# Final Review

ICS332

**Operating Systems**

Henri Casanova (henric@hawaii.edu)

# Final Details

- Closed notes/computer/phone
  - Pocket calculator allowed (but it's all powers of 2!!)
- See https://courses.ics.hawaii.edu/ics332_spring2026/morea/Final/reading-finalexam.html
-

# Full Virtual-To-Physical Address Translation Narrative

- Split address into page number and offset
- Look up the TLB to find the frame number for the page
- If found in the TLB, build the physical address and **Done!**
- If not found in the TLB, then lookup the page table
- If page table entry is valid: build the physical address, update TLB and **Done!**
- If not, then trap to the OS with a page fault and put the process in the blocked state
- The OS checks if there is a free frame in RAM
- If there is no free frame the OS creates one:
  - The OS selects a victim page
  - If the victim is dirty, then the OS writes it back to disk
  - The OS updates the page table (and TLB entry) of process that owns the victim
  - The OS marks the frame of the victim as free
- The OS loads the missing page to RAM into the free frame
- The OS updates the process' page table and schedules the process again (Ready state)
- The process runs at some point issues the logical address (again)
- Split address into page number and offset
- Lookup TLB to find the corresponding frame number
- It will NOT be found in the TLB, so lookup the page table to find the frame number
- Update the TLB with the page table entry
- Build the physical address and **Done!**

# Sample Short Questions

- What is the goal of the TLB?
- What characteristic of programs makes the TLB be effective?
- Why would we ever want a 2-level hierarchical page table?
- Is a page table always updated after a page fault is resolved?
- Is BestFit always better than WorstFit?
- Without a TLB, each time I access a memory location I would actually access how many memory locations (single-level page table)?

# Sample Short Questions

- What is the goal of the TLB?
  - To speed up address translation
- What characteristic of programs makes the TLB be effective?
  - Locality
- Why would we ever want a 2-level hierarchical page table?
  - To avoid having a contiguous page table that's bigger than a page
- Is a page table always updated after a page fault is resolved?
  - Yes
- Is BestFit always better than WorstFit?
  - No
- Without a TLB, each time I access a memory location I would actually access how many memory locations (single-level page table)?
  - 2

# Sample Short Questions

- What is a problem with Contiguous Memory Allocation?
- Does paging remove all fragmentation problems?
- Without a dirty bit, what would be a problem?
- Increasing disk size helps with thrashing?
- Thrashing can be solved by adding cores?

# Sample Short Questions

- What is a problem with Contiguous Memory Allocation?
  - Fragmentation
- Does paging remove all fragmentation problems?
  - No, there is still internal fragmentation
- Without a dirty bit, what would be a problem?
  - Useless disk writes
- Increasing disk size helps with thrashing?
  - No
- Thrashing can be solved by adding cores?
  - No

# Sample Short Questions

- Does a container come with its own kernel?
  - 
- What's the one advantage of HDDs over SSDs?
  - 
- Should I reserve a code for the kernel?
  - 
- With i-nodes, some blocks are faster to access than others?
  - 
- What's a known problem with SSDs
  - 

# Sample Short Questions

- Does a container come with its own kernel?
  - No
- What's the one advantage of HDDs over SSDs?
  - Cost / byte
- Should I reserve a code for the kernel?
  - No
- With i-nodes, some blocks are faster to access than others?
  - True
- What's a known problem with SSDs
  - Write amplification

# Sample Exercise

```
int a = 10;
int p1, p2;

p1 = fork();
if (p1 != 0) {
    a++;
    sleep(200);
    p2 = fork();
    if (p2 == 0) {
      sleep(300);
      a++;
    }
    printf("%d\n",a);
} else {
    printf("%d\n",a);
}
```

- What does this code print?

# Sample Exercise

```
int a = 10;
int p1, p2;

p1 = fork();
if (p1 != 0) {
    a++;
    sleep(200);
    p2 = fork();
    if (p2 == 0) {
      sleep(300);
      a++;
    }
    printf("%d\n",a);
} else {
    printf("%d\n",a);
}
```

- The output is 10, 11, 12

# Sample Exercise

- Two threads, one global variable a = 0;
- Thread #1 does: a+=2;
- Thread #2 does: a-=1;
- What are the possible final values of a?

