# Final Exam

**Due: Wednesday March 21, 2018 11:59PM**

---

## Overview

Welcome to the last assignment: the final exam! 🎉

In the beginning of the term, we noted that we wanted to do things a bit differently in CS140e. The final exam is no exception. Instead of the traditional 2 to 3 hour sit-down, on-paper exam, you'll be writing a 5 - 7 page *course summary*. The summary should consolidate all of the material you've learned in CS140e including everything we've taught in lecture and assignments.

In lecture, we generally discussed fundamental principles of operating systems: things like what an operating system is and does and how file systems work. In assignments, you implemented a specific instance of many of those principles, though not all of them. The primary goal of your summary is to consolidate all of these topics into one cohesive, complete, and pithy story about operating system design and implementation.

Later on, we provide an outline that your summary should adhere to. The outline isn't prescriptive. This is *writing*, after all, so feel free to be creative as long as you touch on most of the points in the outline. We also provide grading criteria with writing samples in an effort to illustrate what we expect. Our intention is for this to be a fruitful, relaxed exercise that solidifies your

understanding of the course material. Good luck!

---

## The Rules

You are explicitly allowed to:

- **Use your personal notes.**
- **Use handouts passed out during lecture.**
- **Use any online material to clarify concepts.**

  You may use any material you find online as a reference. All writing must be your own. Only use external material as a source of truth, never as a source of writing.

- **Discuss with colleagues at a high level.**

  You may (and we *strongly* encourage you to!) converse with fellow classmates about any of the course material. All writing must be your own.

- **Ask questions to clarify material.**

  You may continue to ask for clarification about course topics on Piazza, at lab, and at office hours. If your question includes extensive details about a particular subject, please post the question privately.

Your must not:

- **Plagarize.**

  This is a given; please don't plagirize! Doing so is a serious offense and will require us to escalate the matter.

- **Share writing with a colleague.**

  All writing must be your own original work.

If you have any questions about what's allowed and what isn't, **please ask**.

---

## Getting Started

The page layout for your summary must conform to the following specifications:

- no more than 7 single-spaced, 8.5" x 11" pages, including diagrams

- as many additional pages as needed for references and feedback
- 10-point type on 12-point (single-spaced) leading
- two columns with 0.85cm of separation
- a top, right, bottom, and left margin of 0.80in
- Times Roman or similar font

#### LaTeX

We have provided a LaTeX template that meets these specifications. If you do not already have a LaTeX distribution installed and wish to use the template, we recommend:

- On macOS: `brew cask install mactex` or MacTeX
- On Ubuntu: `apt-get install texlive-full`
- On Fedora: `dnf install texlive-scheme-full`

You can clone the template repository by running:

```
git clone https://web.stanford.edu/class/cs140e/assignments/final/skeleton.git final
```

You can render the `summary.tex` file into `summary.pdf` by calling `make`. To have `latexmk` continuously render the PDF as you edit, run `make watch`.

#### Word

We also provide a Word template, courtesy of Roslyn Cyrus, if you prefer to use Word. You can download it by clicking here. Due to differences between Word versions, you should ensure that the template meets the above specifications on your particular setup before using it.

---

## Outline

Your summary should be between 5 and 7 pages when rendered as a PDF. Fewer pages are allowed if you feel that you have covered all of the material, but you must not exceed 7 pages. Our expectation is that you will spend no more than a day or two working on this summary.

The outline below lists specific topics that your summary should include. **You shouldn't write towards the outline.** Instead, you should use the outline to guide your writing towards a cohesive story about operating systems that includes most of these topics. Feel free to write about ideas that aren't explicitly mentioned in the outline. Don't feel that you must touch on absolutely everything in the outline. It is better to be clear about a few important topics than it is to write about every topic in an unclear manner.

We expect your summary to read as a narrative or guide to operating systems, not as a series of answers to questions or fragmented ideas. Think about what future students in CS140e would benefit from reading and write towards them. Feel free to use diagrams, code fragments, illustrations, tables, and figures liberally.

