Jessie Huynh CC0

Operating systems, before they were automated.

☼

☰

◀ · Home · ▶

This chapter claims that computers are responsible for some of societies' ills (when really it's society which is responsible for how computers work) : where is the cause and where is the effect?

Since people are biased, the idea was that 'objective' programs would improve things - and sometimes they do! Of course, how 'objective' the program is depends on who programs it, but in theory at least a carefully designed and transparent automatic system will be more fair than one which has to go through human judgement. Sure, people can individually correct for bad policy...but they can also circumvent good policy.

Some of the critiques of this chapter from the book club are that 'it seems angry about the problems but doesn't suggest solutions'

For technical details, sources seem to be frequently secondhand - it is suggested to get more primary sources.

Book club had a discussion about how much this chapter is about operating systems and how much is about user

Chapter 9 Computing

Operating Systems

by Amy J. Ko, Mara Kirdani-Ryan

Key ideas

* The original computer operating systems were people who managed a computer's limited processing resources.
* The masculine culture of computing emerged from the creation of operating systems.
* Modern operating systems use software components like kernels, memory management, process management, network management, and user interfaces to efficiently and automatically manage a computer's resources and enable multitasking.
* Operating system design generally prioritizes speed over other values like accessibility, security, privacy, and free speech.
* Teaching operating systems is an opportunity to link a technical understanding of operating system components with student values, which often are often in tension with efficiency.

Have not heard that idea before!

sounds like it's all automatic and doesn't give much role to user control

not just speed, also protection of I.P. or giving admins / parents / bosses fine grained control

The first digital computers, released in the 1960's, filled entire rooms[8]

The Eniac was digital and was 1945....in 1950s there was the IBM 610 and Olivetti Elea, both of which didn't need a whole room.

[8] Thomas Haigh, et al. (2016). ENIAC in Action: Making and Remaking the Modern Computer. *MIT Press*

. Because they were so large, and so expensive, only a few companies and universities had them – and they usually just had one. What was essentially a big programmable calculator was therefore also a highly protected, managed resource, and only some people were allowed access. Using a computer therefore involved the following:

1. First, you needed to think about the program you wanted to execute on the computer. This might involve sketching that program out – perhaps on paper – thinking through logic and calculations.

2. After you had a plan, you would translate that program onto paper punch cards that encoded the logic of your program's plan into machine instructions that the computer could understand and execute. A simple program might fit on a single punch card; more complex programs might be hundreds of punch cards[14]

[14] Steven Lubar (1992). "Do Not Fold, Spindle or Mutilate": A Cultural History of the Punch Card. *Journal of American Culture*

.

3. Then, to execute the program, you had to physically carry those punch cards to the room where the computer was stored. At the entry to that room was typically a human computer "operator". On a busy day, you might wait in line before you could give the operator your punch cards, or if you did not want to wait, you could come and submit your program late at night, when demand was lower.

I think you could also schedule time?

4. The operator's job was to receive requests to execute programs, maintain a queue of programs that were waiting to be run, insert

punch cards into the computer, wait for results, and then give the printed results to the person who submitted the program. Operators worked in shifts, often 24/7, to maximize use of the computer's precious time, maintain the expensive computer's hardware, and respond to any urgent jobs that might take precedence over the people waiting in line.

5. If you were careful in your programming, after receiving the program output from the operator, you were done. But more likely, there was a defect in your program, and you'd have to carefully analyze what you had encoded in the punch cards – debugging your program – and once you found the mistake, recreate the punch cards and follow this process all over again.

In this chapter, we examine this process, and chronicle the shift from human operators to the modern operating systems we have today, in which nearly every aspect of an operating system is automated and under the control of a few large private businesses. We then discuss the ways this shift created the dominant cultures of CS we have today, and then end with methods for engaging students in dialogue about these systems, their creators, and the increasingly large effects these systems have on our everyday lives.

but Linux?

Jessie Huyhn

Pop culture steers computers to boys.

From mainframe to personal computer

Early systems for managing a computer's precious resources was entirely a human one. And because of this, culture crept its way into who used computers. At the time in the United States, computer programming was viewed as low-level, low-value, repetitive work, and so operators, and the people writing programs on punch cards, were often women, and often, Black women[16]

really 'low value'? I mean, NASA etc depended on it...

[16] R. Arvid Nelsen (2016). Race and computing: The problem of sources, the potential of prosopography, and the lesson of Ebony Magazine. *IEEE Annals of the History of Computing*

. By 1967, computer programming and computer operation was mostly viewed as a women's profession, and a good paying one, as women with

rephrase? lots of clauses. Maybe: 'but a good paying one. At that time, women with a mathmatics education had few other options than teaching or working for an insurance company.

education in mathematics were often barred from other professions, only leaving them lower paying jobs in teaching or insurance companies. Therefore, just as women were the first computers, and the first programmers, they were also the first computer operators.

The transition from human operators to software followed the same capitalist incentives that spurred the invention of cost-efficient digital computers: researchers were economically incentivized to automate computer operation. The idea was to take all of the tasks that the human operator did, and write programs that would do them automatically. What replaced those human operating systems were operating systems

it wasn't just cheaper: it was also much faster and more accurate.

operating system: A computer program that automatically manages the hardware, netowrk, and energy resources of a computer, maximizing speed and enabling multitasking.

OS

(OS): programs installed on a computer to manage a computer's limited resources and respond to requests to execute programs[19]

[19] Andrew S. Tanenbaum & Herbert Bos (2015). Modern Operating Systems. *Pearson*

. This same shift away from human operators also led to a shift in who used them. Personal computers (PCs) emerged in the 1970's, enabling an individual to purchase a computer and operate it themselves, with an operating system to manage the computer's CPU, memory, storage, and input/output devices. Once it was possible for someone to purchase their own computer, mainframe computers slowly disappeared, and the need for human operators diminished[a]

There was a deliberate marketing campaign to drive personal computer purchases instead of mainframes

[a] The job eventually came back: the supercomputers and data centers behind "cloud" computing require immense human labor to operate and maintain. However, rather than maintaining just one large computer, they maintain the hardware and software of hundreds or more computers, all networked to support complex computations or web scale services that reach millions.

can we draw this causation? (Check unlocking the clubhouse)

Throughout this transformation from mainframe to PC, Hollywood reinforced stereotypes of PCs as a boy's toy in movies like War Games, which led marketers to advertise to boys and their parents. The result was that parents were twice as likely to buy computers for their boys than their girls, and if a family had a computer, it was often placed in the boy's room, as a toy[15]

[15] Jane Margolis & Allan Fisher (2003). Unlocking the Clubhouse: Women in Computing. *MIT Press*

. Popular culture responded, creating an image of using computers as mens' work, rather than womens'. University computer labs, and the CS departments that ran them, were often elitist, sexist, racist, ableist, and dominated by men, and created structures and policies that reinforced those views [13]

[13] Kathleen J. Lehman, et al. (2021). Growing Enrollments Require Us to Do More: Perspectives on Broadening Participation During an Undergraduate Computing Enrollment Boom. *ACM Technical Symposium on Computer Science Education (SIGCSE)*

. Most universities at the time wouldn't admit Black students or women, and certainly not the Black women of color who had played such significant roles in operating and programming mainframes. For instance, Dartmouth University was dedicated to providing free computer access to all students, but these students were almost exclusively wealthy, White men who had already paid for the privilege of attending Dartmouth. At the Kiewit Computing Center, many Dartmouth students played games that focused on football (FOOTBALL) or warfare (SALVO42), centering stereotypically masculine expressions. Established men who worked as systems programmers at Kiewit would regularly prank and belittle novices, and the "arcane deta..s of one's programming expertise"

became a point of pride that one could use to exert power over less-versed peers[18]

[18] Joy Lisi Rankin (2018). Making a Macho Computing Culture. *A People's History of Computing Education in the United States*

.

