Agenda

- Review
- Address Translation
- Caching and Virtual Memory
Virtualizing Resources

**Physical Reality:** different processes/threads share the same hardware

→ Need to multiplex
  - CPU (temporal)
  - Memory (spatial)
  - Disk and devices (later)

Why worry about memory sharing?
  - Complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Don’t want different threads to have access to each other’s memory (protection)
Single vs. Multithreaded Processes

- Threads encapsulate concurrency
- Address spaces encapsulate protection
  - Keep buggy program from trashing the system
Aspects of Memory Multiplexing

Isolation
• Don’t want separate state of processes colliding in physical memory (unexpected overlap → chaos)

Sharing
• Do want option to overlap when desired (for communication)

Virtualization
• Create illusion of more resources than exist in underlying physical system
Choose addresses for instructions & data from standpoint of the processor

```assembly
data1:   dw 32
          ...
start:   lw r1, 0(data1)
         jal checkit
          ...
loop:    addi r1, r1, -1
         bnz r1, r0, loop
          ...
checkit: ...
```

Could we place `data1`, `start`, and/or `checkit` at different addresses?

- Yes
- When? Compile time/Load time/Execution time
Program → Execution

• Phases of Preparation
  – Compile time (gcc)
  – Link/Load time (unix “ld” does link)
  – Execution time (dynamic libs)

• Addresses bound to final values throughout
  – depends on hardware & OS

• Dynamic Libraries
  – Linking postponed until execution
  – Small piece of code (*stub*) used to locate the appropriate memory-resident library routine
  – OS checks if routine is in processes’ memory address
  – Stub replaces itself with the address of the routine, and executes routine
Dynamic Loading

• Routine not loaded until called
• Better memory-space utilization
  • Unused routine never loaded
  • Useful when large amounts of code handle infrequent cases (error handling)
• No special support from the OS needed
Uniprogramming

No Translation or Protection

Application:

- Always runs at same place in physical memory since only one application at a time
- Can access any physical address
- Given illusion of dedicated machine by giving it reality of a dedicated machine

0xFFFFFFFF

Valid 32-bit Addresses

Operating System

Application

0x00000000
Multiprogramming, v1

- No Translation
  - Loader/Linker adjusts addresses (loads, stores, jumps) while program loaded into memory
    - Everything adjusted to memory location of program
    - “Translation” done by linker-loader
    - Pretty common in early days

- No protection
  - Bugs in any program can crash other programs (or OS!)

```
0x00000000
Operating System
Application2
0x00000000
Application1
0xFFFFFFFF
```
Multiprogramming, v1++

Add Protection:

- Two special registers (base and limit) prevent user from straying outside designated area
  - User tries to access an illegal address → error

- During switch, kernel loads new base/limit from PCB
  - User not allowed to change base/limit registers

```
0x00000000
0xFFFFFFFF
```

```
0x00020000
Base=0x20000
Limit=0x10000
0xFFFFF
```

```
0x00000000
```

```
Application1
Application2
Operating System
```
Base and Limit Registers

- **Base** and **Limit** registers define logical address space
Multiprogramming, v2

• **Goals:**
  – Protection: keep multiple applications from each other
  – Isolation: keep processes and kernel from one another
  – Flexibility: translation that
    • Avoids fragmentation
    • Allows easy sharing between processes
    • Allows only part of process to be resident in physical memory

• **Required Hardware Mechanisms:**
  – General Address Translation
    • Flexible: Can fit physical chunks of memory into arbitrary places in users address space
    • Not limited to small number of segments
    • *Think:* providing a large number (thousands) of fixed-sized segments (called “pages”)
  – Dual Mode Operation
    • Protection base involving kernel/user distinction
Memory Hierarchy

Memory Protection required for correct operation

Program must be brought (from disk) into memory and placed within a process to be run
Agenda

• Review
• Address Translation
  • Concept
  • Flexible Address Translation
  • Efficient Address Translation
  • Memory Protection
• Caching and Virtual Memory
Address Translation

• Mapping virtual $\rightarrow$ physical address
• User program deals with *virtual* (or *logical*) addresses, never sees (real) *physical* addresses

• Performed by Memory-Management Unit (MMU)
  • Hardware device
  • Many possible translation methods
Simple Address Translation: using a relocation register