Your summary should cover the majority of the following:

- **Rust** (2 lectures)

  Your summary should discuss the Rust programming language and its role in operating systems development. You should seek to respond to many, though not necessarily all, of the following:

  1. What are some unique, key properties of Rust?
  2. Describe the role of lifetimes and ownership in preventing memory errors.
  3. What advantages does Rust have when developing operating systems?
  4. Provide a few examples where something that is relatively easy to do in a language like C, such as dereferencing a raw pointer, is more complicated or different in Rust. Why?
  5. Provide a few examples where something that can be done concisely in Rust is more verbose in a language like C.

- **Device Drivers** (1 lecture)

  Your summary should describe what a device driver is and how it fits in with the bigger picture of operating systems. You should describe the key differences between device drivers and other software both in the kernel and outside of the kernel.

- **File Systems** (3 lectures)

  1. Both disk and main memory store values. Discuss the similarities and differences between disk and memory.
  2. Give the shortest, intuitive description of what files and directories do, along with the approximate equivalant Rust constructs. (Directories might need two, depending on how you view them.)
  3. Describe a few common workloads and common FS implementation optimizations for them.

4. Sketch a couple rules for writing in-memory data and metadata to disk so that the file system can be correctly recovered after a crash (as discussed these in class). Give one example mistake and its consequences.

5. Give the intuition for how a logging (not log structured) file system works. What happens on crash? What do you do if you crash while repairing a crash?

6. Quickly sketch (few sentences at most) how to systematically test a storage system using the `choose()` interface described in class.

- **Linking** ($1/2$ lecture)

  Your summary should describe the linker's role in compilation. In particular, it should answer the following questions:

  1. What is the linkers primary task? How does it work?

  2. How does the role of a linker change based on the language being compiled?

  3. What and how much control do programmers have of the linker's actions?

- **Memory Allocation** ($1/2$ lecture)

  Your summary should describe memory allocation at two levels: first, in the kernel, and second, in user space. It should also describe how these two levels interact. You should seek to answer the following questions:

  1. What is the role of a memory allocator? Why do they exist?

  2. What are important characteristics of "good" memory allocators?

  3. What are characteristics of "bad" memory allocators?

  4. How does an allocator's interface affect the implementation of the allocator?

  5. How do different programming language interface with memory allocators?

  You should describe particular kinds of memory allocators and their trade-offs and differences. When describing particular implementations of memory allocators, illustrations and code examples will be particularly

useful.

- **Virtual Memory** (2 lectures)

    1. What problems would show up if we didn't have virtual memory (VM)?
    2. Compare page-based VM to segmentation-based. Despite its downsides (which are?) page-based has largely won: what's the crucial advantage advantage it has? What's the advantage of segmentation over page-based and give a workload where it'd work better.
    3. Describe what happens when a TLB fault occurs for both a hardware-controlled and a software-controlled TLB.
    4. If we do not map page tables using virtual memory (i.e., they are referred to with physical addresses), what are advantages/disadvantages compared to mapping them virtually? How do we handle translations faults during translation faults if they are virtual?
    5. When are memory allocation and virtual memory used to do approximately the same thing? What are some reasons that their approaches are so different?

- **Processes** (4 lectures)

    Your summary should describe the design, structure, and implementation of processes. It should also describe when processes are allowed to communicate with eachother and how they do so. Your discussion about processes should be detailed. Your summary should seek to answer the following questions:

    - **Processes and IPC**

        1. Why do processes exist? What problems do they solve?
        2. What abstractions do processes provide and why?
        3. How are processes implemented?
        4. How are processes managed and protected by the operating system?

    - **Scheduling**

        Your summary should describe how operating systems schedule tasks.

It should describe the characteristics of good and bad schedulers with details about specific scheduling algorithms and the implications of their design.

