Operating Systems (OS)

Preamble

Operating system is the collection of services needed to safely interface the hardware with applications. Core topics focus on the mechanisms and policies needed to virtualize computation, memory, and I/O. Overarching themes that are reused at many levels in computer systems are well illustrated in operating systems (e.g. polling vs interrupts, caching, flexibility costs overhead, similar scheduling approaches to processes, page replacement, etc.). OS should focus on how those concepts apply in other areas of CS - trust boundaries, concurrency, persistence, safe extensibility.

Operating systems remains an important Computer Science Knowledge Area in spite of how OS functions may be redistributed into computer architecture or specialized platforms. A CS student needs to have a clear mental model of how a pipelined instruction executes to how data scope impacts memory location. Students can apply basic OS knowledge to domain-specific architectures (machine learning with GPUs or other parallelized systems, mobile devices, embedded systems, etc.). Since all software must leverage operating systems services, students can reason about the efficiency, required overhead and the tradeoffs inherent to any application or code implementation. The study of basic OS algorithms and approaches provides a context against which students can evaluate more advanced methods. Without an understanding of sandboxing, how programs are loaded into processes, and execution, students are at a disadvantage when understanding or evaluating vulnerabilities to vectors of attack.

Changes since CS 2013

The core of operating systems knowledge from CC2013 has been propagated from CC2023 to the updated knowledge area. Changes from CC2013 include moving of File systems knowledge (now called File Systems API and Implementation) and Device Management to KA Core from elective and Performance and Evaluation knowledge units to the Systems Fundamentals Knowledge area. The addition of persistent data storage and device I/O reflects the impact of file storage and device I/O limitations on the performance (e.g. parallel algorithms, etc.). More advanced topics in File Systems API and Implementation and Device Management were moved to a new Knowledge Unit Advanced File Systems. The Performance and Evaluation knowledge unit moved to Systems Fundamentals with the idea that performance and evaluation approaches for operating systems are mirrored at other levels and are best presented in this context.

Systems programming and creation of platform specific executables are operating systems related topics as they utilize the interface provided by the operating system. These topics are

listed as knowledge units within the Foundations of Programming languages (FPL) knowledge area because they are also programming related and would benefit from that context.

Overview

“Role and purpose of operating systems” and “Principles of operating systems” provide a high-level overview of traditional operating systems responsibilities. Required computer architecture mechanisms for safe multitasking and resource management are presented. This provides a basis for application services needed to provide a virtual processing environment. These items are in the CS Core because they enable reasoning on possible security threat vectors and application performance bottlenecks.

“Concurrency” CS Core topics focus on programming paradigms that are needed to share resources within and between operating systems and applications. “Concurrency” KA Core topic provides enough depth into concurrency primitives and solution design patterns so that students can evaluate, design, and implement correct parallelized software components. Although many students may not become operating systems developers, parallel components are widely used in specialized platforms and GPU-based machine learning applications. Non-core topics focus on emerging concepts and examples where there is more integration between architecture, operating systems functions and application software to improve performance and safety.

“Protection and security” CS Core overlaps the dedicated Security Knowledge Area. However, operating systems provide a unique perspective that considers the lower level mechanisms that must be secured for safe system function. “Protection and security” KA Core extends the CS Core topics to operating systems access and control functions that are available to applications and end-users. Non-core focuses on advanced security mechanisms within specific operating systems as well as emerging topics.

“Scheduling”, “Process model”, “Memory Management”, “Device management” and “File systems API and Implementation” KA Core provide depth to the CS Core topics. They provide the basis for virtualization and safe resource management. The placement of these topics in the KA Core does not reduce their importance. It is expected that many of these topics will be covered along with the CS Core topics. Non-core topics focus on emerging topics and provide additional depth to the KA Core topics.

“Society, Ethics and Professionalism” KA Core focuses on open source and life cycle issues. These software engineering issues are not the sole purview of operating systems as they also exist for specialized platforms and applications level knowledge areas.

“Advanced File Systems”, “Virtualization”, “Real-time and Embedded Systems”, and “Fault tolerance” KA Core and Non Core include advanced topics. These topics overlap with “Specialized Platform”, “Architecture”, “Parallel and Distributed Systems” and “Systems Fundamentals” Knowledge Areas.