# Sample Exercise

- Two threads, one global variable a = 0;
- Thread #1 does: a+=2;
- Thread #2 does: a-=1;
- What are the possible final values of a?

- The “clean” execution is a = 1
- Each thread could have a lost update, which would lead to -1 or 2
- Answer: {-1, 1, 2}

# Sample Exercise

- Consider the following code fragment:

```
1 int a = 0;
2
3 void f(int x) {
4   x++;
5   a  = a + x;
6 }
```

- Where would you add lock() and unlock() calls so that multiple threads can safely call this function “at the same time”, while making the critical section as short as possible?

# Sample Exercise

- Consider the following code fragment:

```
1 int a = 0;
2
3 void f(int x) {
4   x++;
    lock();
5   a  = a + x;
    unlock();
6 }
```

- Where would you add lock() and unlock() calls so that multiple threads can safely call this function “at the same time”, while making the critical section as short as possible?

# Lock: Spinning vs. Blocking

- Spinning:
  - burn CPU cycles checking the lock continuously, which is wasteful
  - but as soon as the lock is released by whoever had it you grab it, which is good
- Blocking lock:
  - Go to ”sleep” asking for somebody to wake you up when the key’s ready, which avoids being a useless CPU hog
  - But this requires much more work as the OS is now involved to put you to sleep and wake you up
    - Moving your PCB from various queues, etc.
- Rules of thumb:
  - Spinning for a long time is wasteful (wasted CPU)
  - Blocking for a short time is wasteful (high overhead)

# Deadlocks

- Make sure you understand resource allocation graphs
  - If the ”boxes” have more than ”one dot”: if there is a cycle, there may be a deadlock
  - If the ”gray boxes” have only ”one dot”: if there is a cycle, there is a deadlock
- Should we do an example?

- Remember the in-class exercise with 9 locks and 2 threads?

# Sample Exercise

- Given 22-bit logical addresses, and a 64KiB page size, how is a logical address split into page number and offset for a single-level page table?

# Sample Exercise

- Given 22-bit logical addresses, and a 64KiB page size, how is a logical address split into page number and offset for a single-level page table?

- 64KiB = $2^{16}$ bytes
- Therefore: offset is 16-bit
- Therefore: page number is 22 - 16 = 6-bit

# Sample Exercise

- Given 32-bit virtual addresses, and a 8KiB page size, and a 4-byte page table entry size, how is the address split into page number and offset for a 2-level page table, assuming that the inner page table fits exactly in one page?

# Sample Exercise

- Given a 32-bit address space, and a 8KiB page size, and a 4-byte page table entry size, how is the address split into page number and offset for a 2-level page table, assuming that all inner page table pages are full?
- 8KiB = 213 bytes
- Therefore: offset is 13-bit
- Number of page table entries that can fit in a page: $2^{13}/4 = 2^{11}$
- The inner virtual page number is 11-bit (because each inner page table page is full)
- Remains 32-13-11 = 8 bits for the outer virtual page number

# Sample Exercise

- Consider  24-bit virtual addresses, and a 8K page size. We know that the single-level page table uses only 1/2 a page. How big are the page table entries?

# Sample Exercise

- Consider a 24-bit address space, and a 8K page size. We know that the single-level page table uses only 1/2 a page. How big are the page table entries?
  - address space is $2^{24}$ bytes
  - Page size: $2^{13}$ bytes
  - Number of pages: $2^{24}/2^{13} = 2^{11}$
  - Number of page table entries: $2^{11}$
  - Let s be the size of a page table entry
  - We have: $2^{11} \times s = (1/2) \times 2^{13}$ Which gives: $s = 2^{12-11} = 2$ bytes

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- Was (virtual) address 1900 written to?
- What does (virtual) address 999 translate to?
- Give a virtual address that'll cause a page fault
- What (virtual) address corresponds to byte 7321 in physical memory?
- Is it possible that (virtual) address 132 was written to?

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- Was (virtual) address 1900 written to?