- **Interrupts and Exceptions**

  Your summary should explain what interrupts and exceptions are as well as their role in an operating system. It should seek to answer the following questions:

  1. What utility do interrupts and exceptions provide?
  2. What abstractions do operating systems create for interrupts and exceptions?
  3. What mechanisms do CPUs use to deliver interrupts and exceptions?

  It should describe specific scenarios where interrupts are key to the implementation of operating systems.

- **Privilege Levels and System Calls**

  Your summary should explain the role of system calls in operating systems and why they are necessary. It should describe system calls in the context of privilege levels and privilege separation.

  The discussion on system calls should include details on how they are implemented and what effect their interface has on the overall interface of an operating system. It should also describe how typical users interact with system calls.

- **Synchronization** (1 lecture)

  Your summary should discuss synchronization concerns in programs with concurrent execution. In particular, it should seek to the answer the following questions:

  1. What is *synchronization*? What is *cache coherence*?
  2. What mechanisms can be used to synchronize operations?
  3. What mechanisms and algorithms can be used to provide cache coherence?
  4. What role does the CPU play in enabling synchronization and coherence?

- **Virtual Machines** (1 lecture)

  Note: the following are more suggestions than requirements, since the readings didn't discuss some of these aspects.

  1. Why would you use a virtual machine monitor (VMM)? What's the difference between virtualization and simulation (hint: in class, we claimed not much)?

  2. If you have a workload where the program counter (pc) is almost always in user code, and another where the pc is almost always in systems code, how do you expect these to perform on a VMM?

  3. What's the general methodology for running an OS on top of a VMM as a guest OS "process"?

  4. Explain "trap and emulate".

  5. Why doesn't VMWare statically translate guest OS code before-hand?

  6. Generally, people define "code" as what's in the code (text) segment of an executable. How does VMWare's definition of code differ? How does it translate code? (Note that binary program checkers such as valgrind and pin both define and translate code in much the same way.) Does it have to translate user-level application code? Explain.

---

## Grading

Your summary will be graded on the following criteria:

- **Accuracy**

  Factual correctness is paramount. Incorrect or inaccurate statements will be penalized.

- **Completeness**

  Omission of important topics or connections will be penalized.

- **Pithiness**

  Overly lengthy prose will be penalized. Your writing should be clear, concise, and illustrative.

- **Creativity**

Unique insights or observations, including those not specifically mentioned in class, will be rewarded.

---

## Writing Samples

To help you guage your writing, we provide three writing samples for a particular outline topic. For each writing sample, we list the grade that the summary would receive as well why the summary received that grade. Note that in your writing, you should seek to explore relationships between topics as much as possible.

### Topic

The topic for these writing samples is:

- **Access Control**

  Your summary should explain the role of access control in operating systems. In particular, it should seek to answer the following questions:

  1. What are the common forms of access control?
  2. What are the advantages and disadvantages of different access control implmentations?
  3. How do operating systems enforce access control?

### Writing Sample A

The following summary for the topic would receive an **A**:

- **Background**

  The introduction of multi-user operating systems begged the question about how to share and protect data associated with different users on the same physical machine. How could Alice prevent Bob from reading her confidential files while allowing Carol to read them? Solutions to this problem, and more generally to the problem of enforcing policies on resources owned by different users, is known as *access control*.

  There are many forms of access control with complex and entwined relationships, but all forms have some shared characteristics. In particular,

they all ascribe the concept of an identity to users. A user's identity is used to specify policies. Identities aren't necessarily unique. In Unix based operating systems, for instance, each user is associated with a user ID, a unique identity, and a group ID, a non-unique identity. Both identities can be used to enforce access control.

## Mandatory Access Control

Two forms of access control see pervasive use today. The first, *mandatory access control* or MAC, allows an administrator to specify policies that the system enforces at every access point. The policy is typically specified in one place. It is consulted on every access to a resource. The policy can never be modified or violated by a non administrative user; it is always enforced by the operating system.