This masculine style of computing spread to other New England colleges and high schools through Dartmouth's Kiewit Network. Some women in universities retained access to computers. For instance women at Mount Holyoke used the burgeoning Dartmouth network to correspond with men at Dartmouth, arranging dates while avoiding expensive long-distance calls[18]

[18] Joy Lisi Rankin (2018). Making a Macho Computing Culture. *A People's History of Computing Education in the United States*

. But men, especially White, wealthy men, enjoyed greater access. As computers became a widespread signal of social status, men, especially those at wealthy single-sex private schools, had access at much higher rates. Computing education was tied to mathematics, and, in general, men were encouraged to take math courses, while women were encouraged to become wives and homemakers. The history of women in computing and their role as computer programmers and operators was thus forgotten and replaced by a world shaped by wealthy elite men at Dartmouth specifically, and higher education broadly. This transition from mainframes to personal computers produced the male-dominated computing culture still found today in many CS learning contexts.

In other countries, such as India, women are encouraged to go into CS because it's seen as conforming to gender stereotypes about it being 'safe' and 'clean'

Jessie Huynh CCO

Private enterprise overtakes scientific endeavor

From hobby to marketplace

The transition from mainframe to hobbyist PC in academia produced a variety of operating systems, all borrowing liberally from each others' innovations. IBM created the System/360 operating system[1]

[1] Gene M. Amdahl, et al. (1964). Architecture of the IBM System/360. *IBM Journal of Research and Development*

, and was one of the first to allow multiple programs to be run on a single computer at the same time. Dozens of other operating systems emerged, each closely tied to a particular design of computer. AT&T's Bell Labs created an operating system named Unix for "mini" computers (so named because it was smaller than a room). Many of these research and

enterprise efforts inspired smaller consumer efforts, like Apple's operating system for the Lisa computer and Microsoft's Disk Operating System (MS-DOS), which mirrored other operating systems at the time. These personal computer operating systems grew rapidly, ultimately producing Apple's Mac OS and Microsoft's Windows. Meanwhile, researchers at the University of California, Berkeley, cloned Bell Lab's Unix, and created the Berkeley Software Distribution (BSD), which Apple used with Carnegie Mellon's research on the Mach Kernel to create it's modern version of Mac OS and iOS. Linus Torvalds, dissatisfied with the copyright restrictions in modifying these two commercial operating systems, created a clone of Bell Lab's Unix called Linux, which was open source, meaning that anyone could read or modify its source code. Google eventually built upon Linux to create the Android operating system.

This liberal copying and sharing was also accompanied by fierce, anti-competitive practices, as each company looked to establish a permanent foothold in the computing space through operating system platforms. AT&T only allowed AT&T devices to connect to their phone network. IBM used their market power to gain control over 70% of the punch card, tabulating machine, and computer markets. In the early 1990s, Microsoft bundled their applications with their operating system, and barred users from installing applications of competitors[6]

[6] Nicholas Economides & Ioannis Lianos (2009). The Elusive Antitrust Standard on Bundling in Europe and in the United States in the Aftermath of the Microsoft Cases. *Antitrust Law Journal*

. More recently, U.S. Department of Justice investigations into Amazon, Apple, Facebook, and Google have shown that all engaged in anti-competitive practices. Many of these business trends followed the

I mean, IBM started these markets, so...'used market power to build market power?'

all? Surely it was _some_ applications of competitors?

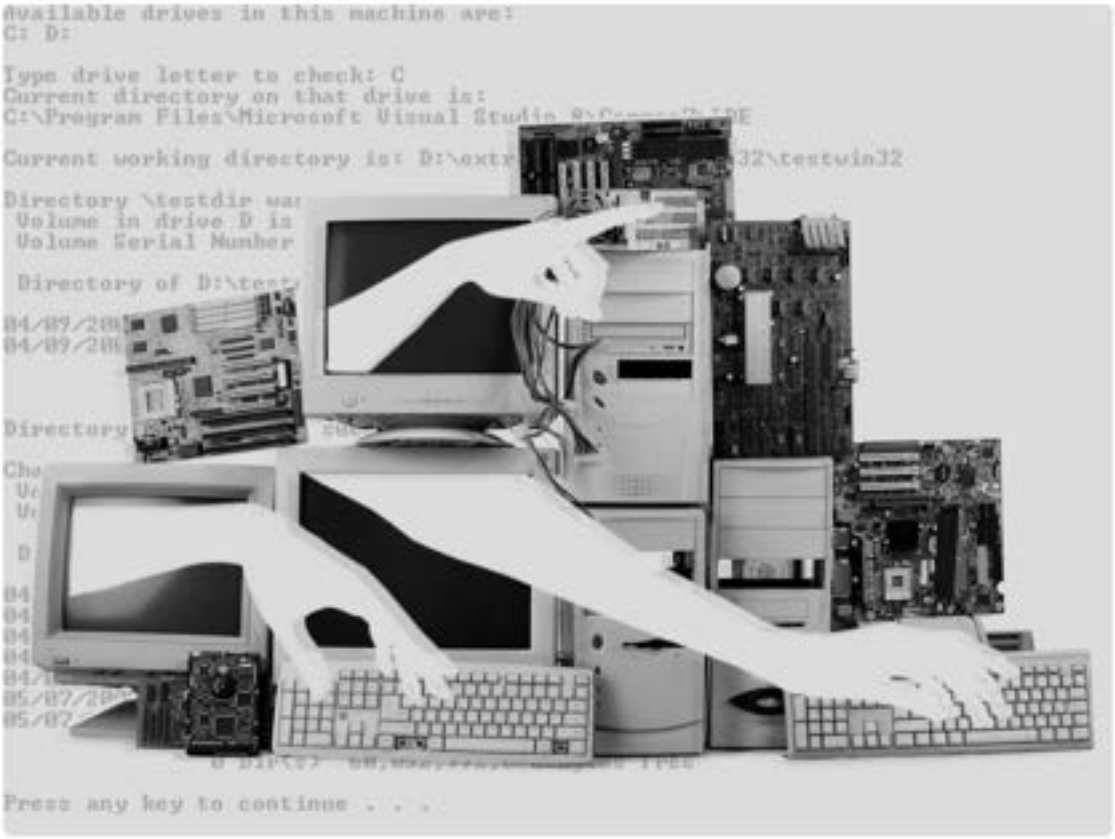
dominant economic ideology of the 1970's, often described as **neoliberalism**[9]

[9] David Harvey (2007). A Brief History of Neoliberalism. *Oxford University Press*

: a policy ideology that emphasized free-market policies through the elimination of government regulation, privatization of public services (e.g., the mainframes operated by NASA and universities) and deemphasized social reforms and supports.

