CSE 421/521 Midterm Solutions
—SOLUTION SET—

26 Mar 2014

This midterm exam consists of three types of questions:

1. **10 multiple choice** questions worth 1 point each. These are drawn directly from lecture slides and intended to be easy.
2. **6 short answer** questions worth 5 points each. You can answer as many as you want, but we will give you credit for your best four answers for a total of up to 20 points. You should be able to answer the short answer questions in four or five sentences.
3. **2 long answer** questions worth 20 points each. **Please answer only one long answer question.** If you answer both, we will only grade one. Your answer to the long answer should span a page or two.

Please answer each question as **clearly** and **succinctly** as possible. Feel free to draw pictures or diagrams if they help you to do so. **No aids of any kind are permitted.**

The point value assigned to each question is intended to suggest how to allocate your time. So you should work on a 5 point question for roughly 5 minutes.

Statistics:

- **150** students took this exam.
- **35** was the median score.
- **35.62** was the average score.
- **6.66** was the standard deviation of the scores.

Multiple Choice

1. (10 points) Answer all **ten** of the following questions. Each is worth **one** point.

   (a) What song was played *twice* before class this semester?
   [✓ **"Breathing Underwater" by Metric.**] ◯ "Ready to Start" by Arcade Fire. ◯ "The Hockey Song" by Stompin' Tom Connors. ◯ This is a trick question; GWA never plays a song twice!

   (b) Which of the following instructions can cause an exception?
   [✓ **All of them.**] ◯ `addiu` ◯ `lw` ◯ `syscall`

   (c) Con Kolavis was particularly interested in improving what aspect of Linux scheduling?
   ◯ Overhead. ◯ Throughput. [✓ **Interactive performance.**] ◯ Awesomeness.

   (d) What interface does something have to support to look like memory?
   ◯ `lock()` and `unlock()`. ◯ `malloc()` and `free()`. ◯ `fork()` and `exec()`. [✓ **Load and store.**]

   (e) Acquiring a lock will never create a synchronization problem.
   ◯ True. [✓ **False.**]

   (f) Which of the following is *not* a requirement for deadlock?
   ◯ Multiple independent resource requests. [✓ **A linear dependency graph.**] ◯ Protected access to shared resources. ◯ No resource preemption.

   (g) Threads can be implemented without kernel support.
   [✓ **True.**] ◯ False

   (h) Which of the following is the simplest scheduling algorithm to implement?
   ◯ Rotating staircase. ◯ Multi-level feedback queue. ◯ Round-robin. [✓ **Random.**]

   (i) Which of the following is *not* an example of an operating system mechanism?
   ◯ A context switch. ◯ Loading a virtual address into the TLB. ◯ Locks and semaphores. [✓ **Prioritizing interactive threads.**]

   (j) All of the following are a good fit for the address space abstraction *except*
   ◯ virtual-to-physical address translation. ◯ `sbrk`. ◯ the TLB. [✓ **base-and-bounds address translation.**]

Short Answer

Choose **4 of the following 6** questions to answer. You may choose to answer additional questions, in which case you will receive credit for your best four answers.

2. (5 points) We have discussed several cases where operating systems provide a useful illusion to processes. Name one, describe why it is useful, and briefly explain how it is provided, identifying any hardware cooperation required.

Graded by Zihe Chen.

Rubric:

- **+1 point** for identifying the illusion.
- **+2 point** for explaining why it is useful.
- **+2 points** for describing how it is provided.