An example of a policy might read that only users in a particular group can read file "foo.txt". When the operating system receives a `read()` system call for a particular file, it checks the user's group against the policy. If the policy allows it, the `read()` is allowed. Otherwise it is denied.

## Discretionary Access Control

The second, *discretionary access control* or DAC, propagates the idea of "ownership" to resources managed by the operating system. The distinguishing feature of a DAC based access control system is that it allows any user with a resource to share that resource forward. In other words, if a user has access to a resource, then it can do with the resource what it wishes.

As an example, consider that a policy allows user A to open and read a particular file but not user B. When user A tries to open the file via the `open()` system call, the operating system will allow it. If user B tries to open the same file, the call will be denied. On the other hand, if user A opens the file and then passes the file descriptor to B, the operating system will subsequently allow B to read from the file. User A has effectively shared its permissions to the file at his/her discretion.

## Conclusion

In reality, the complexity of sharing brings about complex access control systems that are neither purely discretionary nor mandatory. Mandatory

access control is typically too course grained to suffice, while discretionary access control is typically too flexible. Access control in Linux, for instance, is both discretionary and mandatory. Linux makes use of the traditional Unix permission bit based DAC for files. With the advent of SELinux, however, global policies can be specified that can restrict the propagation of permissions.

This summary has the following **positive** properties:

- It answers all of the questions in the outline using narrative.
- It provides a fairly complete and accurate picture of access control.
- It provides and justifies a singular definition for access control.
- It provides context for why access control exists, illustrating real problems that it solves.
- It provides examples to help the reader understand the concepts better.
- It explains, in short, how access control applies to modern systems.
- It does not short-change the complexity of the subject.
- While not exceedingly short, it is also not exceedingly long, vying on the reader's background to shorten the discussion.

This summary has the following **negative** properties:

- No specific example of an implementation for DAC is mentioned.

**Writing Sample B**

The following summary for the topic would receive a **B**:

- ## Background

  Operating systems manage resources for users. Users expect the operating system to protect their resources from other users. The mechanisms by which an operating system accomplishes this is known as access control.

  There are many forms of access control. In many of them, users are given an identity, such as user IDs in Unix based operating systems. When a resource is accessed by a user, the operating system checks the user's ID against a policy. The access is only allowed if the policy allows it.

There are two primary kinds of policies in use today.

## Mandatory Access Control

The first, *mandatory access control* or MAC, allows an administrator to specify policies that are enforced by the system every time an access occurs. There is typically one policy. The policy can only be modified by an administrator, and the operating system makes sure to check the policy for every access.

As an example, consider the following policy:

```
user A can read foo
user B cannot read foo
```

If a user A tries to read foo, then the operating system will allow the read. On the other hand, if a user B tries to read the same file, it won't be allowed. Neither A nor B can modify this policy.

## Discretionary Access Control

The second, *discretionary access control* or DAC, allows users to share permissions for a particular resource. Using the same policy from the previous example in a DAC context, a DAC based access control system would allow user A to pass a handle to foo to user B. User B would then be able to read the file because A used discretion to pass the permissions to user B.

## Comparison

Whether you use DAC or MAC depends on which properties the system desires. In particular, if resources are viewed as being "owned", then DAC's abilities to share resource is more appropriate. DACs can prove to be too lenient, however; it is not always the case that a user's access to some resources implies that it can share that access.

If, on the other hand, there is a central policy authority, then MAC is the better option. The downside is, of course, is that MAC can be too restrictive and static. Once the decision is made, it is difficult to change.

## Conclusion

It is also possible for an operating system to use both MAC and DAC. Continuing with the example above, a MAC/DAC hybrid might allow A to share foo with B but still deny B from reading the file. Linux, for example, uses a model that is closer to this.