These economic trends interacted closely with the creation of modern operating systems. Before this shift, merging with a major competitor, acquiring a smaller competitor before they could grow, and creating vertical monopolies would generally invite antitrust investigations. Today, decades of acquisitions, mergers, and vertical integration have changed the rich, complex, and diverse landscape of computer systems and operating systems of the 1960s into a sterile and stable landscape today, with only Microsoft, Apple, and Google dominating desktop and mobile operating systems, along with most of the internet that we interact with on a regular basis. All of these monopolies still have their roots in the earliest problems that human computer operators had, of trying to maximize use of a computer's CPU to execute programs, prioritize urgent jobs over others, and maintain the hardware utilized by the CPU. In fact, some of these operating systems still use source code written in the 1960's. The history of operating systems is therefore not just a series of events, but a set of artifacts, computer code, still used today by billions.

and



Jessie Huynh CC0

Computers are made of distinct parts, each formerly human work.

Components of modern operating systems

After these decades of innovation, competition, and monopolization, several stable components of operating systems emerged. Each component replaced the role of the human operator with an abstraction: something that enabled computer programmers to use computing hardware without having to understand the specifics of how that hardware was implicated, but in the process, also eliminated the rich abilities of human operators to understand the social context of how a computer was being used.

this seems a stretch - more complete OS dose not mean more fascy. Didn't we just go over these problems historically a few pages ago?

Perhaps the most important component is the kernel

kernel: The first program that runs when a computer boots, responsible for running all other computer programs, responding to input from users, and producing output.

. When a computer first starts, the kernel is the first program loaded into a computer's memory. It handles all requests for input and output from software, much like a human operator of a mainframe might insert punch cards (input) and pick up a printout to give to someone waiting for results (output). It manages the execution of instructions in software by the CPU, much like a human operator used to ensure that punch cards were inserted in the correct order, or fix a jam if a punch card crumpled because of too many holes. It handles communications between a computer and its input and output devices (keyboards, displays, printers, etc.), much like mainframe operators would check the connections between a mainframe and it's printer. The kernel, therefore, was like the brain of the human mainframe operator, making sure the computer continues to operate at full capacity, monitoring for issues that might arise, and fixing them if possible. If not possible, kernels can send exceptions to programs, signalling that an unrecoverable error might have occurred and giving the program an opportunity to recover. (Most of us experience exceptions as messages that say that an unknown error has occurred.) Kernels might also encounter issues that would force all operations to stop, and signal that to us as a **kernel panic**, though these are considerably more rare. Most of us experience kernel panics as a message on a computer's display that says that the computer must be restarted.[b]

[b] Many kernel panics in Windows are caused by devices breaking assumptions made by the kernel: the kernel gives devices some amount of execution time, the software running on the device runs for longer, the kernel notes that a promise was broken, and sends a panic. This is different from a "freeze", which is usually caused by multiple programs waiting for each other to finish a task or get access to a locked resource, and thus unable to proceed.

Kernels also manage memory

random access memory: A temporary place to store data while a computer is running; when the computer is shut off, it is erased.

memory, *RAM*

. Memory, as we discussed in the Chapter 8, is used to store computer programs while they are executing, as well as any data they are processing. Memory is, in the simplest terms, a long list of bytes, each byte a list of 8 bits. Each of these bytes is given an address, like a street address, meaning one can refer to a specific byte by a specific address. Modern computers may have many *gigabytes* of memory – 1 gigabyte is 1,000,000,000 bytes, but even that may not be enough for all of the applications a user wants to run. Therefore, the kernel's job is to coordinate memory usage between different programs, ensuring that one program doesn't overwrite data being used by another. Mainframe computers, while they did much of this memory management automatically too, still relied on human operators to coordinate memory usage and respond to situations where the computer ran out and needed more memory to complete a program.

Using the memory it manages, kernels are responsible for executing programs

program: A series of instructions for taking input, computing something, and producing output, including anything from a word processor, a game, or an operating system.

app, application

not all programs are written by humans

, which are a collection of instructions that human programmers write, much like the simple ones we showed in Chapter 6. (In modern terms, we

might also call these applications, or just "apps"). When a program is loaded into memory (just like mainframe operators loaded in punch cards), the kernel keeps track of information about the program executing, including where the executing program is stored, who's executing it, who has permission to see its results, what other resources it is using (files, external devices like a printer), the current program instruction being executed, and other details. This information is stored in a record of metadata called a process

process: Data stored by a kernel about a program's current execution state, including which instruction is being executed, what data it is using, who has permission to see its results.

. In the mainframe era, the human operator stored this information in their human memory, keeping track of who'd submitted the punch cards so they could return the cards and the program output to the right person, track the progress of a program executing, and resume it if it got interrupted or needed to be restarted; if there were a lot of programs to run, they might write down this information on a paper log.

While mainframes only had one process executing a time, a modern personal computer might be executing *thousands* of processes: every application we are running in parallel, as well as other background processes like messaging applications, timers, and anti-virus software. Operating systems, therefore, enable a higher degree of multitasking than a human operator of a mainframe would have been able to manage, and could be run 24/7 just as operators worked in shifts. An OS tries to give a user the same abstraction as a human operator – presenting your program, waiting for your program to be scheduled, and receiving results, but sometimes that abstraction breaks down. A process, for example,

might **hang**, getting stuck waiting for input, or trapped in a never ending infinite loop of data processing. Therefore, operating systems, much like their human counterparts, have to monitor for stuck processes and decide what to do with them, before they disrupt the execution of other processes. We often experience this now as an operating system interrupting us with a message like, "*This program is not responding; do you want to stop it?*", much like a human operator might have done with a program that didn't seem to be finishing on a mainframe.

As computers were utilized to perform more complex tasks, programs utilized more and more computer memory so storage devices

secondary storage: A semi-permantent place to store data such as files and operating systems; common media include hard drives, solid state drives, flash memory, and disks. When the computer is shut off, the data is not lost.

storage

such as magnetic tape, hard drives, and floppy disks proliferated. As these became more common, mainframe operators needed to manage the use of these storage devices, tracking which programs needed which storage devices. Even the storage devices themselves needed kernel help to keep data organized on the device, leading to the concept of a **file**, which is a collection of data with a name and location on the storage device. Modern software operating systems manage storage devices and files automatically, allowing programs to read from and write to any device that is connected to a computer. To do this, kernels have to manage lists of all of the devices that are connected, have device-specific software called **device drivers** for communicating with them, and keep track of which programs are using them in process meta-data, to avoid conflicts

when multiple programs are trying to use the same device at the same time.

