Thank Dr. Dakai Zhu, Dr. Palden Lama, and Dr. Tim Richards (UMASS) for providing their slides.

Outline
- 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 and modeling
- Working set of processes

Memory Hierarchy
- CPU can directly access main memory and registers only
- But programs and data must be brought (from disk) into memory
- Memory accesses can be the bottleneck
  - Cache between memory and CPU registers
- Memory Hierarchy
  - Cache: small, fast, expensive; SRAM
  - Main memory: medium-speed, not that expensive; DRAM
  - Disk: many gigabytes, slow, cheap, non-volatile storage

Memory
- The ideal memory is
  - Very large
  - Very fast
  - Non-volatile (doesn’t go away when power is turned off)
- The real memory is
  - Not very large
  - Not very fast
  - Affordable (cost)!
  - Pick any two...
- Memory management goal: make the real world look as much like the ideal world as possible
Limitations without virtual memory

Protection of memory: using base and limit registers

Many Questions:
1. How to generate the addresses for a process?
2. How to assign physical memory?

Swapping

Consider a multi-programming environment:
- Each program must be in the memory to be executed
- Processes come into memory and
- Leave memory when execution is completed

What if no free region is big enough?

Swapping (cont’d)

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- Backing store – large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Swapping can free up memory for additional processes

Swapping (cont’d)

- Major part of swap time is transfer time;
  - Total transfer time is directly proportional to the amount of memory swapped (e.g., 10MB process / 40MB per sec = 0.25 sec)
  - May take too much time to be used often
- Standard swapping requires too much swapping time
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows), but it is often disabled
Outline

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

Virtual Memory

- Basic idea: allow OS to allocate more memory than the real
- Program uses virtual addresses
  - Addresses local to the process
  - Can be any size limited by # of bits in address (32/64)
  - 32 bits: 4G
  - 64 bits:
- Virtual memory >> physical memory

Motivations for Virtual Memory

- Use physical DRAM as cache for the disk
  - Virtual pages of processes can exceed physical memory size
- Simplify memory management
  - Multiple processes resident in main memory
  - Each with its own address space
  - Only “active” code and data is actually in memory
- Provide protection
  - One process can't interfere with another
  - Because they operate in different address spaces
  - User process cannot access privileged information
  - Different sections of address spaces have different permissions

Virtual Memory for Multiprogramming

- Virtual memory (VM) is helpful in multiprogramming
  - Multiple processes in memory concurrently
  - Each process occupies small portion of memory
  - CPU schedules process B while process A waits for its long I/O operations (e.g., retrieve data from disks)
- Physical Memory de/allocation
  - Keep recently used content in physical memory
  - Move less recently used stuff to disk
  - Movement to/from disk handled by the OS

How to get physical address from the virtual one?!
Virtual and Physical Addresses

- Virtual address space
  - Determined by instruction width
  - Same for all processes

- Physical memory indexed by physical addresses
  - Limited by bus size (# of bits)
  - Amount of available memory

Paging: a memory-management scheme that permits address space of process to be non-continuous.

Paging and Page Systems

- Virtual address
  - Divided into pages

- Physical memory
  - Divided into frames

- Page vs. Frame
  - Same size address block
  - Unit of mapping/allocation

- A page is mapped to a frame
  - All addresses in the same virtual page are in the same physical frame
  - Offset in a page

- Physical memory
  - Virtual space

Page Table

- Each process has one page table
  - Map page number to physical page number
  - Number of PTEs in page table
  - Number of total pages in virtual space
  - Not just the pages in use, why?

- Page table is checked for every address translation
  - Where to store page table?

- Not all pages need map to frames at the same time

- Not all physical frame need be used

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)

- Virtual address: 16 bits
  - How many virtual pages?

- Frame #:3bits
  - Offset: 12 bits
  - physical address: 15 bits
  - How many physical frames?

- Page #:4bits
  - Offset: 12 bits

---

**Address Translation**

- Suppose the logical address space is $2^m$ and page size is $2^n$ so the number of pages is $2^m / 2^n$, which is $2^{m-n}$

- Logical Address ($m$ bits) 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

- Page number: $p$
- Page offset: $d$

---

**Address Translation Architecture**

- Address Translation:
  - Virtual address: 16 bits
  - How many virtual pages?

