Operating Systems (OS)

Preamble

The operating system is a 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 Input/Output (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 vs costs, scheduling approaches to processes, page replacement, etc.). The Operating Systems knowledge area contains the key underlying concepts for other knowledge areas — trust boundaries, concurrency, persistence, and safe extensibility.

Changes since CS2013

Changes from CS2013 include:

- Renamed File Systems knowledge unit to File Systems API and Implementation knowledge unit,
- Moved topics from the previous Performance and Evaluation knowledge unit to the Systems Fundamentals (SF) knowledge area,
- Moved some topics from File Systems API and Implementation and Device Management to the Advanced File Systems knowledge unit, and
- Added topics on systems programming and the creation of platform-specific executables to the Foundations of Programming Languages (FPL) knowledge area.

Core Hours

Knowledge Unit	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		2
Process Model		2
Memory Management		0.5 +1.5 (AR)
Device Management		0.5+0.5 (AR)

File Systems API and Implementation		2
Advanced File Systems		1
Virtualization		1
Real-time and Embedded Systems		1
Fault Tolerance		1
Society, Ethics, and the Profession	Included in SEP hours	
Total	8	13 (+ 2 counted in AR)

Knowledge Units

OS-Purpose: Role and Purpose of Operating Systems

CS Core:

1. Operating systems mediate between general purpose hardware and application-specific software.
2. Universal operating system functions (e.g., process, user and device interfaces, persistence of data)
3. Extended and/or specialized operating system functions (e.g., 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, tradeoffs between error checking and performance, flexibility and performance, and security and performance) (See also: SEC-Engineering)
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.
   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).
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 System

CS Core:

1. Operating system software design and approaches (e.g., monolithic, layered, modular, micro-kernel, unikernel)
2. Abstractions, processes, and resources
3. Concept of system calls and links to application program interfaces (e.g., Win32, Java, Posix). (See also: AR-Assembly)
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-Assembly)
   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-Assembly)
   Example concepts:
   a. Timer interrupts for implementing time slices
   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-Assembly)
8. Mechanism for invoking system calls, the corresponding mode and context switch and return from interrupt (See also: AR-Assembly)
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 (e.g., passing parameters, mode change).
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 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-Programs)
3. Deadlocks and starvation (See also: PDC-Coordination)
4. Multiprocessor issues (spin-locks, reentrancy).
5. Multiprocess concurrency vs multithreading

KA Core:

6. Thread creation, states, structures (See also: SF-Foundations)
7. Thread APIs
8. Deadlocks and starvation (necessary conditions/mitigations) (See also: PDC-Coordination)
9. Implementing thread safe code (semaphores, mutex locks, condition variables). (See also: AR-Performance-Energy, SF-Evaluation, PDC-Evaluation)
10. Race conditions in shared memory (See also: PDC-Coordination)

Non-Core:

11. Managing atomic access to OS objects (e.g., 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-Foundations)
2. Attacks and antagonism (scheduling, etc.) (See also: SEC-Foundations)
3. Review of major vulnerabilities in real operating systems (See also: SEC-Foundations)
4. Operating systems mitigation strategies such as backups (See also: SF-Reliability)

KA Core:

5. Policy/mechanism separation (See also: SEC-Governance)
6. Security methods and devices (See also: SEC-Foundations)
   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-Foundations, SEC-Crypto)

Illustrative Learning Outcomes:

CS Core:

1. Understand the requirement for protection and security mechanisms in 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 (e.g., first come, first serve, shortest job first, priority, round robin, multilevel) (See also: SF-Resource)
3. Concepts of Symmetric Multi-Processor (SMP) scheduling and cache coherence (See also: AR-Memory)
4. Timers (e.g., building many timers out of finite hardware timers) (See also: AR-Assembly)
5. Fairness and starvation

Non-Core:

6. Subtopics of operating systems such as energy-aware scheduling and real-time scheduling (See also: AR-Performance-Energy, SPD-Embedded, SPD-Mobile)
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. Explain the relationships between scheduling algorithms and application domains.
3. Explain the distinctions among types of processor scheduler 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 process vs threads.
6. List some contexts benefitting from preemption and deadline scheduling.

Non-Core:

7. Explain the ways that the logic embodied in scheduling algorithms are 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 (e.g., stack, heap, etc.) (See also: AR-Assembly)
3. Creating and loading executables, shared libraries, and dynamic linking (See also: FPL-Translation)
4. Dispatching and context switching (See also: AR-Assembly)
5. Interprocess communication (e.g., shared memory, message passing, signals, environment variables) (See also: PDC-Communication)

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-Memory, MSF-Discrete)
2. Impact of memory hierarchy including cache concept, cache lookup, and per-CPU caching on operating system mechanisms and policy (See also: AR-Memory, SF-Performance)

3. Logical and physical addressing, address space virtualization (See also: AR-Memory, MSF-Discrete)
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 concept: Arenas, slab allocators, free lists, size classes, heterogeneously sized pages (huge pages)
6. Memory caching and cache coherence and the effect of flushing the cache to avoid speculative execution vulnerabilities (See also: AR-Organization, AR-Memory, SF-Performance)
7. Security mechanisms and concepts in memory management including sandboxing, protection, isolation, and relevant vectors of attack (See also: SEC-Foundations)