As computers connected to the internet, operating systems also needed ways of managing data coming in from other computers, and managing data being sent to other computers. In the mainframe era, this would be like a mainframe operator receiving physical shipments of punch cards and storage devices from other computers via mail, deciding where to store them while it was decided what to do with them. Modern operating systems do the same thing with **network management**, receiving data, storing it in memory until it can be processed by a program, which might store it on a storage device (saving a copy of an email just received) or do some computation on it (rendering a web page that was just sent by another computer on a display).

Throughout all of these shifts from human operator to computer operating system, the interface

interface. A program and input and output devices that receives input from users and produces output, including command line interfaces, graphical user interfaces, and voice interfaces.

user interface, UI

to using a computer changed. In the mainframe era, one would say to the human operator, "*Can you run this program?*" and the human operator would say "*Sure, but there are a few more programs before you. Give me an hour?*" Operating systems first replaced this human interaction with **command line interfaces** (CLIs), where one would engage in a more restricted dialog with a computer. Fo. example, to run a program in Unix, one needs to simply navigate to the file folder containing the program

and then type its name (if one wanted to run the popular text editor vim, and programs were stored in a folder named programs, one would type `cd programs;` to open the programs folder, and `./vim` to run the program[c]

semicolon?

[c] The "./" preceding the program name signals that the program is located in the currently open folder, which is represented by a "." in Unix.

). These command line interfaces required users to remember many commands and follow their syntax precisely.

Early conceptions of computer interfaces, however, went well beyond command lines, and actually became the dominant way we interact with computers through operating systems, through **graphical user interfaces** (GUIs). The earliest visions of these interfaces came in 1945 by Vannevar Bush, who headed the U.S. Office of Scientific Research and Development[17]

[17] From Memex To Hypertext (1991). James M. Nyce, Nyce, Paul Kahn. *Elsevier Science*

. He imagined a machine called a Memex[4]

[4] Vannevar Bush (1945). As We May Think. *The Atlantic Monthly*

, which would allow people to freely explore a collection of documents, follow links between them, and create documents of their own. His notion of computers as a workstation for working with information was radically different from the mainframes at the time, which all required carefully constructed computer programs, written in programming languages. Programming was slow, laborious, and error prone, and Bush imagined computing to be something much more interactive and accessible than code. Bush's vision inspired a series of innovations, including Douglas

connection between NLS and demo could be confusing

Englebart's NLS system, which contributed a networked computer with keyboard, mouse, display, and applications[7]

[7] Douglas Engelbart (1962). Augmenting Human Intellect: A Conceptual Framework. *Bloomsbury Publishing*

. Englebart's demo inspired inventors at Xerox PARC, an industry research lab, who created the first GUIs and envisioned ideas like windows, scroll bars, buttons, files, folders, and applications[10]

[10] Michael A. Hiltzik (2009). Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age. *HarperCollins*

why do we say 'jobs brought to Apple' but then not 'Gates brought to microsoft?

. Xerox PARC's demonstrations inspired Steve Jobs, who brought the ideas to Apple Computer, which inspired Microsoft, and led to the ubiquity of GUIs that we all use today.

And interface innovation continues, including the voice interfaces often found in smart speakers, the many physical controls and feedback mechanisms found in modern cars, as well as gestural interfaces found in many video games and modern televisions.

Jessie Huynh CC0

Part of an operating system's job is keeping people out.

Operating systems as gatekeepers

This is the key point!

None of these design choices about how to organize an operating system were inevitable: operating system designers across history have had particular values, which served particular visions for how computers might be used.

The single most important value in the history of operating systems, much like the rest of CS, was efficiency

efficiency: The time it takes for a computer to compute something; more efficient computer hardware and operating systems execute more instructions per second. This can also refer to *memory* efficiency, which is how many bits of memory some computation requires to make its calculations.

speed

. This value stemmed from the underlying value behind the invention of computers, which was to replacing slower human computers with faster machines. Operating systems carried that value into the operation and maintenance of computers, replacing the slower and less reliable human operators of mainframes with significantly faster operating systems. Speed, of course, is in service of other values: faster computers are ultimately intended to make "faster" *people*, and faster people are ultimately intended to advance economic productivity in capitalist economic systems. It might be strange to imagine operating systems that value anything but speed, but that is more because of the ubiquity of the value than because other values are impossible to prioritize. For example, imagine an operating system that slowed users down to make them more mindful of their work, or an operating system that mandated work breaks to encourage users' physical health. Such operating systems can exist, they just don't, because they have been counter to operating system designers' core value of efficiency.

and more accurate!

I mean, outlook now nags me when I send email out of business hours to fake delay it till morning so that people can rest...

I would not like any of these suggestions. What about OS that prioritized privacy (actually see below!), or electricity usage, or beauty of the user experience?

Also IP protection is becoming a major design priority

Some values other than efficiency have emerged as secondary values in operating system design. For example, accessibility

accessibility: The extent to which a computer programs and user interfaces can be used by someone independent of their physical and cognitive abilities. For example, graphical user interfaces that require use of a mouse are not accessible to people with motor impairments that limit their ability perform fine movements of their hands.

a11y

– the extent to which people of varying abilities can use a computer and its applications – has largely been an afterthought in operating system design. Graphical user interfaces, for example, are a powerful and now ubiquitous idea for how to operate a computer, freeing many computer

Accessibility is better now than it ever has been with computers, and computers have expanded accessible options enormously over options before computers.

users from having to memorize complex commands, understand confusing error messages, and navigate an invisible hierarchy of folders and documents. In this sense, this concept for computer operation was a great success. However, requiring visual interaction with a computer fundamentally excluded people who are blind or have low vision. Therefore, tens of millions of people in the world cannot use modern operating systems without cumbersome screen reading software, which translates the visual contents of the operating system's display into spoken text. For many blind and low vision people, this is harder to use than a command line interface, because they can't as easily find and open programs and files, navigate the complex two-dimensional layouts of graphical user interfaces, or use interactions like drag and drop, which depend heavily on sight. And even command lines are harder to use than simply having a conversation with a human computer operator. Operating system designers therefore have immense power in deciding who can and can't easily operate a computer, and equally immense responsibility in ensuring that operating systems provide universal, diverse forms of access.

you just said text based OS had 'invisible hierarchy of folders' but then longed for them for blind or visually impaired people? Maybe a different choice of words?

but command lines are more precise! And yes, affordable and therefore economically accessible

Another value that has emerged in operating system design is privacy

privacy: The ability for individuals to control information about their identity and activity.

. As we discussed earlier in Encoding Information, computing is often used to gather private data without consent; operating systems are no exception, and are in fact a key enabler of privacy invasion. The Windows operating system, for example, tracks which programs you run, when you run them, and when they start and stop, and sends all of

that data to engineers in Redmond, Washington to be analyzed by data scientists for problems in the operating system, but also trends in how software is being used. This process is often referred to as **telemetry**, which refers to logs of all of the applications that people are running on their computers, when they are opened or closed, and when they crash or hang. Microsoft asks for your permission to be watched — but only once, when you first installed the operating system, and you probably didn't notice, since it was in an innocuous pop-up dialog prompt. The same is true of the Android operating system, which allows apps that run on it to gather all kinds of private information without first transparently asking for consent. Apple is somewhat more restrictive in its App Store, auditing programs before they are allowed to run on its operating system, in an effort to prevent private data from being shared with other companies, but only requires that apps comply with existing privacy laws, which typically means presenting users with a complicated privacy policy that must be accepted before using the app. Therefore, when we choose an operating system, we're not only making a choice about the speed, reliability, compatibility, usability, and accessibility of our computer, but also the trustworthiness of the company that has designed its operating system, requiring us to place great faith in its transparency around what data it collects about our computer use and how that data is used.