**Dynamic Relocation:** value in relocation register added to every address generated by a user process when sent to memory

![Diagram showing CPU, logical address, relocation register, MMU, and memory. Logical address 346 is added to the relocation register value 14000, resulting in physical address 14346.](image)
Contiguous Allocation (1)

- Main memory usually into two partitions:
  - Resident OS, usually held in low memory with interrupt vector
  - User processes then held in high memory

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - **Base register**: value of smallest physical address
  - **Limit register**: range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
Contiguous Allocation (2)

Multiple-partition allocation

- **Hole** = block of available memory; holes of various size scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
  a) allocated partitions
  b) free partitions (holes)
Dynamic Storage-Allocation Problem

- **First-fit**: Allocate *first* hole that is big enough
- **Best-fit**: Allocate *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate *largest* hole; must also search entire list
  - Produces the largest leftover hole
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Can we find a more flexible implementation?
Agenda

- Review
- Address Translation
  - Concept
  - Flexible Address Translation
  - Efficient Address Translation
  - Memory Protection
- Caching and Virtual Memory
Segments

• Note: overloaded term...
• **Chunks of virtual address space**
• Access Protection
  – User/Supervisor
  – Read/Write/Execute
• Sharing
  – Code, libraries
  – Shared memory for IPC
• Virtualization
  – Illusion of more memory than there really is
Segment examples

- **Code**
  - Execute-only, shared among all processes that execute the same code
- **Private Data**
  - R/W, private to a single process
- **Heap**
  - R/W, Explicit allocation, zero-initialized, private
- **Stack**
  - R/W, Implicit allocation, zero-initialized, private
- **Shared Memory**
  - explicit allocation, shared among processes, some read-only, others R/W
Paging: a Conceptual Overview

• Divide physical memory into **frames**:  
  • also called a “page frame”  
  • fixed-sized blocks  
  • size is power of 2, (512 bytes up to 8192 bytes)  
• Divide logical memory into **pages**:  
  • blocks of memory, same size as the frames  
• **Page table** translates logical \( \rightarrow \) physical addresses  
  “page 10 can be found in frame 20”
Paging: a Logical View

• To run a program of size $n$ pages, need to find $n$ free frames and load program.

• Note: physical address space of a process can be noncontiguous
Address Translation Scheme

Address generated by CPU is divided into:

- **Page number** \((p)\) – used as an index into a *page table* which contains base address of each page in physical memory

- **Page offset** \((d)\) – combined with base address to define the physical memory address that is sent to the memory unit

\[
\begin{array}{|c|c|}
\hline
\text{page number} & \text{page offset} \\
\hline
p & d \\
\hline
m - n & n \\
\hline
\end{array}
\]

(Given logical address space \(2^m\) and page size \(2^n\))
struct {
    int frame;
    bit is_valid, is_dirty, ...;
} PTE;

struct PTE page_table[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
    if (page_table[vpn].is_valid)
        return page_table[vpn].frame;
    else...
}
Paging Example

How big is a virtual address?

Which bits are page number?

Which bits are page offset?

How big is a physical address?

32-byte memory
4-byte frames

Page table

logical memory

physical memory
Free Frames

Before allocation

After allocation
Implementation of Page Table

- Page table can be kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of page table
- Every data/instruction access requires 2 memory accesses. One for the page table and one for the data/instruction. *(more later)*
- Software or Hardware maintained? For portability, most kernels maintain their own page tables. Must be translated into MMU tables.
Page Table Size

How big is a page table on the following machine?

**Given:** 32-bit machine, 4KB per page, each PT entry = 4B

- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?
- Page tables can get *big*

- **Many solutions:** Hierarchical Page Table, Hashed Page Tables, Inverted Page Tables
Hierarchical Page Tables

• Break up logical address space into multiple page tables
• For example: two-level page table
Two-Level Paging Example

• A logical address (on 32-bit machine with 1K page size) is divided into:
  – a page offset of 10 bits ($1024 = 2^{10}$)
  – a page number of 22 bits (32-10)
• Since the page table is paged, the page number is further divided into:
  – a 12-bit page number
  – a 10-bit page offset
• Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
Address-Translation Scheme
Hashed Page Tables

- Common in address spaces > 32 bits \( (why?) \)
- Virtual Page Num hashed into page table, which contains chain of elements hashing to same location
- Virtual page numbers compared in chain, searching for a match. Found \( \rightarrow \) corresponding physical frame extracted.