- Physical address: 15 bits
  - How many physical frames?

---

**Computing Physical Address**

1. Compute the page number
2. Check the page table, get physical frame
3. Compute the starting address of physical frame
4. Compute the physical address
Virtual address 0x44, offset 0x44
1. Compute the page number
Page number is #0
2. Check the page table, get physical frame
Frame number is #2
3. Compute the starting address of physical frame
Start Address = 2 * 0x80 = 0x100
4. Compute the physical address
Physical Address = 0x100 + 0x44 = 0x144

Virtual address 0x224, offset 0x24
1. Compute the page number
Page number is #4
2. Check the page table, get physical frame
Frame number is #3
3. Compute the starting address of physical frame
Start Address = 3 * 0x80 = 0x180
4. Compute the physical address
Physical Address = 0x180 + 0x24 = 0x1A4

Suppose the page size is 4-byte pages.
What would you say about the size of logical address space, the size of physical address space, and the size of the page table?
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
  - If not valid, remainder of PTE is irrelevant
- 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

Virtual address: FF:10 bits  Offset: 10 bits

Number of pages: 12 bits → 4K pages
Number of frames: 10 bits → 1K frames

Size of page table:
- Each page entry must have at least 10 bits → 2 bytes
- 4K * 2 bytes = 8KB → requires 8 consecutive frames

Valid (v) or Invalid (i) Bit In A Page Table

Size of A Page/Frame: Another Example

Example:
- Logical (Virtual) address: 22 bits → 4 MB
- Physical address (bus width): 20 → 1MB

Page/frame size: 1KB, requiring 10 bits

Virtual addr: FF:12 bits  Offset: 10 bits

Size of Page/Frame: How Big?

- Determined by number of bits in offset (512B → 16KB and become larger)
- Smaller pages
  - Less internal fragmentation
  - Too large page table: spin over more than one frame (need to be continuous due to index), hard to allocate!
- Larger pages
  - Too small page table so less overhead to keep track of them
  - More efficient to transfer larger pages to/from disk
  - More internal fragmentation; waste of memory

Design principle: fit page table into one frame
- If not, multi-level paging (discussed later)
Implementation of Page Table

Where should we store Page table?

- Registers: fast efficient but limited
- Main memory: dedicated lookup tables

Memory

- Page-table base register (PTBR) points to the page table in memory

Every data/instruction access requires two memory accesses

- One for the page table and one for the data/instruction

Translation Look-ahead Buffers (TLB)

- Translation Look-ahead buffers (TLB)
  - A special fast-lookup hardware cache: associative memory
  - Access by content → parallel search: if in TLB, get frame #
  - Otherwise, access the page table in memory, and get frame # and put it into TLB
    - (if TLB is full, replace an old entry. Wired down entries will not be removed)

- Continuous memory segment

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

Example TLB
Integrating VM and Cache

Most caches are "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

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 → 2^20 PTEs
  - Page table size = PTE size * 2^20 = 4 MB per process → 2^10 frames

- If there are 128 processes
  - Page tables occupy 128 * 4MB = 512 MB
  - 50% of memory will be used by page tables if a system has 1G memory?

How can we get smaller page table?!

Integrating TLB and Cache

“Translation Lookaside Buffer” (TLB)

Outline

- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
  - Translation lookaside 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.
Example: 2-level Address Translation

- **Page number**: 3 bits
- **Page offset**: 12 bits
- **Physical address**: 18 bits

Which tables should be in memory?

- 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 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: 
  \( s = 2^{10}/2^{10}/2^{10} = 8 \)
- Total (minimum) memory for page table: 1st level page table + 8; in total of 9 page tables = 9 X 4KB = 36 KB

**Two-Level Page Tables**

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

**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 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: 
  \( s = 2^{10}/2^{10}/2^{10} = 8 \)
- Total (minimum) memory for page table: 1st level page table + 8; in total of 9 page tables = 9 X 4KB = 36 KB

**Two-Level Page Tables**

- Solution: multi-level page tables
- Virtual address: three parts
  - Level-one page number (10 bits)
  - Level-two page number (10 bits)
  - Offset (12 bits)
- PTE in 1st level page table contains physical frame for one 2nd level page table
- 2nd level page table has actual physical frame numbers for the memory address