App store seems out of scope for this chapter, which already has a lot!

While operating systems themselves can erode privacy, they are also responsible for protecting it through security

security: Strategies that computers, operating systems, and programs use to preserve data privacy, such as passwords, biometric authentication, file permissions.

information assurance

The breathalizer code being bunk would be a great example to bring up here!

worked

. For example, the human computer mainframe operators working for the U.S. military in the 1960's, running programs that computed where a missile was to fire next. This information was highly sensitive; only certain people were to know that a missile was going to be fired and where, and only certain people were allowed to know the logic of that program. Even fewer were allowed access to the program itself, because maliciously modifying it would have led to incorrect trajectory calculations. Therefore, human mainframe operators had to be mindful of security policies, both on the programs themselves, the data that went into the programs, and the data that came out. The same is true of modern operating systems: if any program were able to modify the behavior of any other program or read its data, your data would no longer be private, leaving you vulnerable to identity theft, and disclosure of private communications. Vulnerabilities in operating systems include defects in the operating system software itself, which may allow people to gain access to everything on the computer; defects in how the operating system executes programs, which may allow one program (e.g., your email client) to access data used in another program (e.g., your banking application); or defects in the software used to build applications for a particular operating system. These vulnerabilities can be exploited with computer iruses, which are programs that use defects in operating systems to gain access to private data; but also with **malware**, which is software that users themselves install, without realizing that they are stealing private information, including **passwords**, which we all use to authorize access to our private data. Operating systems can also be kept secure by **encrypting** all data, which involves encoding it using a secret cipher

that only a user can unlock with a password, and by fixing security defects regularly. But even these measures are not perfect, as a user might disclose the password to someone malicious, or might not install software updates regularly. Therefore, many security measures are outside of an operating system's control: it is up to everyone to use unique passwords, keep those passwords secure, install updates, and be skeptical of applications that request passwords. When we fail to do such things, operating systems suffer major hacks, like the 2020 SolarWinds hack, where Russian hackers accessed SolarWind's security software, and secretly changed its behavior.

was this password related?

Because operating system providers determine the rules for which programs can run, they are also central in **speech** rights. For example, after the January 6, 2021 U.S. capital insurrection, many operating system providers decided to disallow particular programs from being installed. This included, most notably, the Parler app, which had been central in supporting the coordination of the insurrection. Apple removed it from its App Store, then Google removed it from the Android store. For those supportive of the insurrection, these decisions by operating system providers might be seen as censorship and a limiting of speech. For those critical of the insurrection, they might have seen these decisions as organizations leveraging their free speech rights to decide what kinds of information and programs would run on their private systems, drawing the line at those used to promote insurrection.

important to make clear dist\inction between free speech by govt and private companies: can this be stronger?

Because operating systems are so central in shaping how we use computers, and yet they have become so privatized and monopolized, operating systems are now gatekeepers, centralizing power, resulting in a single monolithic leverage point for determining policy about what programs and data are allowed on computers. Most operating systems privatize these decisions, shifting them from individual moral judgment or public policy to private decisions by CEOs and boards, shielded by software licenses that regulate who can use operating systems, and copyrights that protect the source code behind operating system policy. And, as we began this chapter discussing, they automate, replacing nuanced human intelligence with algorithms that promise speed, often ignoring human values. Therefore, far from being a mundane, almost invisible interface between hardware and software, operating systems are a center of power and conflict in our increasingly digital world.

private ones? linux doesnt have this problem?

solid point!

Jessie Huynh CC0

Operating systems, and therefore Microsoft, Google, and Apple, control nearly everything about our digital lives.

Teaching operating systems

Operating systems hold a unique position in computer science education. In higher education computer science degrees, it is sometimes a full required course, where students learn the many technical concepts above in greater depth. But it's rarely taught until after students have taken a few programming courses, and in many CS curricula, it is not required at all. And yet, the K-12 CS learning standards, broadly adopted around the world, do include some learning standards related to the hardware/software interface, including the different layers of abstraction

involved in hardware/software communication. This creates a bit of a tension: when should students learn about operating systems, if at all, and in what depth? And how might teaching at secondary and post-secondary levels differ?

One key difference between secondary and post-secondary is the audience. Most CS majors in college aspire to some kind of position designing and engineering computing technologies, and some depth of understanding about operating system implementation can be helpful in these positions. But in secondary, the goals might be different: a school might, for example, have a CS module that *all* students take at some point. Should operating systems be part of that instruction, and if so, how aligned should it be with the K-12 CS standards, which are strictly technical?

One argument that secondary should address operating systems universally is that from the first time a student learns to use a computer, whether at home, school, or a third place such as a library, they are *interacting* with an operating system, and learning its ideas of programs and apps, windows and multitasking, crashing and freezing. And indirectly, operating systems are shaping their rights and privileges, exposing them to corporate surveillance, marketing, and whatever content moderation policies operating system designers have built in. And so while post-secondary students might need a bit of depth into precisely what a kernel is and how to build o to avoid creating software that crashes, a secondary student might need a kind of critical operating

system literacy that connects the design choices that OS companies create to their digital lives.

Unfortunately, research has yet to offer much guidance on teaching operating systems in this way, or even in general. Most research has focused on post-secondary and follows the same basic pedagogy: engage students in creating or modifying operating system behavior, without critically examining the broader social context of operating system design[2,3,5,12]

[2] Jeremy Andrus & Jason Nieh (2012). Teaching operating systems using Android. *ACM Technical Symposium on Computer Science Education (SIGCSE)*

[3] Randal E. Bryant & David R. O'Hallaron (2001). Introducing computer systems from a programmer's perspective. *ACM Technical Symposium on Computer Science Education (SIGCSE)*

[5] Peter J. Desnoyers (2011). Teaching operating systems as how computers work. *ACM Technical Symposium on Computer Science Education (SIGCSE)*

[12] Oren Laadan, et al. (2011). Structured Linux kernel projects for teaching operating systems concepts. *ACM Technical Symposium on Computer Science Education (SIGCSE)*

. One exception to this is a study that taught hardware/software interface concepts socioculturally, through the metaphor of a house[11]

[11] Mara Kirdani-Ryan & Amy J. Ko (2022). The House of Computing: Integrating Counternarratives into Computer Systems Education. *ACM Technical Symposium on Computer Science Education (SIGCSE)*

. This approach to teaching operating systems centered students' critical consciousness of OS design choices and the values that shaped them, challenging students to reconsider those choices in line with their own values. This class was taught to CS majors in college, and yet most reported that they had never considered most of the choices that OS designers made, or that those choices could have individual or societal consequences.