![Diagram](image)
Inverted Page Table

1 entry per real page of memory:
- virtual address of page stored in that real memory location + info about process that owns that page

↓ memory to store page tables
↑ time to search page tables
→ hash table limits search to one — at most a few — page-table entries
Agenda

• Review
• Address Translation
  • Concept
  • Flexible Address Translation
  • Efficient Address Translation
  • Memory Protection
• Caching and Virtual Memory
Translation look-aside buffers (TLBs)

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache (an associative memory)
- Allows parallel search of all entries.
- Address translation (p, d)
  - If p is in TLB get frame # out (quick!)
  - Otherwise get frame # from page table in memory
    - And replace an existing entry
    - But which? (stay tuned)
  - Page table lookup can be either S/W or H/W
Paging Hardware With TLB
Updated Context Switch

- Save current process’ registers in PCB
- Set up Page Table Base Register (PTBR)
  - This info is kept in the PCB
- **Flush TLB**
- Restore registers of next process to run
- “Return from Interrupt”
Agenda

- Review
- Address Translation
  - Concept
  - Flexible Address Translation
  - Efficient Address Translation
  - Memory Protection
- Caching and Virtual Memory
Memory Protection

- Associate protection bits with each page
- MMU enforces protection
  - Throws exceptions on illegal accesses
  - Often also tracks R/W/X accesses
Arch-dependent protection bits

Multiple possibilities, incl:

- Valid/Invalid bit + Writable/Read-only bit
  (no encoding for execute protection)
  *Valid Bit is also known as Present Bit*

- R/W/X bits
  (all off == invalid)
Shared Pages

• PT entries of multiple processes pointing to the same frame
  • “shared frames” would have been a better term

• Examples of Shared Pages
  – Execute-only code (i.e., text editors, window systems)
    – Shared code (typically) must appear in same location in the logical address space of all processes
    – Particularly useful for libraries
  – Read-only data (i.e., strings)
  – Read-write shared data

• Example of Private Pages
  – Read-write private data and stack
Shared Pages Example
(Virtual) Null Page

• Shared page, but made invalid to all
  – Why?
Copy-on-Write Segments

- Useful for “fork()” and for initialized data
- Initially map page read-only
- Upon page fault:
  - Allocate a new frame
  - Copy frame
  - Map new page R/W
  - If fork(), map “other” page R/W as well
Agenda

- Review
- Address Translation
  - Concept
  - Flexible Address Translation
  - Efficient Address Translation
  - Software Protection
- Caching and Virtual Memory
Warning: Page vs Frame...

- Page: virtual
- Frame: physical

*Often used interchangeably, unfortunately*
Before Paging: “Swapping”

• Originally, a way to free frames by copying the memory of an entire process to “swap space”
  – Swap out, swap in a process...
• This technique is not so widely used any more
• “Swapping” now sometimes used as synonymous with “paging”
Swapping

- A process can be swapped temporarily out of memory to a backing store

- Major part of swap time is transfer time; total transfer time is proportional to the amount of memory
Swapping vs Paging

• **Swapping**
  – Loads entire process in memory, runs it, exit
  – Is slow (for big, long-lived processes)
  – Wasteful (might not require everything)

• **Paging**
  – Runs all processes concurrently, taking only pieces of memory (specifically, pages) away from each process
  – Finer granularity, higher performance
  – Paging completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

• The verb “to swap” is also used to refer to pushing contents of a page out to disk in order to bring other content from disk; this is distinct from the noun “swapping”
OS and Paging

• Process Creation
  – Allocate space and initialize page table for program and data
  – Allocate and initialize swap area
  – Info about PT and “swap space” is recorded in process table

• Process Execution
  – Reset MMU for new process
  – Flush the TLB

• Page Faults
  – Bring processes’ pages in memory

• Process Termination
  – Release pages
Handling a Page Fault

- Identify page in which fault occurred and reason (r/w/x)
- If access inconsistent with segment access rights, terminate process
- If r/x access within code or read/only data segment:
  - Check to see if a frame with the code or data already exists
  - If not, allocate a frame and read content from executable file
    - If disk access required, another process can run in the mean time
  - Map page for R/X only
  - Return from interrupt
- If access within non-zero initialized data segment:
  - Check to see if a frame with the code or data already exists
  - If not, allocate a frame and read data from executable file
  - Map page for R/W access
  - Return from interrupt
- If access within zero-initialized data (BSS) or stack
  - Allocate a frame and fill page with zero bytes
  - Map page for R/W access
  - Return from interrupt
Steps in Handling a Page Fault