Core Hours

Knowledge Units	CS Core	KA Core
Role and Purpose of Operating Systems	2	
Principles of Operating System	2	
Concurrency	2	1
Protection and Safety	2	1
Scheduling		1
Society, Ethics and Professionalism (Hours included in SEP		
Process Model		1
Memory Management		2
Device Management		1
File Systems API and Implementation		2
Virtualization		3
Real-time and Embedded Systems		2
Fault Tolerance		3

Knowledge Units

OS-Purpose: Role and purpose of the operating system

CS Core:

1. Operating system as mediator between general purpose hardware and application-specific software
   Example concepts: Operating system as an abstract virtual machine via an API)
2. Universal operating system functions
   Example concepts:
   a. Interfaces (process, user, device, etc)
   b. Persistence of data
3. Extended and/or specialized operating system functions (Example concepts: Embedded systems, Server types such as file, web, multimedia, boot loaders and boot security)

4. Design issues (e.g. efficiency, robustness, flexibility, portability, security, compatibility, power, safety) Example concepts: Tradeoffs between error checking and performance, flexibility and performance, and security and performance
5. Influences of security, networking, multimedia, parallel and distributed computing
6. Overarching concern of security/protection: Neglecting to consider security at every layer creates an opportunity to inappropriately access resources.
   Example concepts:
   a. Unauthorized access to files on an unencrypted drive can be achieved by moving the media to another computer,
   b. Operating systems enforced security can be defeated by infiltrating the boot layer before the operating system is loaded and
   c. Process isolation can be subverted by inadequate authorization checking at API boundaries
   d. Vulnerabilities in system firmware can provide attack vectors that bypass the operating system entirely
   e. Improper isolation of virtual machine memory, computing, and hardware can expose the host system to attacks from guest systems
   f. The operating system may need to mitigate exploitation of hardware and firmware vulnerabilities, leading to potential performance reductions (e.g. Spectre and Meltdown mitigations)

7. Exposure of operating systems functions in shells and systems programming (See also: FPL-Scripting)

Illustrative Learning Outcomes:

CS Core:

1. Understand the objectives and functions of modern operating systems
2. Evaluate the design issues in different usage scenarios (e.g. real time OS, mobile, server, etc)
3. Understand the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve
4. Understand how evolution and stability are desirable and mutually antagonistic in operating systems function

OS-Principles: Principles of operating systems

CS Core:

1. Operating system software design and approaches such as Monolithic, Layered, Modular, Micro-kernel models and Unikernel
2. Abstractions, processes, and resources
3. Concept of system calls and links to application program interfaces (APIs)(See also: AR-C: Assembly Level Machine Organization)
   Example concepts:

   a. APIs (Win32, Java, Posix, etc) bridge the gap between highly redundant system calls and functions that are most aligned with the requests an application program would make
   b. Approaches to syscall ABI (Linux "perma-stable" vs. breaking ABI every release).
4. The evolution of the link between hardware architecture and the operating system functions
5. Protection of resources means protecting some machine instructions/functions (See also: AR-C: Assembly Level Machine Organization)
   Example concepts
   a. Applications cannot arbitrarily access memory locations or file storage device addresses
   b. Protection of coprocessors and network devices
6. Leveraging interrupts from hardware level: service routines and implementations (See also: AR-C: Assembly Level Machine Organization)
   Example concepts
   a. Timer interrupts for implementing timeslices
   b. I/O interrupts for putting blocking threads to sleep without polling
7. Concept of user/system state and protection, transition to kernel mode using system calls (See also: AR-C: Assembly Level Machine Organization)
8. Mechanism for invoking of system calls, the corresponding mode and context switch and return from interrupt (See also: AR-C: Assembly Level Machine Organization)
9. Performance costs of context switches and associated cache flushes when performing process switches in Spectre-mitigated environments

Illustrative Learning Outcomes

CS Core:

1. Understand how the application of software design approaches to operating systems design/implementation (e.g. layered, modular, etc) affects the robustness and maintainability of an operating system
2. Categorize system calls by purpose
3. Understand dynamics of invoking a system call (passing parameters, mode change, etc)
4. Evaluate whether a function can be implemented in the application layer or can only be accomplished by system calls
5. Apply OS techniques for isolation, protection and throughput across OS functions (e.g. starvation similarities in process scheduling, disk request scheduling, semaphores, etc) and beyond
6. Understand how the separation into kernel and user mode affects safety and performance.
7. Understand the advantages and disadvantages of using interrupt processing in enabling multiprogramming
8. Analyze for potential vectors of attack via the operating systems and the security features designed to guard against them

OS-Concurrency: Concurrency

CS Core:

1. Thread abstraction relative to concurrency

2. Race conditions, critical regions (role of interrupts if needed)(See also: PDC-A: Programs and Execution)
3. Deadlocks and starvation
4. Multiprocessor issues (spin-locks, reentrancy)
5. Multiprocess concurrency vs. multithreading

KA Core:

6. Thread creation, states, structures(See also: SF-B: Basic Concepts)
7. Thread APIs
8. Deadlocks and starvation (necessary conditions/mitigations)
9. Implementing thread safe code (semaphores, mutex locks, cond vars) (See also: AR-G: Performance and Energy Efficiency, SF-E: Performance Evaluation)
10. Race conditions in shared memory (See also: PDC-A: Programs and Execution)

Non-Core:

11. Managing atomic access to OS objects Example concept: Big kernel lock vs. many small locks vs. lockless data structures like lists

Illustrative Learning Outcomes

CS Core:

1. Understand the advantages and disadvantages of concurrency as inseparable functions within the operating system framework
2. Understand how architecture level implementation results in concurrency problems including race conditions
3. Understand concurrency issues in multiprocessor systems

KA Core:

4. Understand the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each
5. Understand techniques for achieving synchronization in an operating system (e.g., describe how a semaphore can be implemented using OS primitives) including intra-concurrency control and use of hardware atomics
6. Accurately analyze code to identify race conditions and appropriate solutions for addressing race conditions

OS-Protection: Protection and Safety

CS Core:

1. Overview of operating system security mechanisms (See also: SEC-A: Foundational Security)
2. Attacks and antagonism (scheduling, etc) (See also: SEC-A: Foundational Security)
3. Review of major vulnerabilities in real operating systems (See also: SEC-A: Foundational Security)
4. Operating systems mitigation strategies such as backups (See also: SF-F: System Reliability)

KA Core:

5. Policy/mechanism separation (See also: SEC-F-Security Governance)
6. Security methods and devices (See also: SEC-F-Security Governance)
   Example concepts:
   a. Rings of protection (history from Multics to virtualized x86)
   b. x86_64 rings -1 and -2 (hypervisor and ME/PSP)
7. Protection, access control, and authentication (See also: SEC-F-Security Governance)

Illustrative Learning Outcomes

CS Core:

1. Understand the requirement for protection and security mechanisms in an operating systems
2. List and describe the attack vectors that leverage OS vulnerabilities
3. Understand the mechanisms available in an OS to control access to resources

KA Core:

4. Summarize the features and limitations of an operating system that impact protection and security

OS-Scheduling: Scheduling

KA Core:

1. Preemptive and non-preemptive scheduling
2. Schedulers and policies. Example concepts: First come, first serve, Shortest job first, Priority, Round Robin, and Multilevel **(**See also: SF-C: Resource Allocation and Scheduling)
3. Concepts of SMP/multiprocessor scheduling and cache coherence (See also: AR-C: Assembly Level Machine Organization)
4. Timers (e.g. building many timers out of finite hardware timers) (See also: AR-C: Assembly Level Machine Organization)
5. Fairness and starvation

Non-Core:

6. Subtopics of operating systems such as energy-aware scheduling and real-time scheduling (See also: AR-G: Performance and Energy Efficiency, SPD-Embedded, SPD-Mobile 5-D?-)
7. Cooperative scheduling, such as Linux futexes and userland scheduling

Illustrative Learning Outcomes

KA Core:

1. Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes
2. Understand relationships between scheduling algorithms and application domains
3. Understand each types of processor schedulers such as short-term, medium-term, long-term, and I/O

