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

Reject it! But if you want to support more processes,

Swap out an old process to a disk

(what if no free region is big enough?)
**Swapping**

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

  - **Backing store** – fast disk 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 would be needed to 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

- When the old process is swapped in, can we relocate it? *(depends on address binding)*
  - What if the swapped out process was waiting for I/O
    - Let OS kernel handle all I/O, extra copy from kernel to user space

- 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 \( \rightarrow \) 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

<table>
<thead>
<tr>
<th>Virtual space</th>
<th>Physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–4K</td>
<td>0–4K</td>
</tr>
<tr>
<td>4–8K</td>
<td>4–8K</td>
</tr>
<tr>
<td>8–12K</td>
<td>8–12K</td>
</tr>
<tr>
<td>12–16K</td>
<td>12–16K</td>
</tr>
<tr>
<td>16–20K</td>
<td>16–20K</td>
</tr>
<tr>
<td>20–24K</td>
<td>20–24K</td>
</tr>
<tr>
<td>24–28K</td>
<td>24–28K</td>
</tr>
<tr>
<td>28–32K</td>
<td>28–32K</td>
</tr>
<tr>
<td>32–36K</td>
<td>32–36K</td>
</tr>
<tr>
<td>36–40K</td>
<td>36–40K</td>
</tr>
<tr>
<td>40–44K</td>
<td>40–44K</td>
</tr>
<tr>
<td>44–48K</td>
<td>44–48K</td>
</tr>
<tr>
<td>48–52K</td>
<td>48–52K</td>
</tr>
<tr>
<td>52–56K</td>
<td>52–56K</td>
</tr>
<tr>
<td>56–60K</td>
<td>56–60K</td>
</tr>
<tr>
<td>60–64K</td>
<td>60–64K</td>
</tr>
</tbody>
</table>

Department of Computer Science @ UTSA
Page Table

- Each process has one page table
  - Map page number \(\rightarrow\) 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
Page #: 4 bits
Offset: 12 bits

Frame #: 3 bits
Offset: 12 bits

Physical address: 15 bits

How many virtual pages?
How many physical frames?
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
    - 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

```
+-----------------+-----+-----+-----+-------------------+
| Protection | D   | R   | V   | Frame number      |
+-----------------+-----+-----+-----+-------------------+
```

- Dirty bit
- Referenced bit
- Valid bit
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?
How big the page table is?
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 $128 \times 4$MB = 512 MB
  - 50% of memory will be used by page tables?

*How can we get smaller page table?!*
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.
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
Two-Level Page Tables

- Why it is good?
- We don’t have to allocate all levels initially
- They don’t have to be continuous
Example: 2-level Address Translation

- \( p_1 = 10 \) bits
- \( p_2 = 10 \) bits
- offset = 12 bits

- Page number
- Page offset

0
1

- 1st level page table
- 2nd level page table

Which tables should be in memory?
Memory Requirement of Page Tables

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

- Example: a process with working-set size of 32 MB (recall that 1GB memory and 32 bits virtual address):
  - 4KB / page $\rightarrow$ process has 8K ($8 \times 2^{10}$) virtual pages.
  - One 2\textsuperscript{nd} level page table maps $2^{10}$ pages;
  - Number (minimum) of 2\textsuperscript{nd} level page table needed:
    $$8 = \frac{8 \times 2^{10}}{2^{10}} = 8$$
  - Total (minimum) memory for page table: 1\textsuperscript{st} level page table + 8; in total of 9 page tables $\rightarrow$ 9 $\times$ 4KB = 36 KB.
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 apges: 0x0000000, and 0xFFFFF000)

Page table size = $(2^{10}$ level-0 entries) * 4bytes + $(2^{10}$ level-1 entries * 4 bytes) * 2 = 12 Kbytes
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?
Linux’s 3 level page table

Segment + Offset = 4 GB Linear address (32 bits)
user space = 3 GB (defined by TASK_SIZE macro) and
kernel space = 1GB
Linear Address converted to physical address using 3 levels
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
Size of Page/Frame: How Big?

- Determined by **number of bits in offset** (12 Bit $\rightarrow$ 4 KB)

- 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 $\rightarrow$ fit into one frame
  - 32 bits machine, 10 bits for each level

*How can we make the address translation faster?*
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 \(\rightarrow\) 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

<table>
<thead>
<tr>
<th>Logical page #</th>
<th>Physical frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>29</td>
<td>6</td>
</tr>
<tr>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Example TLB
Address Translation with TLB

What happens when CPU performs a context switch?
Integrating TLB and Cache

“Translation Lookaside Buffer” (TLB)
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 as:
  - 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
Bring a page into memory only when it is needed

DEMAND PAGING
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

- Pager vs. Swapper
  - Page only vs. contiguous space
  - Lazy swapper – bring only the pages that are needed
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated,
- \( v \Rightarrow \text{in-memory}, \) \( i \Rightarrow \text{not-in-memory} \)
- Initially bit is set to \( i \) on all entries
- During address translation, if valid–invalid bit in page table entry is \( i \Rightarrow \text{page fault (trap)} \)
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 $0 \leq p \leq 1.0$
  - 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) x 200 + p (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 should be
  
  \[ 220 > (1 - p) \times 200 + p \times 8,000,000 \]
  
  \[ < 0.00000025 \text{, i.e., 1 page fault out of 400,000 accesses} \]
A process is 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**

- Process migrates from one locality to another
- Localities may overlap

When $\Sigma$ size of locality > total memory size, thrashing occurs...