1. Trap
2. Page is on backing store
3. Bring in missing page
4. Physical memory
5. Free frame
6. Restart instruction

Load M
Reference
Page table
Operating system
Pre-fetching

- Disk/network overhead of fetching pages is relatively very high
- If a process accesses page X in a segment, the process is likely to access page X+1 as well
- Pre-fetch: start fetch even before page fault has occurred
Page Replacement

• What happens if there is no free frame to allocate?
  – Select a frame and deallocate it
    • The frame to eject is selected using the *Page Replacement/Eviction Algorithm*
  – Unmap any pages that map to this frame
    • May involve multiple processes’ page tables
  – If the frame is “dirty” (modified), save it on disk so it can be restored later if needed
    • Upon subsequent page fault, load the frame from where it was stored

• Goal: Select frame that minimizes future page faults
• Note: strong resemblance to caching algorithms
• Also reminiscent of scheduling algorithms
Page Replacement

- **Frame and Valid-Invalid Bit**: There is a frame in the page table with a valid-invalid bit. If a page is marked invalid (0), it needs to be swapped out. If it is marked valid (1), it can be swapped in.

- **Page Table**: The page table contains information about pages in memory.

- **Physical Memory**: The physical memory where pages are located.

- **Swap Operations**:
  1. Swap out a victim page from memory.
  2. Change the valid bit to invalid for the page.
  3. Swap in the desired new page.
  4. Reset the page table for the new page.
Modified/Dirty Bits

• Use hardware **modified (or dirty) bit** to reduce overhead of page transfers:
  – modified pages are written to disk
  – non-modified pages brought back from original source
    • Example: text segments are rarely modified, bring pages back from the program image stored on disk
  – *Small conceptual problem*: dirty bit associated with page instead of frame

• If MMU does not support dirty bit, can simulate it in software by mapping a page “read-only” and mark it dirty upon first page fault
Page Replacement Algorithms

- **Random**: Pick any page to eject at random
  - Used mainly for comparison
- **FIFO**: The page brought in earliest is evicted
  - Ignores usage
- **OPT**: Belady’s algorithm
  - Select page not used for longest time
- **LRU**: Evict page that hasn’t been used the longest
  - Past could be a good predictor of the future
- **MRU**: Evict the most recently used page
- **LFU**: Evict least frequently used page
First-In-First-Out (FIFO) Algorithm

- **Reference string**: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **3 frames** (3 pages in memory at a time per process):
  
<table>
<thead>
<tr>
<th>frames</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

  ← contents of frames at time of reference

  9 page faults
**First-In-First-Out (FIFO) Algorithm**

- **Reference string:** 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **4 frames** (4 pages in memory at a time per process):

  - | frames | reference |
  - |-------|-----------|
  - | 1     | 1         |
  - | 2     | 2         |
  - | 3     | 1         |
  - | 4     | 1         |
  - | 1     | 1         |
  - | 2     | 2         |
  - | 1     | 1         |
  - | 4     | 5         |
  - | 3     | 1         |
  - | 4     | 5         |
  - | 1     | 1         |
  - | 5     | 1         |
  - | 1     | 1         |
  - | 5     | 1         |
  - | 2     | 1         |
  - | 3     | 1         |
  - | 2     | 1         |
  - | 1     | 1         |
  - | 4     | 5         |

  -> contents of frames at time of reference

  - 10 page faults

  - more frames → more page faults?

  **Belady’s Anomaly**
FIFO Illustrating Belady’s Anomaly
Optimal Algorithm (OPT)

• Replace page that will not be used for the longest
• 4 frames example

6 page faults

Question: How do we tell the future?
Answer: We can’t

OPT used as upper-bound in measuring how well your algorithm performs
OPT Approximation

• In real life, we do not have access to the future page request stream of a program
  – No crystal ball
  – no way to know which pages a program will access

→ Need to make a best guess at which pages will not be used for the longest time
Least Recently Used (LRU) Algorithm

• Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

8 page faults
Implementing* Perfect LRU

• On reference: Timestamp each page
• On eviction: Scan for oldest frame
• Problems:
  – Large page lists
  – Timestamps are costly
• Solution: approximate LRU

Q: “I thought LRU was already an approximation...”
A: “It is... Oh well...”