Solution:

There were multiple possible answers to this question. Here are a few of the obvious ones (although you may have received credit for another if you convinced us it was legitimate):

1. **Concurrency.** One single-core systems concurrency is an illusion, since there are never really two threads running at once. This is useful since it provides users with the illusion that multiple things are happening at once—music is playing while the web page is updating while they are typing into the terminal. Concurrency is provided by rapidly switching between threads fast enough to human perceptual limitations. This requires the ability to perform context switches between threads, stopping one's use of the processor to allow another to continue, and then returning the first to exactly where it was stopped.
2. **Atomicity.** When we discussed synchronization we identified cases where we wanted to make multiple actions that actually occurred separately happen all at once, or *atomicly*. An example is a modification to a shared data structure that requires multiple operations that should all happen at once—imagine adding an entry to a synchronized linked list, which requires updating multiple pointers. Atomicity is more than useful, and in certain cases (like our example) is actually required for correctness. One way we provide the illusion of atomicity is by locking shared data structures, which forces other users to sleep while we make our modifications, allowing them to all look atomic.
3. **Address spaces.** Address spaces comprise multiple illusions: that processes have access to a large (1) amount of private (2) contiguous (3) memory (4). In reality, (1) the size of the address space may be larger than the amount of available

physical memory. In addition, the same physical memory may be temporally-multiplexed between processes, making it not entirely private (2). And it is certainly not contiguous, with any virtual page mapping to any physical page (3), and may not even be memory at all (4) if the page has been swapped to disk or the virtual address is set up to point to a file by `mmap`. But the address space abstraction is useful because it simplifies how processes deal with memory. They can lay out their memory the same way each time and logically separate regions, such as the stack and heap, in ways that allow them to grow dynamically. At some level all of these illusions are provided by translating virtual addresses into physical addresses, with the extra level of indirection allowing the OS to move memory around and reuse it for other purposes as long is the virtual addresses obey the memory interface.

Statistics:

- **134 out of 150** students answered this question.
- **5** was the median score. (This question was too easy.)
- **4.43** was the average score.
- **0.95** was the standard deviation of the scores.

3. (5 points) First, explain how locking shared data structures can reduce performance on multicore systems. Second, describe how to perform more intelligent locking to improve the performance of the following code snippet *without* sacrificing correctness or significantly increasing the amount of space needed to store the items. As a hint, you can assume that the loop frequently has to examine many entries before it finds one that is available. The code to remove entries is not shown, but you can assume that entries are periodically removed.

```
struct item {
  bool valid;
  int value;
};

struct item array[32768];
struct lock * arrayLock;

int
saver(int doubleRainbow)
{
  bool failed = 1;

  // Assume that arrayLock was properly initialized.

  lock_acquire(arrayLock);

  for (int i = 0; i < 32768; i++) {
    if (array[i].valid == 0) {
      array[i].value = doubleRainbow;
      array[i].valid = 1;
      failed = 0;
      break;
    }
  }

  lock_release(arrayLock);
  return failed;
}
```

Graded by Jinghao Shi.

Rubric:

- **+1 point** for identifying the multicore performance problem.
- **+2 points** for a correct solution using per-item locks.
- **+4 points** for a correct solution that reduces the locking granularity using the read-lock-read-alter pattern.

Solution:

As mentioned in class during the midterm, we asked you to solve this *without* using another primitive such as a reader-writer lock (obvious) or condition variable (?).

On multicore systems extra locking can reduce concurrency in certain cases. Imagine two threads on separate cores are trying to save a `doubleRainbow` using the code we provided. Once one of them grabs the big lock and begins to walk the entire array, the other has to wait until it is complete. This is despite the fact that a lot of the work can be done *without* holding the lock.

How? By using the read-lock-read-alter synchronization pattern. Here instead of using the lock to ensure correctness during the whole loop, we walk the array without holding the lock and use the current state of `array[i].valid` as a *hint* indicating that this entry might be free. At that point we have a candidate entry to examine further, and then we grab the lock. Once we have acquired the lock, we need to recheck the entry to make sure that it's still available, but if so we can save our `doubleRainbow` and exit. There will be a few cases where concurrent access will result in two threads locking and checking the same entry, at which point only one should win and the other must continue. But this is unlikely, and could be made even less likely by having each thread start their search at a random point in the array rather than at the beginning.