Increase locality in your programs!
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references, example: 10,000 instruction

- $WSS_i$ (working set of Process $P_i$) = 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

- $D = \sum WSS_i \equiv$ 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 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-Fault Frequency (PFF) Scheme

- Working set is a clumsy way to control thrashing
- PFF is a more direct way
  - High PFF $\rightarrow$ more thrashing
  - Establish “acceptable” page-fault rate
    - If actual rate is too low, process loses frame
    - If actual rate is too high, process gains frame
  - Suspend a process if PFF is above upper bound and there is no free frames!
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 improvement, 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 0 1 2 0 3 0 4 2 3 0 3 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

reference string

```
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
```

page frames

```

``
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

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

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

\[
\begin{array}{cccccccccccccccc}
7 & 0 & 1 & 2 & 0 & 3 & 0 & 4 & 2 & 3 & 0 & 3 & 2 & 1 & 2 & 0 & 1 & 7 & 0 & 1 \\
7 & 7 & 7 & 2 & 2 & 4 & 4 & 4 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

page frames
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 performance

- **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

![Diagram of stack implementation](image-url)
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
  - Enhance it modify bit, avoid replacing modified pages

What if all bits are 1 …. All pages will get second chance…. Degenerates FIFO
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>
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
Kernel Memory Allocation

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

- Kernel memory allocators
  - Buddy system
  - Slab allocation
Buddy System (Dividing)
Free Page’s List

2^n pages

8 pages

4 pages

2 pages

2 pages

1 page

1 page

1 page
Page Allocation

order free_area_t
zone->free_area

0
1
2
3
4
5
6
7
8
9
MAX_ORDER

Requesting Process

2 x 2^2 block
2 x 2^3 block
1 x 2^4 block
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?
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 $2b \times 2^{12}$.
- 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 allocator (2)

Cache 1
Objects of 512 bytes

Allocated 512 bytes object

Cache 2
Objects of 1024 bytes

Free 1024 bytes object

4 KB page
Slab Allocator

- **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)

```c
#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 allocation

```c
#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 33K) are usable. `ptr` is actually 0x804b008.
Memory Mapping (mmap or brk)

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

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

0804b000-0806c000 rw-p [heap]
Memory Mapping (mmap or brk)

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

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

0804b000-0806c000 rw-p [heap]
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?
Tracking Memory Usage: Bitmaps

- Keep track of free / allocated memory regions with a bitmap
  - One bit in map corresponds to a fixed-size region of memory
  - Bitmap is a constant size for a given amount of memory regardless of how much is allocated at a particular time

- Chunk size determines efficiency
  - At 1 bit per 4KB chunk, we need just 256 bits (32 bytes) per MB of memory
  - For smaller chunks, we need more memory for the bitmap
  - Can be difficult to find large contiguous free areas in bitmap

![Diagram of memory regions and bitmap]
Tracking memory usage: Linked Lists

- Keep track of free / allocated memory regions with a linked list
  - Each entry in the list corresponds to a contiguous region of memory
  - Entry can indicate either allocated or free (and, optionally, owning process)
  - May have separate lists for free and allocated areas

- Efficient if chunks are large
  - Fixed-size representation for each region
  - More regions => more space needed for free lists

```
Memory regions
```

```
A  0  6  |  -  6  4  |  B  10  3  |  -  13  4  |  C  17  9
```

```
D  26  3  |  -  29  3
```

Linked List: Allocating Memory

- Search through region list to find a large enough space
- Suppose there are several choices: which one to use?
  - First fit: the first suitable hole on the list
  - Next fit: the first suitable after the previously allocated hole
  - Best fit: the smallest hole that is larger than the desired region (wastes least space?)
  - Worst fit: the largest available hole (leaves largest fragment)
- Option: maintain separate queues for different-size holes

Allocate 20 blocks first fit
Allocate 12 blocks next fit
Allocate 13 blocks best fit
Allocate 15 blocks worst fit

Department of Computer Science @ UTSA
Linked List: Memory De-allocation

- Allocation structures must be updated when memory is freed
- Easy with bitmaps: just set the appropriate bits in the bitmap
- Linked lists: modify adjacent elements as needed
  - Merge adjacent free regions into a single region
  - May involve merging two regions with the just-freed area
Memory Allocator for multithreaded programs (Hoard)

Super Blocks

“Thread” 1

“Thread” n
Memory Allocator for multithreaded programs (Hoard)

```plaintext
t1: x9 = malloc(s);

global heap

heap 1
x1
x2
x9

heap 2
y1
y2
y3
y4

t1: free(y4);

global heap

heap 1
x1
x2
x9

heap 2
y1
y2
y3

t2: free(x2);

global heap

heap 1
x1
x9

heap 2
y1
y2
y3

t2: free(x9);

global heap

heap 1
x1

heap 2
y1
y2
y3
```
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