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- Was (virtual) address 1900 written to?
- Virtual address 1900 is in page 1900/500 = 3
- The dirty bit for the entry for page 3 is NOT set
- So the answer is: NO

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- What does (virtual) address 999 translate to?

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- What does (virtual) address 999 translate to?
- Virtual address is in page 999/500 = 1
- Offset within the page is 999 % 499 = 499
- Page 1 is in frame #11
- The physical address is thus 11 * 500 + 499 = 5999

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- Give a virtual address that'll cause a page fault

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- Give a virtual address that'll cause a page fault
- The page table entry for Page #4 is marked as invalid, so let's access it
- For instance, 4×500+42 is in page 4
- Therefore, accessing address 2042 will cause a page fault

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- What (virtual) address corresponds to byte 7321 in physical memory?

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- What (virtual) address corresponds to byte 7321 in physical memory?
- This address is in frame 7321/500 = 14
- Per the page table, frame 4 contains page 3
- The offset in the frame is 7321%500 = 321
- Therefore, the virtual address is $3 * 500 + 321 = 1821$

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- Is it possible that virtual address 132 was written to?

# Sample Exercise

- Page size = 500 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 0 | 3 | 1 | 1 |
| 3 | 14 | 1 | |
| 4 | 12 | 0 | |
| 1 | 11 | 1 | 1 |
| 2 | 33 | 1 | |
| 200 | | 0 | |

- Is it possible that virtual address 132 was written to?
- Virtual address 132 is in page 0
- The entry for page 0 says that the page is dirty
- So yes, it's possible

# Sample Exercise

- Page size = 1000 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

- Fill in information in the page table
  - Address 1010 has been successfully written to
  - Address 2100 has been read successfully but never been written to
  - At physical address 3099 we have a byte that we have read once and that is in logical page x where x > 10

# Sample Exercise

- Page size = 1000 bytes, decimal addresses

| Page # | Frame | Valid | Dirty |
|---|---|---|---|
| 1 | | 1 | 1 |
| 2 | | 1 | |
| x | 3 | 1 | |
| | | | |
| | | | |
| | | | |

- Fill in information in the page table
  - Address 1010 has been successfully written to
  - Address 2100 has been read successfully but never been written to
  - At physical address 3099 we have a byte that we have read once and that is in logical page x where x > 10

# Sample Exercise

- Page Replacement exercise
- Given a number of frames, given a sequence of logical page references, and given a page replacement algorithm, determine which page references will page fault
- We've seen only 2 algorithms: FIFO, LRU
- Let's do them all on one example again…

# 4 Memory Frames

- FIFO

| Page ref: | 0 | 1 | 7 | 2 | 3 | 2 | 7 | 1 | 0 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 3 |
| Frame 1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| Frame 2 | | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| Frame 3 | | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Fault | ✓ | ✓ | ✓ | ✓ | ✓ | | | | ✓ | |

# 4 Memory Frames

- LRU

| Page ref: | 0 | 1 | 7 | 2 | 3 | 2 | 7 | 1 | 0 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 0 | 0 |
| Frame 1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Frame 2 | | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| Frame 3 | | | | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| Fault | ✓ | ✓ | ✓ | ✓ | ✓ | | | | ✓ | ✓ |

- In this example LRU is worse than FIFO!

# Thrashing

- Make sure you're ready to answer questions about Thrashing
  - Why does it occur?
  - What are solutions?
- Sample Question: Consider a server that's currently running many processes, none of them doing a lot of I/O, and yet we observe 20% CPU utilization and 99.9% disk utilization. Which of these options would help this situation:
  - Install a faster CPU
  - Install a bigger disk
  - Allow more processes into the ready queue
  - Kill some processes
  - Buy more RAM
  - Buy a faster disk

# Thrashing

- Make sure you're ready to answer questions about Thrashing
  - Why does it occur?
  - What are solutions?
- Sample Question: Consider a server that's currently running many processes, none of them doing a lot of I/O, and yet we observe 20% CPU utilization and 99.9% disk utilization. Which of these options would help this situation:
  - Install a faster CPU
  - Install a bigger disk
  - Allow more processes into the ready queue
  - Kill some processes
  - Buy more RAM
  - Buy a faster disk