**Example: 2-level Address Translation**

- **Page number**: 10 bits
- **Page offset**: 12 bits
- **Physical address**: 18 bits

Which tables should be in memory?
Quiz: Memory Requirement

- 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
  - We will have at most 2^10 (defining by 10 bits of 1st level) pages/frames in 2nd level
  - Therefore, all of these frames at minimum can have one entry inside.
  - Total (maximum) memory for page table: 1st level page table + 1024 2nd level page table → 1025 X 4KB

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) * 4 bytes + (2^10 level-1 entries * 4 bytes) * 2 = 12 Kbytes

Other questions

- Why the page table has to be physically continuous?

- If we have two pages (0x00000000 and 0x00201000), what is the size of 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 maximum memory for the page table?
  - 4KB/page → process has 8K (8*2^10) virtual pages
  - 32MB = 8K pages, and the total number of 2nd level entries 2^10
  - 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
Linux's 3 level page table

Linear Address converted to physical address using 3 levels

- Index into Page Dir.
- Index into Page Middle Dir.
- Index into Page Table
- Page Offset

What is the benefit to use 3-level page table?
What is the shortcoming?

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

Designing of 2 level page table

1. Determine **number of bits of page offset** (12Bit→4KB)
2. Determine the number of bits for the page table entry
3. Determine the number of bits for 2nd level page table
   - One principle: page table → fit into one frame
4. Determine the number of bits for 1st level page table

Designing of Multi-level Page Table

- Suppose that a system has a **24-bit logical address** space and is byte-addressable. The amount of physical memory is **1MB** (i.e., the physical address has **20 bits**) and the size of a page/frame is **1K bytes**. How to design a two-level page table?
  1. The size of page table entry will be 2 bytes (larger than 13 bits)
  2. One page can hold 512 entries (1K/2=512)
  3. Thus, we need 8 bits for the 2nd level page table
Designing of Multi-level Page Table

Suppose that a system has a 24-bit logical address space and is byte-addressable. The amount of physical memory is 1MB (i.e., the physical address has 20 bits) and the size of a page/frame is 1K bytes. What about this design?

<table>
<thead>
<tr>
<th>4 bits</th>
<th>10 bits</th>
<th>10 bits</th>
</tr>
</thead>
</table>

Second level 10 bits, can’t be fitted into one page. Then we may need to have multiple pages (4 pages) that are continuous, which can’t guarantee or increase the complexity of OS design.

Designing of Multi-level Page Table (2)

Suppose that a system has a 16-bit logical address space and is byte-addressable. The amount of physical memory is 64KB (i.e., the physical address has 16 bits) and the size of a page/frame is 256 bytes. How to design a two-level page table?

1. The size of page table entry will be 2 bytes (larger than 11 bits)
2. One page can hold 256 entries (256/2 = 128)
3. Thus, we need 7 bits for the 2nd level page table

How Many Levels are Needed?

- New architectures: 64-bits address?
  - Suppose 4KB page/frame (12-bit offset)
  - Then we need a page table with \(2^{12}/2^{12} = 2^{12}\) entries!!!!
  - If we use two-level paging, then inner page table could be one page, containing 1K entries (10-bit) (assuming PTE size is 4 bytes)
  - So the outer page needs to contain 256 entries !!!
  - Similarly, we can page the outer page, giving us three-level paging
  - If we continue this way, we will have 6-level paging

- Problems with multiple-level page table
  - One additional memory access for each level added (if not in TLB)
  - Multiple levels are possible; but prohibitive after a few levels

Is there any other alternative to manage page/frame?

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 the percentage of time that a logical page number is found in the TLB
  - More TLB entries usually mean a 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

Effective Access Time: An Example

- TLB Lookup = 20 nanosecond
- Memory cycle time is 100 nanosecond
- Hit ratio = 80%
- Effective Access Time (EAT) would be
  \[ EAT = (20+100) \times 0.8 + (20+100+100) \times (1-0.8) \]
  \[ = 140 \text{ nanosecond} \]
- 40% slow down
- What about 98% hit rate? What is the EAT?