The code follows. Note that instead of holding the lock for the entire loop, we acquire it only when we think we have found an available entry but then need to double-check that this is actually true. (You could actually make this a bit slicker by only setting the `valid` flag inside the lock and then setting the value once you have marked it as in use.)

Per the rubric we gave partial credit for solutions that introduced per-item locks. In a lot of cases this is *not* a good idea, since the overhead for locks isn't worth it for large arrays. Since we specifically told you not to significantly increase the size needed to store items this solution did not receive full credit.

```
int
saver(int doubleRainbow)
{
  bool failed = 1;

  // Assume that arrayLock was properly initialized.

  for (int i = 0; i < 32768; i++) {

    // Safe to check without the lock

    if (array[i].valid == 0) {
      lock_acquire(arrayLock);
      if (array[i].valid == 0) {
        array[i].value = doubleRainbow;
        array[i].valid = 1;
        failed = 0;
      }
      lock_release(arrayLock);
      if (failed == 0) {
        break;
      }
    }
  }

  return failed;
}
```

Statistics:

- **72 out of 150** students answered this question.
- **1** was the median score. (This question was hard.)
- **2.11** was the average score.
- **1.45** was the standard deviation of the scores.

Grader Feedback:

Some students propose to move the `lock_acquire` and `lock_release` in the for loop, like this

```
for (int i = 0; i < 32768; i++) {
  lock_acquire(arrayLock);
  if (array[i].valid == 0) {
    array[i].value = doubleRainbow;
    array[i].valid = 1;
    failed = 0;
    break;
  }
  lock_release(arrayLock);
}
```

While this may improve simultaneous access to the data structure somewhat, it doesn't really solve the problem which is the *amount* of time that each thread is locking the data structure in order to find a free entry.

Some students know to first check valid before acquiring the lock, but forget to check valid again after:

```
for (int i = 0; i < 32768; i++) {
  if (array[i].valid == 0) {
    lock_acquire(arrayLock);
    array[i].value = doubleRainbow;
    array[i].valid = 1;
    failed = 0;
    lock_release(arrayLock);
    break;
  }
}
```

This has obvious problems.

Finally, other students proposed one lock per entry.

4. (5 points) Identify three system calls that allocate new virtual addresses (i.e., allow the process to use them) and describe the semantics associated with each.

Graded by Zihe Chen.

Rubric:

- **+1 point** for identifying each system call.
- **+1 point** for each system call if properly described.

Solution:

This one came pretty much directly from the lecture slides.

1. `exec()`: loads content from the ELF file and sets up virtual addresses mainly that point to the process's code area and any staticly-declared variables.
2. `fork()`: copies the parent's entire address space, including all the code pages, the entire heap, and one (or more) thread stacks, allowing the child to use all of these virtual addresses which will (eventually) have to point to private physical addresses.
3. `sbrk()`: extends the "break point", or the top of the process's heap, allocating new virtual addresses that point to physical memory. Not usually called by processes directly, but instead by `malloc` libraries when their subpage allocator runs out of space.
4. `mmap()`: asks the kernel to map a portion of virtual addresses to point to a region of a file.

Statistics:

- **94 out of 150** students answered this question.
- **4** was the median score.
- **4** was the average score.
- **1.24** was the standard deviation of the scores.

Grader Feedback:

Most students got `fork()` and `execv()`; few listed `mmap()` or `sbrk()`. `read()` and `write()` were popular wrong answers.

5. (5 points) So far we have discussed memory as being uniform, meaning that from the perspective of the core (or cores) the memory access time is constant for each byte of physical memory. On some systems, however, this assumption fails and access time varies with location; we refer to these systems as having a NUMA (non-uniform memory access) design.

   Consider a NUMA system where each core has access to a small amount of (relatively) fast physical memory and a larger amount of (relatively) slow physical memory. Assume that memory management hardware can map process virtual addresses to either part of physical memory. How does this design complicate the address space abstraction? At a high level, describe a way to make use of this NUMA property. (One of the systems design principles we have discussed may come in handy.)

