00000000, and 0xFFFFF000)

  Page table size = (2*10 level-0 entries) * 4bytes + (2*10 level-1 entries * 4 bytes) + 2 = 12 Kbytes

Linux’s 3 level page table

Linear Address converted to physical address using 3 levels

What is the benefit to use 3-level page table?
What is the shortcoming?
Size of Page/Frame: How Big?
- Determined by number of bits in offset (12Bit → 4KB)
- Smaller pages have advantages
  - Less internal fragmentation
  - Better fit for various data structures, code sections
- Larger pages are better because
  - Less overhead to keep track of them
  - More efficient to transfer larger pages to and from disk
- One principle: page table → fit into one frame

How can we make the address translation faster?

Fragmentation
- External Fragmentation
  - total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation
  - allocated memory larger than requested; this size difference is called internal partition.
- How can we reduce external fragmentation
  - Compaction: migrate memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time

Integrating VM and Cache
- Most caches “physically addressed”
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time else context switch = cache flush
  - Allows multiple processes to share pages
  - Cache doesn’t need to be concerned with protection issues
  - Access rights checked as part of address translation
- Perform address translation before cache lookup
  - Could involve memory access itself (to get PTE)
  - So page table entries can also be cached

Translation Lookaside Buffer (TLB)
- Small Hardware: fast
- Store recent accessed mapping of page → frame (64 ~ 1024 entries)
- If desired logical page number is found, get frame number from TLB
  - If not,
    - Get frame number from page table in memory
    - Use standard cache techniques
    - Replace an entry in the TLB with the logical & physical page numbers from this reference
    - Contains complete page table entries for small number of pages

Address Translation with TLB

What happens when cpu performs a context switch?
Memory Accesses Time

- Assuming:
  - TLB lookup time = a
  - Memory access time = m
- Hit ratio (h) is percentage of time that a logical page number is found in the TLB
  - More TLB entries usually means higher h
- Effective Access Time (EAT) is calculated (don’t include cache effect)
  - EAT = (m + a)h + (m + m + a)(1-h) = a + (2-h)m
- Interpretation
  - Reference always requires TLB lookup, 1 memory access
  - TLB misses also require an additional memory reference

Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated,
  - v ⇒ in-memory,
  - i ⇒ not-in-memory
- Initially bit is set to i on all entries
- During address translation, if valid–invalid bit in page table entry is i ⇒ page fault (trap)

Page Fault

1. Reference to a page.
   - If Invalid reference ⇒ abort
2. If not in memory, page fault occurs (trap to OS)
3. OS allocates an empty frame
4. Swap page into frame
5. Reset page tables, set validation bit = v
6. Restart the instruction that caused the page fault

Performance of Demand Paging

- Page Fault Rate 0 ≤ p ≤ 1.0
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)
  - EAT = (1 – p) x memory_access + p x page_fault_time
- page_fault_time depends on several factors
  - Save user reg and proc state, check page ref, read from the disk there might be a queue, (CPU can be given to another proc), get interrupt, save other user reg and proc state, correct the page table, put this process into ready queue.... Due to queues, the page_fault_time is a random variable
**Demand Paging Example**

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- \[ EAT = (1 - p) \times 200 + p \times 8,000,000 \]
  \[ = 200 + p \times 7,999,800 \]
- If one out of 1,000 access causes a page fault, then EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- If we want just 10% performance degradation, then \( p < 0.0000025 \), i.e., 1 page fault out of 400,000 accesses

**THRASHING**

**Thrashing**

- If a process does not have “enough” pages, the page-fault rate is very high.
- E.g. a process needs 6 pages, but only have 5 frames. Thus it will evict a page from existing 5 pages. Frequent faults

This leads to:
- low CPU utilization
- OS increase the degree of multiprogramming
- another process added to the system, worse case

**Locality and Thrashing**

- To prevent thrashing we should give enough frames to each process
- But how much is “enough”

**Locality model**
- Process migrates from one locality to another
- Localities may overlap
- When \( \Sigma \text{size of locality} > \text{total memory size} \)

Thrashing occurs...

**Working-Set Model**

- 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 is to use the recent needs of a process to predict its future needs.

**Working-Set Definition**

- Choose \( \Delta \), the working set parameter. At any given time, all pages referenced by a process in its last \( \Delta \) seconds of execution are considered to comprise its working set
- Pages outside the working set may be discarded at any time.
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
  - Timer interrupts after every 5000 time units
  - Keep reference bit and in-memory bit for each page
  - At a timer interrupt, copy and set the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set

Why is this not completely accurate?
- Improve = 10 bits and interrupt every 1000 time units

Page Replacement

- No free frame in memory, a page needs to be replaced.
- Pages that are replaced might be needed again later.
- We need algorithms to minimize the number of page faults.
- Include other improvements, e.g., use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk

Basic Page Replacement

- Find the desired page on disk
- If there is a free frame, use it
- If there is no free frame, use a page replacement algorithm
  1. Select a victim frame, swap it out (use dirty bit to swap out only modified frames)
  2. Bring the desired page into the (newly) free frame;
  3. Update the page and frame tables
- Restart the process

Page Replacement Algorithms

- How to select the victim frame?
  - You can select any frame, the page replacement will work; but the performance???
  - Gives the lowest page-fault rate
- Evaluate an algorithm by running it on a particular string of memory references (reference string) and compute the number of page faults on that string
  - In all our examples, we will have 3 frames and the following reference string
    \[
    7 \quad 0 \quad 1 \quad 2 \quad 0 \quad 3 \quad 0 \quad 4 \quad 2 \quad 3 \quad 0 \quad 3 \quad 2 \quad 1 \quad 2 \quad 0 \quad 1 \quad 7 \quad 0 \quad 1
    \]

First-In-First-Out (FIFO) Algorithm

- Maintain an FIFO buffer
  - + The page used before may not be needed
  - - An array used early, might be used again and again
- Easy to implement
- Belady’s Anomaly: more frames $\Rightarrow$ more page faults

Reference string

\[
\begin{array}{cccccccccccccccc}
7 & 7 & 0 & 1 & 2 & 0 & 3 & 0 & 4 & 2 & 3 & 0 & 3 & 2 & 1 & 2 & 0 & 1 & 7 & 0 & 1
\end{array}
\]
FIFO Illustrating Belady’s Anomaly

Reference string (12 accesses)
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

Replace a page that will not be used for longest time
reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

How do you know the future?

Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

Use recent past as an approximation of the future
Select the page that is not used for a long time...

OPT if you look at from backward
NO Belady’s Anomaly: so more frames ⇒ less page faults

Reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Given the reference string of page accesses: 1 2 3 4 2 3 4 1 2 1
1 3 1 4 and a system with three page frames, what is the final
configuration of the three frames after the true LRU algorithm is
applied?

Problem of LRU:
How to implement it efficiently?

Full LRU needs to sort all time of reference

LRU Algorithm (Cont.)

Counter (logical clock) implementation
Increase the counter every time a page is referenced
Save it into time-of-use field
Find one with the smallest time-of-use value
Problems: Counter overflow and linear search overhead

Stack implementation – keep a stack of page numbers
in a double link form:
(1) Not good for the parallelism (lock)
(2) Hardware support

Stack implementation – keep a stack of page numbers
in a double link form:
(1) Not good for the parallelism (lock)
(2) Hardware support

LRU Approximation Algorithms

Reference bit
With each page associate a reference bit, initially = 0
When page is referenced, set this bit to 1 by hardware
Replace the one which is 0 (if one exists)
We do not know the order, however
Additional bits can help to gain more ordering information

Second chance Alg (uses one reference bit)
FIFO with an inspection of ref bit
If ref bit is 0,
replace that page
set ref bit to 1
If ref bit is 1, “give a second chance”
set ref bit to 0
leave page in memory
Arrival time set to current time
go to next one
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - High priority processes can take all frames from low priority ones (cause thrashing)
  - A process cannot control its page fault rate
- **Local replacement** – each process selects from only its own set of allocated frames
  - Consistent performance
  - Lower utilization of memory and less throughput

Summary: Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out useful pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but hard to implement exactly</td>
</tr>
<tr>
<td>OPT (Optimal)</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
</tbody>
</table>

Kernel Memory Allocation

- E.g., Linux PCB (struct task_struct)
  - > 1.7 Kbytes each
  - Created on every fork and every thread create (e.g., `clone()`)
  - Deleted on every exit
- **Kernel memory allocators**
  - Buddy system
  - Slab allocation

ALLOCATING KERNEL MEMORY

Treated differently from user memory (allocate 1 page even when 1 byte is needed)
- Often allocated from a different free-memory pool
- Kernel requests memory for structures of varying sizes
- Some kernel memory needs to be contiguous

Buddy System (Dividing)

Two continuous blocks with the same size; the first one will start as $2^n$
**Page Allocation**

- **Example:** Need to allocate 65 contiguous page frames.
  - Look in list of free 128-page-frame blocks.
  - If free block exists, allocate it, else look in next highest order list (here, 256-page-frame blocks).
  - If first free block is in 256-page-frame list, allocate a 128-page-frame block and put remaining 128-page-frame block in lower order list.
  - If first free block is in 512-page-frame list, allocate a 128-page-frame block and split remaining 384 page frames into 2 blocks of 256 and 128 page frames. These blocks are allocated to the corresponding free lists.

**Buddy De-Allocation**

- When blocks of page frames are released the kernel tries to merge pairs of “buddy” blocks of size $b$ into blocks of size $2b$.
- Two blocks are buddies if:
  - They have equal size $b$.
  - They are located at contiguous physical addresses.
  - The address of the first page frame in the first block is aligned on a multiple of $2b\times2^{12}$ (starting at $2b$ page)
- The process repeats by attempting to merge buddies of size $2b$, $4b$, $8b$ etc...

**Slab Allocator**

- Performs the following functions:
  - Allocate memory
  - Initialize objects/structures
  - Use objects/structures
  - Deconstruct objects/structures
  - Free memory
- `/proc/slabinfo` – gives full information about memory usage on the slab level. (see also `/usr/bin/slabtop`)

**Slab** is one or more physically contiguous pages

**Cache** consists of one or more slabs

- Single cache for each unique kernel data structure (process descriptions, file objects, semaphores)
- Each cache filled with objects – instantiations of the data structure
- When cache created, filled with objects marked as free
- When structures stored, objects marked as used
- If slab is full, next object is allocated from empty slab. If no empty slab, new slab allocated

**Benefits include**
- no fragmentation,
- memory request is satisfied quickly
WHAT HAPPENS WHEN ALLOCATING MEMORY

Memory allocation (using mmap/brk)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int * ptr = malloc(4);
    *ptr = 1;
    free(ptr);
}
```

Memory Mapping (mmap or brk)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int * ptr = malloc(4);
    *ptr = 1;
    free(ptr);
}
```

What we learn here?

- Typically, the user will ask one big block of memory and setup its page table.
- Then this memory will be managed by user space memory manager.
  - How to manage the memory inside user space?

Currently, no heap space at all because we didn’t use any heap.
Summary

- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
- Translation lookaside buffer (TLB)
- Multi-level page table
- Page replacement algorithms
- Working set of processes
- Kernel Memory Management