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 \( \Rightarrow \) reference to it
  - Invalid reference \( \Rightarrow \) abort
  - Not-in-memory \( \Rightarrow \) bring to memory

DEMAND PAGING
Valid-Invalid Bit

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

Page Fault

1. Reference to a page, 
   If invalid reference \( \implies \text{abort} \)
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. \textbf{Restart the instruction} that caused the page fault

Performance of Demand Paging

- Page Fault Rate \( 0 \leq p \leq 1.0 \)
  - If \( p = 0 \) no page faults
  - If \( p = 1 \), every reference is a fault
- Effective Access Time (EAT)
  \[
  EAT = (1 - p) \times \text{memory_access} + p \times \text{page_fault_time}
  \]
- 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
- \[
  EAT = (1 - p) \times 200 + p \times 8,000,000
  = (1 - p) \times 200 + p \times 8,000,000
  = 200 + p \times 7,999,800
  \]
- If one out of 1,000 access causes a page fault, then 
  \( EAT = 8.2 \text{ microseconds} \). This is a slowdown by a factor of 40!!
- If we want just 10% performance degradation, then \( p \) should be \( 220 > (1 - p) \times 200 + p \times 8 \text{ milliseconds} \)
  \( p < 0.0000025 \), i.e., 1 page fault out of 400,000 accesses
Processes are busy swapping pages in and out

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: locality → a set of pages that are actively used
  - Process migrates from one locality to another
  - Localities may overlap
When \( \Sigma \) size of locality > total memory size thrashing occurs...

Why locality model

- Suppose we allocate enough frames for a process to accommodate its locality
- It will fault for the pages in its locality until all these pages are in memory. Then, it will not fault again until it changes its localities.
- Otherwise, if we do not allocate enough frames, the process will thrash, since it cannot keep in memory all the pages that it is actively using.
- The working-set model is based on the assumption of locality.
Working-Set Model

- \( \Delta \) = working-set window = a fixed number of page references, example: 10,000 instruction
- \( \text{WSS} \) (working set of Process \( P \)) = total number of pages referenced in the most recent \( \Delta \)
  - if \( \Delta \) too small, will not encompass entire locality
  - if \( \Delta \) too large, will encompass localities
  - if \( \Delta = \infty \) \( \Rightarrow \) will encompass entire program
- \( D = \Sigma \text{WSS}_i \) = total demand frames
  - if \( D > (\text{available frames}) m \) \( \Rightarrow \) Thrashing

Thus, if \( D > m \), then suspend one of the processes

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: \( \Delta = 10,000 \)
  - Timer interrupts after every 5000 references
  - 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 references

Working-Set Definition

- 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.
  - 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.

PAGE REPLACEMENT

- What happens if there is no free frame?
  - Terminate a user program
  - Swap out some page
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

1. 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
4. Resume 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 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 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 ⇒ more page faults

<table>
<thead>
<tr>
<th>reference string</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td>page frames</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
**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
- 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

Given the reference string of page accesses: 1 2 3 4 2 3 4 1 2 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
Implementing LRU Algorithm

- 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:
  - Page referenced:
    - move it to the top
    - requires 6 pointer ops to be changed
    - No search for replacement
    - Least recently used one is at the bottom
- 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 its 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

Circular queue implementation:
1. A pointer indicates which page will be replaced next
2. When a frame is needed, it advances its pointer until find one with a 0 reference bit.

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>

Department of Computer Science @ UTSA

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

ALLOCATION KERNEL MEMORY

Kernel Memory Allocation

- E.g., Linux PCB (struct task_struct)
  - > 1.7 Kbytes each
  - Created on every fork and every thread create
  - : clone()
  - deleted on every exit

- Kernel memory allocators
  - Buddy system
  - Slab allocation

Buddy System (Dividing)

Two continuous blocks with the same size
the first one will start as $2^n$
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.

**Question:** What is the worst-case internal fragmentation?
Slab Allocator

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

Slab

- 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 slabs, new slab allocated

Benefits include
- No fragmentation,
- Memory request is satisfied quickly

WHAT HAPPENS WHEN ALLOCATING MEMORY
Memory allocation (using mmap/brk)

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

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

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

Memory Mapping (mmap or brk)

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

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

Now, the heap is allocated from the kernel, which means the virtual address from 0x0804b000 to 0x0806c000 (total 33 pages) are usable. ptr is actually 0x804b008.
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?

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