Graded by Jinghao Shi.

Rubric:

- **2 points** for identifying the complication.
- **3 points** for describing an approach to using this NUMA layout.

Solution:

The complication is that NUMA challenges the address space assumption that all memory is uniform. Now, some parts of each process's address space will be faster than others, and if care is not taken these faster parts may change over time as pages move to different regions of physical memory.

The simplest approach to using the faster memory is to treat it as a cache for the slower memory, leveraging our design principle as creating the machine as a series of caches. The operating system should try to put frequently-used pages in the fast memory and less-frequently used pages in the slow memory, which also requires some way of tracking page usage. (Since we haven't covered that yet, not mentioning it didn't hurt your score.)

Statistics:

- **44 out of 150** students answered this question.
- **4.5** was the median score. (This question was too easy.)
- **4** was the average score.
- **1.19** was the standard deviation of the scores.

Grader Feedback:

This question seemed to scare students despite it being not really that difficult. Essentially all you had to do was identify speed differences in main memory and propose to use the faster memory as a cache; you didn't even have to explain for what! The final exam will have more questions like this that ask you to apply the systems design principles you've learned to a new situation.

6. (5 points) Describe two scheduling algorithms, only one of which can be from the "no-nothing" group. For each, provide a one sentence explanation and describe one pro or con of the approach.

Graded by Jinghao Shi.

Rubric:

- **+1 point** per answer for identifying each scheduler.
- **+1 point** per scheduler for a description.
- **+1 point** per scheduler for a pro or con.

Solution:

We discussed several schedulers, including those listed below. Pros and cons here aren't exhaustive and it's likely that I've forgotten a few schedulers and omitted some benefits and limitations.

- **Random.** (This is from the no-nothing group.) Choose a thread at random. Pros: simple and serves as a good comparison point. Cons: too simple and unlikely to produce good performance.
- **Round-Robin.** (This is from the no-nothing group.) Simply establish an ordering between threads and run them in that order. Pros: also simple. Cons: also unlikely to produce good performance, doesn't reward interactivity, doesn't incorporate priorities, etc.
- **Shortest-Job First.** (This is from the know-it-all group.) Order jobs by how long they take to complete and run them until they do. Pros: minimizes average waiting time. Cons: can't be implemented, also doesn't time-share between tasks.
- **Shortest-Remaining Time First.** (This is from the know-it-all group.) Order jobs by how long they will run before they block and run them in that order. Pros: also does a good job at reducing waiting time and improving interactive performance. Cons: can't be implemented, and may starve long-running non-interactive threads.
- **Multi-Level Feedback Queues.** Maintain a list of priority queues. Jobs run round-robin (or random) and always from the top queue first. If a job runs to the end of its quantum, it may be demoted to a lower priority level; if it completes early, it may be promoted to a higher priority level. Pros: rewards interactive use. Cons: may starve long-running background tasks, and all solutions to this issue are somewhat ugly.
- **Rotating Staircase Deadline Scheduler.** I'm not going to try to describe this briefly—instead, refer to the lecture notes. Pros are that it can provide guarantees about interactive response time. The only con that I can think up is that it never made it into mainline Linux, but I'm sure that there are others.

Statistics:

- **147 out of 150** students answered this question.
- **5** was the median score. (This question was too easy.)
- **4.88** was the average score.
- **0.46** was the standard deviation of the scores.

Grader Feedback:

Clearly this question was far too easy. About the only thing that tripped some students up is they forgot what RSDL stood for—nobody lost points for this, however. (It's the **R**otating **S**taircase **D**ead**L**ine scheduler, so the acronym isn't quite perfect.)