* the blue shading in the previous frame diagram
Approx. LRU: Clock Algorithm aka Second-Chance Algorithm

• Each page has a reference bit
  – Set on use, reset periodically by the OS
  – If no H/W, can be emulated in S/W

• Algorithm:
  – FIFO + reference bit (keep pages in circular list)
    • Scan: if ref bit is 1, set to 0, and proceed. If ref bit is 0, stop and evict.
  – Implements “Not-Recently-Used”

• Problems:
  – Low accuracy for large memory
    • “Recent” depends on size of memory
  – When to run
    • Periodically or upon page fault
LRU with large memory

- Solution: Add another hand
  - Trailing edge clears ref bits
  - Trailing edge evicts pages with ref bit 0

- What if angle small?
- What if angle big?

- Sensitive to sweeping interval and angle
  - Fast: lose usage information
  - Slow: all pages look used
Other Algorithms

• MRU: Remove the most recently touched page
  – Works well for data accessed only once, e.g. a movie file
  – Not a good fit for most other data, e.g. frequently accessed items

• LFU: Remove page with lowest usage count
  – No record of when the page was referenced
  – Use multiple bits. Shift right by 1 at regular intervals.

• MFU: remove the most frequently used page

• LFU and MFU do not approximate OPT well
Complete Page Table Entry (PTE) ...

<table>
<thead>
<tr>
<th>Valid</th>
<th>Protection R/W/X</th>
<th>Ref</th>
<th>Dirty</th>
<th>Index</th>
</tr>
</thead>
</table>

*Index* is an index into
- table of memory frames (if bottom level)
- table of page table frames (if multilevel page table)
- backing store (if page is not valid)

**Synonyms:**
- Valid bit == Present bit
- Dirty bit == Modified bit
- Referenced bit == Accessed bit
Where is the page?

(the content of) a virtual page can be

- mapped
  - to a physical frame
- not mapped:
  - in a physical frame, but not currently mapped
  - still in the original program file
  - zero-filled (heap/BSS, stack)
  - on backing store ("paged or swapped out")
  - illegal: not part of a segment
Thrashing

• **Thrashing** = excessive rate of paging
  – May stem from lack of resources
  – Or caused by bad or badly matched eviction algorithm...
    • Keep throwing out page that will be referenced soon
      ⇒ Keeps accessing memory that is not there

• Why does it occur?
  – Poor locality, past $\neq$ future
  – There is reuse, but process does not fit model
  – Too many processes in the system
Global vs. Local Replacement

• **Global replacement**
  – Single memory pool for entire system
  – On page fault, evict oldest page in the system
  **Problem:** lack of performance isolation

• **Local (per-process) replacement**
  – Have a separate pool of pages for each process
  – Page fault in one process can only replace pages from its own process
  **Problem:** might have idle resources
Page Fault Frequency

- Thrashing viewed as poor ratio of fetch to work
- **PFF** = page faults / instructions executed
  - PFF above threshold $\rightarrow$ process needs more memory
    - not enough memory on the system $\rightarrow$ Swap out
  - PFF below threshold $\rightarrow$ memory can be taken away
Working Set

Original definition:
“collection of [a process’] most recently used pages”

*The Working Set Model for Program Behavior,*
Peter J. Denning, 1968

Formal definition:
pages referenced by process in last $\Delta$ time-units
Working Sets

- **Working set size**: *num* pages in working set
  - *num* pages touched in the interval \((t-\Delta .. t]\).
- Working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - During that period, you have a larger working set size
- **Goal**: keep WS for each process in memory
  - If \(\Sigma |WS_i|\) for all *i runnable* processes > |physical memory| \(\rightarrow\) suspend a process
## Working Set Approximation

- Approximate with interval timer + reference bits
- Example: $\Delta = 10,000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - When timer interrupts: 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?
  - Cannot tell (within interval of 5000) where reference occurred
- Improvement: 10 bits and interrupt every 1000 time units

<table>
<thead>
<tr>
<th>page</th>
<th>Last 5000</th>
<th>Next 5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>