This summary has the following **positive** properties:

- It answers all of the questions in the outline using narrative.
- It provides examples to help the reader understand the concepts better.
- It somewhat explains how access control applies to modern systems.
- It does not short-change the complexity of the subject.
- While not exceedingly short, it is also not exceedingly long, vying on the reader's background to shorten the discussion.

This summary has the followentwineding **negative** properties:

- The writing has no cohesive narrative.
- No background, or "why?" for access control is provided.
- Many details are introduced and left undiscussed.
- The described implementation of access control is vague.
- Specifics about particular implementations, such as Unix's, are generalized incorrectly.

**Writing Sample C**

The following summary for the topic would receive a **C**:

- **Background**

  Operating systems manage resources for users. Users expect the operating system to protect their resources from other users. The mechanisms by which an operating system accomplishes this is known as access control.

  There are two forms of access control, both of which give users a unique identity. When a resource is accessed by a user, the operating system checks the user's ID against some kind of policy, which depends on the kind of access control. The access is only allowed if the policy allows it.

The first kind of access control, *mandatory access control* or MAC, allows an administrator to globally enforce a policy. This usually done by allowing the administrator to set permissions bits on resources. In Linux, for example, only the administrator can change the permissions of files. Once the change is made, access to that resources are enforced by the operating system. It is also possible to specify policies in a single file.

The second, *discretionary access control* or DAC, allows users to share permissions for a particular resource. For example, a user can pass a file that is owned by them to a different user. This way, the user can read the file where previously they could not. It is up to the user (their discretion) on whether they want to share the resource or not.

### Comparison

DAC is usually faster since the permissions are checked on the resource itself. There is no global policy, so checks can be made locally. Furthermore, DAC allows you share permissions. It is more flexible that MAC.

MAC is typically stricer since the same policy is enforced on all files instead of on a per-file basis. It can also be less performant.

### Enforcement Implementation

Operating systems enforce access control at the system call layer. For every system call that’s made, the operating system will check that the user ID is identified as having the permissions needed in the policy. For DAC, this means checking the bits in the file. For MAC, this means checking the policy file.

### Conclusion

It is also possible for an operating system to use both MAC and DAC. In fact, Linux uses both at the same time.

This summary has the following **positive** properties:

- It provides a singular definition for access control.
- It answers all of the questions in the outline using narrative.

- The described implementation of access control is fairly clear.

This summary has the following **negative** properties:

- The writing feels very unpolished.
- The writing has no cohesive narrative.
- It overgeneralizes details about specific access control implementations.
- It provides incorrect examples related to MAC and DAC.
- It makes inaccurate statements about DAC and MAC.
- It compares the access control methods on performance, an unrelated factor.
- Many details are introduced and left undiscussed.
- Specifics about particular implementations, such as Unix's, are generalized incorrectly.

---

## Submission

When you are ready, submit the generated `summary.pdf` file at the submission page. Your summary should not exceed 7 pages. If you have any feedback on anything related to the course, you may append any number of feedback pages to your summary.

---

## Prologue

A sincere **thank you** for taking part in this inaugural incarnation of CS140e. It has been an absolute pleasure interacting with each and every one of you. I could not have asked for a better group of students.

I genuinely hope that you have learned a great deal throughout the course and that you are able to apply what you have learned in your future endeavors. Operating systems are complex, but they are not complicated. I trust that you have seen this first-hand through the course assignments. Of course, there is an incredible wealth of information about operating systems that we did not cover, but you should now be in a position to discover and acquire that material on your own. If you have any questions, I am always up for a chat about operating systems.

A warm thank you to the course staff for their time and efforts: **Dawson Engler**, for allowing me to shoehorn Rust and 64-bit ARM into a brand new operating systems course, for having invaluable insights about what works and doesn't, and for confiding in me to deliver on the course. And **Jennifer Lin**, for exceeding all expectations about what it means to be a wonderful TA.

Until next time!
Sergio

## Table of Contents