Virtual Page Number	Physical Page Number	Permissions
4366	8308	RE
5437	578	RW
4758	5133	R
5173	8179	W
365	8308	RWE
94	2	R

Table 1: **Page Table.**

7. (5 points) Given the current process page table above, indicate the result of the following five load, stores, and fetches (load and execute.) **Note: to make things easier on everyone the question uses base-10 arithmetic and 1000-byte pages.**

Rubric:

Graded by Zihe Chen.

- **+1 point** for each translation.

Solution:

1. store `365004`: store `8308004`
2. fetch `475876`: exception! Virtual page `475` is not in the page table. (Note that `4758` is a valid virtual page but this address is on page `475`. Tricky.)
3. load `94230`: load `2230`
4. fetch `234900`: exception! Virtual page `234` is not in the page table.
5. store `4366100`: exception! Virtual page `4366` *is* in the page table but is marked as read and execute only. (You may have been confused by the fact that both virtual page `4366` and `365` map to physical page `8308`, but (a) this is a valid setup and required to support certain uses and (b) permissions are applied to virtual addresses, not physical addresses, for obvious reasons—physical addresses change and are not exposed to processes.)

Statistics:

- **137 out of 150** students answered this question.
- **4** was the median score.
- **4.08** was the average score.
- **1.1** was the standard deviation of the scores.

Grader Feedback:

On #2, as expect some students mistakenly used `4758`, rather than `475`, as the virtual page number. On #5 some of you missed the permissions.

Long Answer

Choose 1 of the following 2 questions to answer. **Do not answer both questions.** If you do, we will only read one, most likely the one that looks shorter and more incorrect. If you need additional space, continue and clearly label your answer on other exam sheets.

8. (20 points) Choose one of the following two questions to answer:

   1. **Interactivity Detection on Smartphones.** Smartphones are the fastest growing computing platform, with most now running platforms built on top of operating systems with roots in desktop computers such as Linux (Android) and Windows. Just as we discussed in class, performing effective thread scheduling on mobile devices can have a big impact on performance. Mobile smartphones, however, have some important differences from older computing devices that impact the scheduling process, particularly when considering *interactivity*.

      First, describe the interactivity detection problem and explain why this information is generally important to thread scheduling. Continue by presenting two reasons why interactivity detection could be even more important on mobile smartphones. You will want to consider patterns of use, changes in the environment caused by mobility, and constraints specific to smartphones.

      Second, consider how the differences between smartphones and traditional devices affect the interactivity detection problem and propose a new approach to interactivity detection that responds to these differences. You will want to think about what is different about how users interact with mobile phones, as well as the different features on these devices compared to laptops and desktops.

---

   2. **Jumbo Pages.** While operating system pages have traditionally been 4K, some modern operating systems support "jumbo" pages as large as 64K. Based on your excellent and fast implementation of virtual memory for CSE 421 ASST3 you are hired as a kernel developer for the new Lindows © operating system company. Unfortunately, your boss didn't take CSE 421 and derives most of his understanding of operating systems from the movie "Her"[1]. At present, Lindows does not support jumbo pages, but once your boss hears about them he is desperate to include them into Lindows © version 0.0.0.2. He asks you for help.

      First, explain why how and in what cases 64K pages would improve or degrade OS performance. What information about virtual memory use could help the OS decide whether to locate content on a jumbo or regular-sized page? Second, explain how, in certain cases, you can implement jumbo-page-like functionality on top of an existing system that supports 4K pages *without* changing the underlying memory management hardware. What MMU features are required for this to work? Which benefits of jumbo pages are preserved or lost by your approach?

[1]Her is a 2013 American science fiction romantic comedy-drama film centering on a man who develops a relationship with an intelligent OS with a female voice and personality. (Wikipedia)

Rubric:

Graded by Jinghao Shi.

Both long answers were divided into two 10-point sections, each of which covers several issues. Unless superseded by a more detailed rubric, based on your answer you might receive:

- **10 points** for an excellent discussion of all or most of the issues,
- **8 points** if you covered most of the issues but made minor mistakes,
- **5 points** if some of your points are wrong,
- **3 points** for attempts that cover at least one issue,
- **0 points** for nothing or a completely irrelevant answer.