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 tradeoffs.
2. Summarize the principles of virtual memory as applied to caching and paging.
3. Evaluate the tradeoffs 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 virtual machine 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-IO)
2. Direct Memory Access (DMA) and polled I/O, Memory-mapped I/O (See also: AR-IO)
   Example concept: DMA communication protocols (e.g., ring buffers etc.)
3. Historical and contextual - Persistent storage device management (e.g., magnetic, Solid State Device (SSD)) (See also: SEP-History)

Non-Core:

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

Illustrative Learning Outcomes:

KA Core:

1. Explain architecture level device control implementation and link relevant operating system mechanisms and policy (e.g., buffering strategies, direct memory access).
2. Explain OS device management layers and the architecture (e.g., device controller, device driver, device abstraction).
3. Explain 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
6. File system structures comprising file allocation including various directory structures and methods for uniquely identifying files (e.g., 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. Explain 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-Advanced-Files: 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-IO)
4. Special-purpose file systems
5. Naming, searching, access, backups
6. Journaling and log-structured file systems (See also: SF-Reliability)

Non-Core: (including emerging topics)
1. Distributed file systems
2. Encrypted file systems
3. Fault tolerance

Illustrative Learning Outcomes:
KA Core:
1. Explain 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. Explain specific file systems requirements and the specialized file systems features that meet those requirements.
5. Explain the use of journaling and how log-structured file systems enhance fault tolerance.

Non-Core:
6. Explain purpose and complexity of distributed file systems.
7. List examples of distributed file systems protocols.
8. Explain 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-Performance)
2. Advanced paging and virtual memory. (See also: SF-Performance)
3. Virtual file systems and virtual devices.
4. Containers and their comparison to virtual machines.
5. Thrashing (e.g., Popek and Goldberg requirements for recursively virtualizable systems).

Non-core:
6. Types of virtualizations (including hardware/software, OS, server, service, network). (See also: SF-Performance)
7. Portable virtualization; emulation vs isolation. (See also: SF-Performance)
8. Cost of virtualization. (See also: SF-Performance)
9. Virtual machines and container escapes, dangers from a security perspective. (See also: SEC-Engineering)
10. Hypervisors- hardware virtual machine extensions, hosts with kernel support, QEMU KVM

Illustrative Learning Outcomes:
KA Core:
1. Explain how hardware architecture provides support and efficiencies for virtualization.
2. Explain the difference between emulation and isolation.
3. Evaluate virtualization tradeoffs.

Non-Core:

4. Explain hypervisors and the need for them in conjunction with different types of hypervisors.

OS-Real-time: Real-time and Embedded Systems

KA Core:

1. Process and task scheduling.
2. Deadlines and real-time issues. (See also: SPD-Embedded)
3. Low-latency vs ”soft real-time" vs "hard real time." (See also: SPD-Embedded, FPL-Event-Driven)

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

Non-Core:

4. Explain specific real time operating systems features and mechanisms.

OS-Faults: Fault tolerance

KA Core:

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

Non-Core:

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

1. Explain how operating systems can facilitate fault tolerance, reliability, and availability.

2. Explain the range of methods for implementing fault tolerance in an operating system.
3. Explain how an operating system can continue functioning after a fault occurs.
4. Explain the performance and flexibility tradeoffs that impact using fault tolerance.

Non-Core:

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

OS-SEP: Society, Ethics, and the Profession

KA Core:

1. Open source in operating systems. (See also: SEP-IP)
   Example concepts:
   a. Identification of vulnerabilities in open-source kernels,
   b. Open-source guest operating systems,
   c. Open-source host operating systems, and
   d. Changes in monetization (paid vs free upgrades).
2. End-of-life issues with sunsetting operating systems.
   Example concept: 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. Explain 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.

Professional Dispositions

- **Proactive**: Students must anticipate the security and performance implications of how operating systems components are used.
- **Meticulous**: Students must carefully analyze the implications of operating system mechanisms on any project.

Mathematics Requirements

Required:

- MSF-Discrete

Course Packaging Suggestions

Introductory Course to include the following:

- OS-Purpose (3 hours)
- OS-Principles (3 hours)
- OS-Concurrency (7 hours)
- OS-Scheduling (3 hours)
- OS-Process (3 hours)
- OS-Memory (4 hours)
- OS-Protection (4 hours)
- OS-Devices (2 hours)
- OS-Files (2 hours)
- OS-Virtualization (3 hours)
- OS-Advanced-Files (2 hours)
- OS-Real-time (1 hour)
- OS-Faults (1 hour)
- OS-SEP (4 hours)

Prerequisites:

- AR-Assembly
- AR-Memory
- AR-Reliability
- AR-IO
- AR-Organization
- MSF-Discrete

Course objectives: Students should understand the impact and implications of operating system resource management in terms of performance and security. They should understand and implement inter-process communication mechanisms safely. They should be able to differentiate between the use and evaluation of open-source and/or proprietary operating systems. They should understand virtualization as a feature of safe modern operating system implementation.

Committee

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

- Qiao Xiang, Xiamen University, Xiamen, China
- Mikey Goldweber, Denison University, Granville, OH, USA
- Marcelo Pias, Federal University of Rio Grande (FURG), Rio Grande, RS, Brazil
- Avi Silberschatz, Yale University, New Haven, CT, USA
- Renzo Davoli, University of Bologna, Bologna, Italy