4. Evaluate a problem or solution to determine appropriateness for asymmetric and/or symmetric multiprocessing.
5. Evaluate a problem or solution to determine appropriateness as a processes vs threads
6. Understand the need for preemption and deadline scheduling

Non-Core:

7. Understand the ways that the logic embodied in scheduling algorithms is applicable to other operating systems mechanisms, such as first come first serve or priority to disk I/O, network scheduling, project scheduling, and problems beyond computing

OS-Process: Process Model

KA Core:

1. Processes and threads relative to virtualization-Protected memory, process state, memory isolation, etc
2. Memory footprint/segmentation (stack, heap, etc)(See also: AR-C: Assembly Level Machine Organization)
3. Creating and loading executables and shared libraries**(**See also: FPL-H: Language Translation and Execution or Systems Interaction)

    Examples:

    a. Dynamic linking, GOT, PLT
    b. Structure of modern executable formats like ELF

4. Dispatching and context switching (See also: AR-C: Assembly Level Machine Organization)
5. Interprocess communication (See also: PDC-B: Communication)

Example concepts: Shared memory, message passing, signals, environment variables, etc

Illustrative Learning Outcomes

KA Core:

1. Understand how processes and threads use concurrency features to virtualize control
2. Understand reasons for using interrupts, dispatching, and context switching to support concurrency and virtualization in an operating system
3. Understand the different states that a task may pass through and the data structures needed to support the management of many tasks
4. Understand the different ways of allocating memory to tasks, citing the relative merits of each
5. Apply the appropriate interprocess communication mechanism for a specific purpose in a programmed software artifact

OS-Memory: Memory Management

KA Core:

1. Review of physical memory, address translation and memory management hardware(See also: AR-D: Memory Hierarchy)
2. Impact of memory hierarchy including cache concept, cache lookup, etc on operating system mechanisms and policy (See also: AR-D: Memory Hierarchy, SF-D: System Performance)
   Example concepts:
   a. CPU affinity and per-CPU caching is important for cache-friendliness and performance on modern processors
3. Logical and physical addressing, address space virtualization(See also: AR-D: Memory Hierarchy)
4. Concepts of paging, page replacement, thrashing and allocation of pages and frames
5. Allocation/deallocation/storage techniques (algorithms and data structure) performance and flexibility
   Example concepts:
   a. Arenas, slab allocators, free lists, size classes, heterogeneously sized pages (hugepages)
6. Memory caching and cache coherence and the effect of flushing the cache to avoid speculative execution vulnerabilities(See also: AR-F: Functional Organization, AR-D: Memory Hierarchy, SF-D: System Performance)
7. Security mechanisms and concepts in memory mgmt including sandboxing, protection, isolation, and relevant vectors of attack
   Non-Core:
8. Virtual Memory: leveraging virtual memory hardware for OS services and efficiency

Illustrative Learning Outcomes

KA Core:

1. Explain memory hierarchy and cost-performance trade-offs
2. Summarize the principles of virtual memory as applied to caching and paging
3. Evaluate the trade-offs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed
4. Describe the reason for and use of cache memory (performance and proximity, how caches complicate isolation and VM abstraction)
5. Code/Develop efficient programs that consider the effects of page replacement and frame allocation on the performance of a process and the system in which it executes

Non-Core:

6. Explain how hardware is utilized for efficient virtualization

OS-Devices: Device management

KA Core:

1. Buffering strategies (See also: AR-E: Interfacing and Communication)

2. Direct Memory Access and Polled I/O, Memory-mapped I/O Example concept: DMA communication protocols (ring buffers etc)(See also: AR-E: Interfacing and Communication)
3. Historical and contextual - Persistent storage device management (magnetic, SSD, etc)

Non-Core:

4. Device interface abstractions, HALs
5. Device driver purpose, abstraction, implementation and testing challenges
6. High-level fault tolerance in device communication

Illustrative Learning Outcomes

KA Core:

1. Understand architecture level device control implementation and link relevant operating system mechanisms and policy (e.g. Buffering strategies, Direct memory access, etc)
2. Understand OS device management layers and the architecture (device controller, device driver, device abstraction, etc)
3. Understand the relationship between the physical hardware and the virtual devices maintained by the operating system
4. Explain I/O data buffering and describe strategies for implementing it
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted

Non-Core:

6. Describe the complexity and best practices for the creation of device drivers

OS-Files: File Systems API and Implementation

KA Core:

1. Concept of a file including Data, Metadata, Operations and Access-mode
2. File system mounting
3. File access control
4. File sharing
5. Basic file allocation methods including linked, allocation table, etc
6. File system structures comprising file allocation including various directory structures and methods for uniquely identifying files (name, identified or metadata storage location)
7. Allocation/deallocation/storage techniques (algorithms and data structure) impact on performance and flexibility (i.e. Internal and external fragmentation and compaction)
8. Free space management such as using bit tables vs linking
9. Implementation of directories to segment and track file location

Illustrative Learning Outcomes

KA Core:

1. Understand the choices to be made in designing file systems

2. Evaluate different approaches to file organization, recognizing the strengths and weaknesses of each
3. Apply software constructs appropriately given knowledge of the file system implementation

OS-SEP: Society, Ethics and Professionalism

KA Core:

1. Open source in operating systems (See also: SEP-Intellectual Property)
   Example concepts
   a. Identification of vulnerabilities in open source kernels
   b. Open source guest operating systems
   c. Open source host operating systems
   d. Changes in monetization (paid vs free upgrades)
2. End-of-life issues with sunsetting operating systems (See also: SE-XXXXXXX)
   Example concepts: Privacy implications of using proprietary operating systems/operating environments, including telemetry, automated scanning of personal data, built-in advertising, and automatic cloud integration

Illustrative Learning Outcomes

KA Core:

1. Understand advantages and disadvantages of finding and addressing bugs in open source kernels
2. Contextualize history and positive and negative impact of Linux as an open source product
3. List complications with reliance on operating systems past end-of-life
4. Understand differences in finding and addressing bugs for various operating systems payment models

OS-AdvFiles: Advanced File systems

KA Core:

1. File systems: partitioning, mount/unmount, virtual file systems
2. In-depth implementation techniques
3. Memory-mapped files (See also AR-E: Interfacing and Communication)
4. Special-purpose file systems
5. Naming, searching, access, backups
6. Journaling and log-structured file systems (See also SF-F: System Reliability)

Non-Core: (including Emerging topics)

1. Distributed file systems (e.g NAS, OSS, SAN, Cloud, etc)
2. Encrypted file systems
3. Fault tolerance (e.g. fsync and other things databases need to work correctly).

Illustrative Learning Outcomes

KA Core:

1. Understand how hardware developments have led to changes in the priorities for the design and the management of file systems
2. Map file abstractions to a list of relevant devices and interfaces
3. Identify and categorize different mount types
4. Understand specific file systems requirements and the specialize file systems features that meet those requirements
5. Understand the use of journaling and how log-structured file systems enhance fault tolerance

Non-Core:

6. Understand purpose and complexity of distributed file systems
7. List examples of distributed file systems protocols
8. Understand mechanisms in file systems to improve fault tolerance

OS-Virtualization: Virtualization

KA Core:

1. Using virtualization and isolation to achieve protection and predictable performance (See also: SF-D-System Performance)
2. Advanced paging and virtual memory
3. Virtual file systems and virtual devices
4. Containers (See also: SF-D-System Performance)

Example concepts: Emphasizing that containers are NOT virtual machines, since they do not contain their own operating systems [where operating system is pedantically defined as the kernel]

5. Thrashing

a. Popek and Goldberg requirements for recursively virtualizable systems

Non-core:

6. Types of virtualization (including Hardware/Software, OS, Server, Service, Network) (See also: SF-D-System Performance)
7. Portable virtualization; emulation vs. isolation (See also: SF-D-System Performance)
8. Cost of virtualization (See also: SF-D-System Performance, SF-E: Performance Evaluation)
9. VM and container escapes, dangers from a security perspective (See also: SF-D-System Performance, SEC-Engineering)
10. Hypervisors- hardware virtual machine extensions

    Example concepts:

    a. Hypervisor monitor w/o a host operating system
    b. Host OS with kernel support for loading guests, e.g. QEMU KVM

Illustrative Learning Outcomes

KA Core:

1. Understand how hardware architecture provides support and efficiencies for virtualization

2. Understand difference between emulation and isolation
3. Evaluate virtualization trade-offs

Non-Core:

4. Understand hypervisors and the need for them in conjunction with different types of hypervisors
    a. Dynamic recompilation as an intermediary between full emulation and use of hardware hypervisor extensions on non-virtualizable ISAs whenever the guest and host system architectures match

OS-Real-time: Real-time/embedded

KA Core:

1. Process and task scheduling
2. Deadlines and real-time issues (See also: SPD-Embedded)
3. Low-latency/soft real-time" vs "hard real time" (See also: SPD-Embedded, FPL-S: Embedded Computing and Hardware Interface)

Non-Core:

4. Memory/disk management requirements in a real-time environment
5. Failures, risks, and recovery
6. Special concerns in real-time systems (safety)

Illustrative Learning Outcomes

KA Core:

1. Understand what makes a system a real-time system
2. Understand latency and its sources in software systems and its characteristics.
3. Understand special concerns that real-time systems present, including risk, and how these concerns are addressed

Non-Core:

4. Understand specific real time operating systems features and mechanisms

OS-Faults: Fault tolerance

KA Core:

1. Reliable and available systems (See also: SF-Reliability 1)
2. Software and hardware approaches to address tolerance (RAID) (See also: SF-Reliability)

Non-Core:

3. Spatial and temporal redundancy (See also: SF-Reliability 2)
4. Methods used to implement fault tolerance (See also: SF-Reliability 2,3)
5. Error identification and correction mechanisms (See also: AR-Memory)
    a. Checksumming of volatile memory in RAM
6. File system consistency check and recovery
7. Journaling and log-structured file systems (See also: SF-Reliability5)
8. Use-cases for fault-tolerance (databases, safety-critical) (See also: SF-Reliability1)

9. Examples of OS mechanisms for detection, recovery, restart to implement fault tolerance, use of these techniques for the OS's own services (See also: SF-Reliability)

Illustrative Learning Outcomes

KA Core:

3. Understand how operating system can facilitate fault tolerance, reliability, and availability
4. Understand the range of methods for implementing fault tolerance in an operating system
5. Understand how an operating system can continue functioning after a fault occurs
6. Understand the performance and flexibility trade offs that impact using fault tolerance

Non-Core:

7. Describe operating systems fault tolerance issues and mechanisms in detail

Professional Dispositions

- **Proactively considers** the implications for security and performance of decisions
- **Meticulously** considers implication of OS mechanisms on any project

Math Requirements

Required:

- Discrete math

Course Packaging Suggestions

Introductory Course to include the following:

- OS-Purpose: Role and Purpose of Operating Systems- 3 hours
- OS-Principles: Principles of Operating Systems- 3 hours
- OS-Concurrency: Concurrency- 7 hours
- OS-Scheduling: Scheduling- 3 hours
- OS-Process: Process Model- 3 hours
- OS-Memory: Memory Management- 4 hours
- OS-Protection: Protect and Safety- 4 hours
- OS-Devices: Device Management- 2 hours
- OS-Files: File Systems API and Implementation- 2 hours
- OS-Virtualization: Virtualization- 3 hours
- OS-AdvFiles: Advanced File Systems- 2 hours
- OS-Real-time: Real-time and Embedded Systems- 1 hours
- OS-Faults: Fault Tolerance- 1 hours
- OS-SEP: Social, Ethical and Professional topics- 4 hours

Pre-requisites:

- Assembly Level Machine Organization from Architecture

- Memory Management from Architecture
- Software Reliability from Architecture
- Interfacing and Communication from Architecture
- Functional Organization from Architecture

Skill statement: A student who completes this course should understand the impact and implications of operating system resource management in terms of performance and security. A student should understand and implement interprocess communication mechanisms safely. A student should differentiate between the use and evaluation of open source and/or proprietary operating systems. A student should understand virtualization as a feature of safe modern operating systems implementation.

Committee

Chair: Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA

Members:

- Renzo Davoli
- Avi Silberschatz
- Marcelo Pias, Federal University of Rio Grande (FURG), Brazil
- Mikey Goldweber, Xavier University, Cincinnati, USA
- Qiao Xiang, Xiamen University, China