Statistics:

For "Interactivity Detection on Smartphones" (8.1):

- **85 out of 150** students answered this question.
- **12** was the median score.
- **11.69** was the average score.
- **4.07** was the standard deviation of the scores.

For "Jumbo Pages" (8.2):

- **64 out of 150** students answered this question.
- **10** was the median score.
- **11.25** was the average score.
- **4.43** was the standard deviation of the scores.

Grader Feedback:

Generally these questions were not that hard if you read them carefully, figure out what they were asking, and then *followed instructions* when crafting your answer. Both questions include a fairly clear template for what your solution was intended to look like: "First . . . continue . . . You will want to consider . . . Second . . . You will want to think about . . . ." Sadly some of you seemed to fail to read this part of the question!

More question-specific grader feedback follows each answer below.

Solution: Interactivity Detection on Smartphones.

Rubric:

- **10 points** for describing the interactivity problem.
  - **4 points** for defining the problem of identifying interactive tasks.
  - **3 points** for each reason that interactivity detection is important on mobile devices.
- **10 points** for an approach to smartphone scheduling incorporating interactivity.
  - Do you respond to differences between smartphone and other devices?
  - Do you use new smartphone features to help detect interactivity?

Solution:

Schedulers face the challenge of determining when users are waiting, in some form or another, or a task to finish—to animate a cursor or draw a character glyph in response to input, as one example, or to render a page that has been retrieved for the Internet as another. Because users are aware of the performance of interactive tasks, but not aware of the performance of others, schedulers may want to allocate these interactive processes more resources or provide them prioritized access to the processor.

Smartphones are a unique computing device that is both (a) always on and (b) battery powered. Desktops may be left on but are not battery powered, while laptops are battery-powered but usually shut down when not in use and moved around. These two features combine to produce a device that has both an important resource limitation (energy) and the potential for a large amount of background usage. This makes it even more important that smartphone schedulers identify interactive tasks and ensure that non-interactive work does not drain the device's battery. Adding to this challenge is the fact that smartphones face a constantly-changing environment caused by mobility: one minute they have a connection over a high-speed and energy-efficient Wifi network, the next they are forced to use a slower and more power-hungry 3G mobile data network. These changes create the potential for non-interactive background tasks to do even more damage to device lifetimes if they are allowed to use the smartphone inefficiently.

There were several different ways to incorporate the unique features and interaction patterns of smartphones into interactivity detection within the smartphone thread scheduler. Two important aspects of smartphones that you may have noticed and utilized:

- Unlike computing devices utilizing windowing environments allowing multiple apps to be present on the screen at one time, the limited size of smartphone displays means that users are usually only interacting with a single app at one time. While this doesn't eliminate the interactivity detection problem entirely—apps may have multiple threads, some performing interaction, others doing background jobs—it does reduce the problem to determining which of the threads of the foreground app are interactive and which are not. In addition, when the screen is off you can probably assume that *nothing* interactive is going on, although you can't shut down work altogether because apps deliver notifications to users by performing background tasks and may be doing other useful things to optimize the foreground experience. But if the user isn't looking at the device they probably aren't waiting on it!

- Smartphones also feature a number of sensors that may simplify the interactivity detection problem. Onboard cameras may be able to determine whether the user is looking at the device and use that to help guess if they are waiting on an action to complete. Orientation sensors may detect movement associated with gameplay and trigger additional performance for that highly-interactive set of apps.

That said, since the point of this question was to get you to think creatively we will definitely accept other reasonable answers. If you really have a great idea about how to solve this problem, why not come join the `blue` Systems Research Group and try your ideas out for real on the hundreds of smartphones deployed as part of PHONELAB?