Unit sketch: Teaching operating systems critically

Book club had a whole debate about design vs engineering and how much do the engineers have the ability to push back on bad design? Many decisions are product design level, engineers wouldn't have a say : however perhaps CS ought to have a system of professional ethics (like medicine?)

To explore the possibilities at the secondary level, here we present a unit sketch that illustrates one way of making the power of operating systems visible. The approach is to introduce a diversity of narratives around operating systems: students' own narratives, disciplinary computer science narratives, and then narratives around disability justice, privacy, and free speech. The unit ends with a discussion of policy, challenging students to reconcile these many different narratives and propose rules that they think should regulate (or deregulate) the power of operating systems. Throughout, the lessons engage the core components of operating systems at a conceptual level, linking them to the social issues.

The learning objectives for the unit are:

1. Students will be able to describe the major components of an operating system.
2. Students will be able to explain how operating systems create structural barriers to computer use by people with disabilities.
3. Students will be able to explain how operating system maintainers determine what data can and can't be gathered by computer programs.
4. Students will be able to explain how operating system maintainers determine what programs can and can't be run on a computer.

The first unit begins by stirring up a debate about students' OS preferences:

Session 1: My operating system is better

* Begin the session by describing operating systems as something we generally know by name: e.g., Windows, Mac OS, iOS, Android, Chrome OS.

* Divide the class into small groups of 2-3 and prompt them to discuss what operating systems they use, which they prefer, and why.

* Bring the class back together and elicit the preferences and reasons, writing names of operating systems in columns with reasons why they like it in the rows of columns. Students will likely share reasons such as liking particular apps that are only available on a platform, liking how they look, or liking particular features of a device.

* Identify the two most popular computing operating systems, then arrange the class in a philosophical chairs discussion format, dividing the class into two sides. For each facet in the brainstormed list (appearance, speed, functionality), ask for a student representing a side to come forward and defend their position.

* End the debate by observing that operating systems aren't better or worse in absolute terms, but better or worse at particular things, and that these things are shaped by the designers of the operating system.

* Give an overview of the coming sessions, connecting it with these different dimensions along which operating systems can be better or worse.

Students should leave the first session with a sense of identity around the operating system they defended, perhaps believing in their preference even more strongly. This creates a stronger contrast for later sessions, which complicate what operating systems are, who makes them, and the values that shape their choices.

The next session offers the first complicating narrative, the disciplinary one from computer science. To engage students in the relatively technical concepts, the lesson uses an "unplugged" approach that has students embody the components of an operating system.

Session 2: Operating system algorithms

* Characterize the first session's discussion as one particular consumer narrative about operating systems, shaped by our experiences with them as users. Contrast it with the topic of this session, which deconstructs operating systems from the designers' perspective, as something to be created and maintained to make computer applications work.
* Explain that the session will try to simulate a chat program executing alongside two other programs.
* Explain the concept of a kernel and applications, then ask four students to come to the front to act out the work of a kernel. One plays the role of the kernel, one plays the role of applications. The kernel's job is to "run" a few instructions of an application and then stop, switching to another application. The applications' job can be any multi-step activity, like a dance, or some pantomimed activity like making a sandwich.
* Explain the concept of a memory. Ask an additional four students to come to the front. The original four students keep

their roles. One of the new students is memory and the other three represent data A, B, and C. When the applications execute, they should now request data A, B, or C from the kernel, and the kernels should ask memory to retrieve the corresponding data.

* Explain the concept of a network interface. Ask two students' volunteer to be network interfaces, sending data that will be stored in A, B, and C. One sends a message, and the other receives it, giving it to the kernel, which decides whether to store in A, B, and C, and the applications, when they execute, can request the messages.

* Finally, explain the concept of a user interface. Ask one student to represent the screen that displays information. Applications can ask the kernel to display a message received; when that student receives a message, it yells it out.

* Thank everyone for modeling an operating system's behavior, then synthesize what everyone observed, summarizing the core components of the operating system, using an example from another application (such as presentation software) to step through the ideas. Explain why computers are often better at multitasking than people, because they can save the context of what they were doing exactly and come back to it later.

* Finally, ask the class how the computer science notion of an operating system relates to their experience with operating systems. How are they different? How are they related?

After the second session, students should have a clearer sense of their own personal perspective on operating systems, but also a sense of the

technical mechanics of an operating system as a mindless, procedural process that happens very quickly (meeting the first learning objective).

The next session begins to bridge the abstract technical understanding of operating systems with students' personal sense of operating systems by offering the perspective of people with disabilities.

Session 3: Accessibility

* Remind students that one part of an operating system is receiving inputs from a user interface, and providing outputs.
* **Formative assessment**. Ask students to brainstorm the different ways that computers allow us to provide input and receive output (e.g., mouse, keyboard, display, touchscreen, speakers). For each of the brainstormed input and output devices, ask students to brainstorm the assumptions that they make about our abilities (e.g, seeing, hearing, pointing, grasping). For each of the brainstormed assumptions, ask students to speculate how someone who lacks the ability would provide input to a computer, or receive output. Engage students in shaping where the brainstorms will be captured and shared.
    * This is *responsive* because it asks students to reflect on their own abilities and the abilities of the people around them.
    * This is *participatory* because it gives students agency in shaping the collective brainstorming project.
    * This is *educative* because students may only see some assumptions from their own view, but can learn of others from their peers' diverse perspectives.
* Begin a socratic seminar, asking whether it is fair that operating systems exclude people with particular disabilities, and why operating system designers might have decided to exclude.

* Explain that more modern operating systems try to include some people with disabilities, then give a demonstration of the screen reading software built into an operating system (e.g., Mac OS or iOS VoiceOver, Google Talk Back).
* Continue the Socratic seminar, asking whether screen readers adequately include people who are blind or have low vision.
* End by explaining that some operating systems are better than others at supporting different disabilities, giving examples of operating systems that provide high levels of support (e.g., Apple), and operating systems that provide minimal support (e.g., Google).

After this session, students should begin to recognize that operating systems are more than just apps and features, but also more than just algorithms: they are also the gateways between people and applications, determining who can and cannot access them. Students should begin to wonder whether their original preference for an operating system incorporated other people's experiences, meeting the second learning objective.

The next session builds upon this more complicated narrative by raising questions about privacy and security.

Session 4: Privacy and security

* Remind students about session 2's demonstration of an operating system executing a chat application, and how information was passed from the network interface to the kernel, and then from the kernel to memory, and then finally the program.

* Pose the question: what if that information was private, like most chats are? What stops someone from seeing the private message? Ask students to speculate about the role the kernel, applications, memory, network interfaces, and user interfaces play.

* Discuss how companies that create operating systems have access to our data and often use it to monitor our activities, personalize advertisements, and sometimes share that data with the government to surveil our activities.

* Shift from privacy to security, explaining that security is about ensuring only people with permission to see information can access it, privacy is about who has that permission.

* Explain that all components play a role in keeping the information secure. Provide direct instruction on examples where part of an operating system was not secure (e.g., a video on the SolarWinds hack), explaining what kind of data was breached, and what the consequences of that might be. Connect the break to software updates, which can repair security defects, but also create security defects.