Grader Feedback:

Most students were able to define the problem of identifying interactive tasks and some of the resource limitations specific to smartphones, particularly battery. However many students missed the "always on" aspect of smartphones and their ability to run background tasks.

When it came to actually brainstorming new scheduling approaches, Very few of you were able to propose some smartphone-specific approach such as using sensors or exploiting the fact that only one app usually runs at a time. Many students just mentioned conventional way, like counting how many time a thread sleeps.

Sadly, nobody expressed interest for PhoneLab—yet!

Solution: Jumbo Pages.

Rubric:

- **10 points** for discussing jumbo page performance.
  - **4 points** for identifying and explaining spatial locality as the factor in determining whether to use jumbo or normal 4K pages.
  - **4 points** for identifying why jumbo pages improve performance: TLB can map more entries (2 point) and potentially fewer page faults with sufficient spatial locality (2 points).
  - **2 points** for identifying why jumbo pages would degrade performance: more unnecessary IO bandwidth and page faults without sufficient spatial locality.
- **10 points** for the implementation of jumbo pages on top of 4K pages.
  - Do you identify that this requires a software-managed TLB?
  - Does your solution work?
  - Can you describe what performance benefits of jumbo pages are preserved (fewer TLB faults) and what are lost (can't map more memory)?

Solution:

First, how and in what cases do 64K pages improve or degrade performance? This is related to the discussion we had in class regarding page size. A straightforward benefit of 64K pages is that they allow the TLB to map more memory with the same number of entries, potentially leading to fewer TLB misses and associated TLB faults. Since 64K pages are larger than 4K ones, the question reduces to in what case do larger pages help and hurt? They help when there is sufficient *spatial locality* in memory accesses within the contents of the 64K page and they hurt when they do not.

One way to think about it is once I touch one byte of memory on the 64K page, what is the likelihood I will touch bytes on the 15 other 4K pages inside the 64K page? If it's high, I might as well make room for the whole page as soon as I touch any byte inside of it; if it's low, I'm going to be moving a lot of memory around without any gain over locating the byte on a standard 4K page[2] So spatial locality is what the OS would like to know. Maybe the best way of finding out would be to ask the process to tell me directly, so if it has a data structure that spans multiple 4K pages it would be better stored on a 64K page. (In addition, I might be able to tell from accesses on groups of 4K pages that they really all belong to one larger page, but it's not clear if the content can be relocated at this point.)

So how do we implement pseudo-64K pages without actual hardware support, i.e. on 4K underlying pages? Easy: whenever we have a TLB fault within a 64K region identified as a 64K page we load entries (and page contents) for not only the 4K page that caused the fault but for all of the other 15 pages within the 64K page. In a way, we can be even more flexible than real 64K pages, since we could load N extra pages on each side of the faulting page (for example, although $2N+1$ is hard to make even). Clearly this requires a software-managed TLB to do in all cases, since a hardware-managed TLB will load the page translation when it is in memory and not alert the OS. (You could still implement the same behavior on page faults, but that's not exhaustive enough to cover all cases.)

The benefits of 64K pages that we preserve are fewer TLB faults since all 15 extra 4K pages on the 64K page are now loaded in the TLB and into memory. Assuming sufficient spatial locality, this is a good thing! The benefit that is lost is that the amount of memory that the TLB can map is still limited by the underlying 4K page size.

Grader Feedback:

Most students correctly identified spatial locality as the factor in determining whether to use jumbo or normal-size page, and most were able to list both reasons why jumbo pages improve performance: TLB can map more memory, and fewer page faults with sufficient spatial locality. Some students received partial credit for pointing out that kernel paging-related data structures would be smaller.

When discussing the performance degradation from jumbo pages students frequently identified both internal fragmentation (which the solution missed) and less-frequently IO bandwidth (which the solution mentioned). Both received full credit.

The implementation that most of you proposed combined 4K pages properly, but many students forgot that doing this requires a software-managed TLB.