* Poll the students on whether they update their phones when there is an update. Discuss why they do or don't, and discuss the consequences of that on privacy (e.g., release of secrets, identity theft).

* Return to the first session's debate of operating systems, and raise the question: how does one know if an operating system is secure? What makes one trust an operating system?

* End the session showing that all companies have had data breaches, showing examples of data breaches of the two most popular operating systems in the class.

Building upon students' conception of operating systems as both technical and social, this session should reconnect it to students' personal data, and raise questions of trust in operating systems and operating system maintainers. Students should begin to wonder about what operating system software updates are doing to their devices, and where they are getting information about which operating systems to trust (meeting the third learning objective).

The next session presents the last narrative, connecting operating systems to speech.

Session 5: Free speech

* Recall the discussion about privacy and security, and connect it to an application that is privacy-invasive, such as Facebook, WeChat, or TikTok. Explain how those applications invade privacy.
* Explain that on January 6, 2021, some operating system maintainers (Apple, Google, Amazon), decided to remove some applications from their platforms because they were used to incite a violent insurrection on the U.S. capital. Frame those decisions as a choice about what applications would be allowed to run on each operating system.
* Remind students of free speech laws in the United States, which bar the U.S. government from limiting speech, except in

particular circumstances (speech that harms, speech that is obscene, speech that defames).

* Summative assessment. Begin a philosophical chairs discussion with the question: should operating system companies have the right to decide which applications run on their operating systems? Discuss with students how they want to organize presentations and how they want to judge each others' presentations. Then facilitate discussion about both sides of this issue, one the position that companies are protected by free speech law to decide to disallow an application, and the other position that the companies are limiting speech by removing applications that support speech. Focus the discussion on who should decide.
    * This is *responsive* because it centers students' interaction with the operating systems in their family and school's devices.
    * This is *participatory* because it gives students agency in shaping the terms of debate success.
    * This is *educative* because it reveals diverse student perspectives on the role of private companies in speech.
* After the discussion, break the class into small groups, and prompt them to draft a new law that would clarify this debate about who decides. They should draft some language, and a justification for the rule.
* Students then present their laws and rationale and students vote on whether they would pass the law.

This last session meets the fourth learning objective, while also linking operating systems to law, policy, free speech, and power. Students should

leave this lesson seeing the operating systems on the computers that they use as far more than colors on a screen, and an exclusionary gateway to computer applications, but also a gateway controlled by powerful technology companies that are not currently beholden to the public in any way (and continue to utilize their wealth to maintain their power to decide).

While this unit gives a basic introduction to operating systems concepts, it does not attempt to build a robust understanding of operating systems algorithms or issues. This might leave students fluent with the basic concept of an operating system, but it will not develop the kind of detailed technical understanding needed to even troubleshoot operating system problems, let alone discuss the nuanced differences between operating system designs. Such knowledge is not part of most learning standards, though it does appear in higher education computer science courses on operating systems, and is even discussed in popular technology journalism about operating systems. Instead of depth, the unit focuses on critically conscious breadth, helping students see the many diverse and surprising ways that operating systems connect to society.

Conclusion: the kernel of computing culture

In some ways, operating systems are a remarkable feat of engineering: they transformed what once was a highly secure, highly inaccessible scarce resource – the mainframe – into world full of billions of little

personal computing devices, each "operated" by one or more people, often without incident. And all of this digital world, with all of its wonders and all of its harms, was only possible by replacing the small group of mostly women who tended to this large machines with an even smaller group of complex operating system programs — Windows, mac OS, Unix, Linux, Android, iOS — carefully engineered over the course of a half century to provide near limitless computing on demand, anywhere. From an engineering standpoint, this is profound, incomprehensible social impact at a scale rarely seen.

The price we've paid for this digital world comes in multiple currencies. The history of operating system development in academia and industry has given us a CS education culture often centered in masculinity, elitism, exclusion, and shame. The capitalism that fueled OS creation has grown a handful of corporations, now the most wealthy and powerful in the world, who shape our digital experiences with little oversight and accountability in the world, let alone the nations in which they reside. And the complexity of these operating systems have produced a public that is largely unaware of this centralized, privatized power and its influence over our daily lives. Teachers are one of the few in a position to make this history and status quo visible to students, in ways that are culturally and critically situated.

Relevant learning standards

This chapter covered the concepts in the following learning standards:

CSTA Learning Standards	**Original Standard**	***Critically Conscious Revision***
Impacts of Computing		
2-IC-20	Compare tradeoffs associated with computing technologies that affect people's everyday activities and career options.	*Examine power imbalances in the design of computing systems that create, amplify, and reinforce inequities and injustices in society.*
2-IC-23	Describe tradeoffs between allowing information to be public and keeping information private and secure.	*Describe individual and collective tradeoffs of surveillance capitalism.*
3A-IC-24	Evaluate the ways computing impacts personal, ethical, social, economic, and cultural practices.	*Critique how computing amplifies, centralizes, privatizes, and automates social processes in society, impacting individuals, communities, and culture.*
3A-IC-29	Explain the privacy concerns related to the collection and generation of data through automated processes that may not be evident to users.	*Explain how organizations use software to surveil users in order to leverage their activity data for profit and power.*
3A-IC-30	Evaluate the social and economic implications of privacy in the context of safety, law, or ethics.	*Evaluate the interaction between privacy, computing, and power.*
3B-IC-27	Predict how computational innovations that have revolutionized aspects of our culture might evolve.	*Predict how computational innovations will shape culture, power, and equity in global society.*
Networks & the Internet		
2-NI-05	Explain how physical and digital security measures protect electronic information.	*Explain how physical and digital security measures trade usability for data protection, and who controls those measures.*

2-NI-06	Apply multiple methods of encryption to model the secure transmission of information.	*Apply multiple methods of encryption to secure data, while examining how encryption can be a tool for both safety and extortion.*
3A-NI-05	Give examples to illustrate how sensitive data can be affected by malware and other attacks.	*Give examples to illustrate how sensitive data can be exploited by malware, ransomware, and harassers.*
3A-NI-06	Recommend security measures to address various scenarios based on factors such as efficiency, feasibility, and ethical impacts.	*Recommend security measures appropriate for scenarios with varying balances of power.*
3A-NI-07	Compare various security measures, considering tradeoffs between the usability and security of a computing system.	*Compare various security measures, considering tradeoffs between the usability, security, and accessibility of a computing system.*
3A-NI-08	Explain tradeoffs when selecting and implementing cybersecurity recommendations.	*Explain tradeoffs to who holds power to data and services when selecting and implementing cybersecurity recommendations.*
3B-NI-04	Compare ways software developers protect devices and information from unauthorized access.	*Compare the learnability, usability, accessibility, and security of various authorization techniques.*

Algorithms and Programming

3B-AP-18	Explain security issues that might lead to compromised computer programs.	*Explain how security vulnerabilities might lead to individual, community, and sociopolitical consequences.*
3B-AP-20	Use version control systems, int[illegible]ated development environments (IDEs), and collaborative tools and practices	*Use version control, IDEs, documentation, and collaboration tools to faciliate community-based,*

	(code documentation) in a group software project.	*collaborative, iterative software design and development.*
Computing Systems		
2-CS-02	Design projects that combine hardware and software components to collect and exchange data.	*Design and critique projects that combine hardware and software to gather, structure, analyze and store data.*
3A-CS-02	Compare levels of abstraction and interactions between application software, system software, and hardware layers.	*Examine how levels of abstraction in operating systems and hardware shape and constrain what applications are created and who can use them.*
3B-CS-01	Categorize the roles of operating system software.	*Explain how distinct functions within operating system software create barriers to who can access and use computers and software.*

Social Justice Standards	**Original Standard**	**Computing Revision**
Justice		
2	Students will develop language and historical and cultural knowledge that affirm and accurately describe their membership in multiple identity groups.	*Students will develop language and historical and cultural knowledge that affirm and accurately describe their membership in multiple identity groups by examining how software excludes.*

5	Students will recognize traits of the dominant culture, their home culture and other cultures and understand how they negotiate their own identity in multiple spaces.	*Students will recognize traits of the dominant culture, their home culture, and other cultures in computing artifacts and understand how they negotiate their own identity in computing spaces.*
14	Recognize that power and privilege influence relationships on interpersonal, intergroup, and institutional levels and consider how they have been affected by those dynamics.	*Recognize that the power and privilege imbued into computing influences relationships on interpersonal, intergroup, and institutional levels and consider how they have been affected by those dynamics.*
15	Identify figures, groups, events and a variety of strategies and philosophies relevant to the history of social justice around the world.	*Identify figures, groups, events and a variety of strategies and philosophies relevant to the computing and social justice.*
18	Students will speak up with courage and respect when they or someone else has been hurt or wronged by bias.	*Students will speak up with courage and respect when they or someone else has been hurt or wronged by algorithmic or data bias.*

CSTA Teacher Standards	**Original Standard**	**Critically Conscious Revision**
CS Knowledge and Skills		
1a	Apply CS practices.	*Apply CS practices in ways that center equity and justice for marginalized groups.*

1b	Apply knowledge of computing systems.	*Develop critical consciousness of computing systems knowlege.*
1c	Model networks and the Internet.	*Explain how the internet shapes its accessibility, access, and impact on society.*
1f	Analyze impacts of computing.	*Analyze the interaction between computing, power, oppression, and justice.*
Equity and Inclusion		
2a	Examine issues of equity in CS.	*Examine issues of equity and justice in CS.*
2b	Minimize threats to inclusion.	*Create culturally responsive and sustaining learning environments for all students.*
2c	Represent diverse perspectives.	*Make space for diverse perspectives, values, and assets from both students and broader society.*
Professional Growth and Identity		
3b	Model continuous learning.	*Learn alongside students, modeling sociotechnical humility, vulnerability, and curiosity.*
Instructional Design		
4c	Design inclusive learning experiences.	*Design culturally responsive and sustaining learning experiences that advance justice.*
4e	Plan projects that have personal meaning to students.	*Situate CS learning in students' identities, values, goals, and communities.*
4f	Plan instruction to foster student understanding.	*Co-construct learning and assessment to foster student interest, identity, and agency.*
4g	Inform instruction through assessment.	*Co-design culturally responsive, participatory, and educative formative assessments to support learning.*

Classroom Practice

5a	Use inquiry to faciliate student learning.	*Use inquiry and discourse to faciliate students' critical consciousness.*
5b	Cultivate a positive classroom climate.	*Ensure all students feel safe, supported, valued, and heard.*
5c	Promote student self-efficacy.	*Center student agency, assets, values, and culture.*
5d	Support student collaboration.	*Center student collaboration and discourse to foster critical consciousness.*
5e	Encourage student communication.	*Encourage student communication, reflection, writing, and speaking about CS equity and justice.*
5f	Guide students' use of feedback.	*Guide students to both seek and learn from feedback, as well provide it to those with power.*

References

1. Gene M. Amdahl, Gerrit A. Blaauw, and Frederick P. Brooks (1964). Architecture of the IBM System/360. *IBM Journal of Research and Development*.

2. Jeremy Andrus, Jason Nieh (2012). Teaching operating systems using Android. *ACM Technical Symposium on Computer Science Education (SIGCSE)*.

3. Randal E. Bryant, David R. O'Hallaron (2001). Introducing computer systems from a programmer's perspective. *ACM Technical Symposium on Computer Science Education (SIGCSE)*.

4. Vannevar Bush (1945). As We May Think. *The Atlantic Monthly*.

5. Peter J. Desnoyers (2011). Teaching operating systems as how computers work. *ACM Technical Symposium on Computer Science Education (SIGCSE)*.

6. Nicholas Economides, Ioannis Lianos (2009). The Elusive Antitrust Standard on Bundling in Europe and in the United States in the Aftermath of the Microsoft Cases. *Antitrust Law Journal*.

7. Douglas Engelbart (1962). Augmenting Human Intellect: A Conceptual Framework. *Bloomsbury Publishing*.

8. Thomas Haigh, Peter Mark Priestley, Mark Priestley, Crispin Rope (2016). ENIAC in Action: Making and Remaking the Modern Computer. *MIT Press*.

9. David Harvey (2007). A Brief History of Neoliberalism. *Oxford University Press*.

10. Michael A. Hiltzik (2009). Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age. *HarperCollins*.

11. Mara Kirdani-Ryan, Amy J. Ko (2022). The House of Computing: Integrating Counternarratives into Computer Systems Education. *ACM Technical Symposium on Computer Science Education (SIGCSE)*.

12. Oren Laadan, Jason Nieh, Nicolas Viennot (2011). Structured Linux kernel projects for teaching operating systems concepts. *ACM Technical Symposium on Computer Science Education (SIGCSE)*.

13. Kathleen J. Lehman, Julia Rose Karpicz, Veronika Rozhenkova, Jamelia Harris, and Tomoko M. Nakajima (2021). Growing Enrollments Require Us to Do More: Perspectives on Broadening Participation During an Undergraduate Computing Enrollment Boom. *ACM Technical Symposium on Computer Science Education (SIGCSE)*.

14. Steven Lubar (1992). "Do Not Fold, Spindle or Mutilate": A Cultural History of the Punch Card. *Journal of American Culture*.

15. Jane Margolis, Allan Fisher (2003). Unlocking the Clubhouse: Women in Computing. *MIT Press*.

16. R. Arvid Nelsen (2016). Race and computing: The problem of sources, the potential of prosopography, and the lesson of Ebony Magazine. *IEEE Annals of the History of Computing*.

17. From Memex To Hypertext (1991). James M. Nyce, Nyce, Paul Kahn. *Elsevier Science*.

18. Joy Lisi Rankin (2018). Making a Macho Computing Culture. *A People's History of Computing Education in the United States*.

19. Andrew S. Tanenbaum, Herbert Bos (2015). Modern Operating Systems. *Pearson*.

✓ Saved