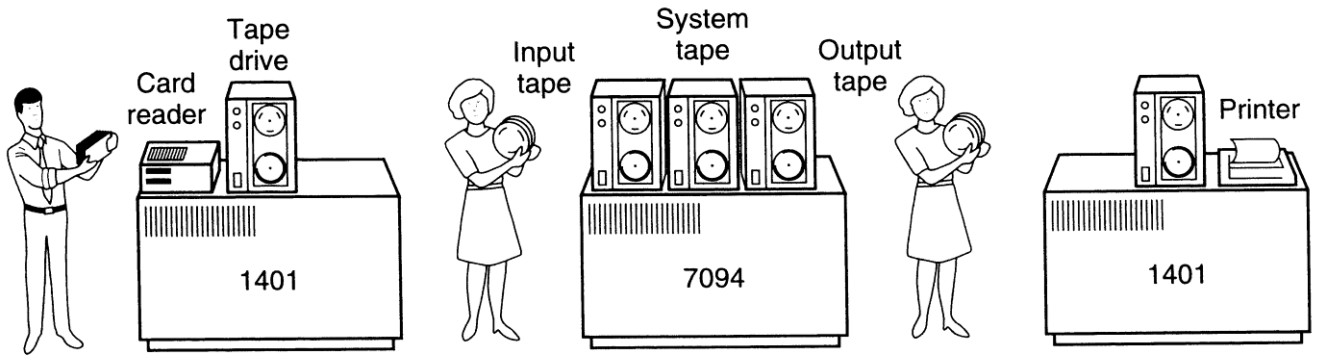
CS 202
Advanced Operating Systems

Winter 26

Lecture 2: Historical Perspective

Instructor: Chengyu Song

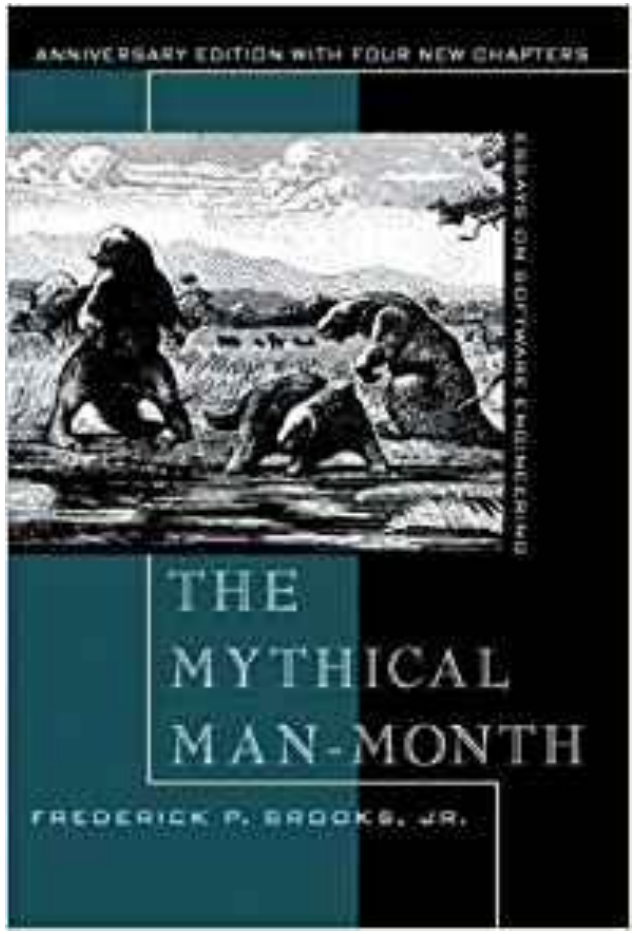
Phase 1: batch systems (1955-1970)



- Computers expensive; people cheap
  - Use computers efficiently – move people away from machine
- OS in this period became a program loader
  - Loads a job, runs it, outputs result, then moves on to next
  - More efficient use of hardware but increasingly difficult to debug
    - » Still less bugs ☺

Phase 1, problems

- Utilization is low (one job at a time)
- No protection between jobs
  - But one job at a time, so what can go wrong?
- Scheduling
- Coordinating concurrent activities
- People time is still being wasted
- Operating Systems didn't really work
  - The mythical man month
  - Birth of software engineering



Phase 2: 1970s

- Computers and people are expensive
  - Help people be more productive
- **Interactive time-sharing**: let many people use the same machine at the same time
- Emergence of minicomputers
  - Terminals are cheap
- Persistence: keep data online on fancy file systems

Modern architecture support for OS

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access
  - Memory isolation and translation
- Generating and handling “events”
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap
- Other stuff
  - Mechanisms to handle concurrency, isolation, virtualization …

Protected Instructions

- OS must have exclusive access to hardware and critical data structures
- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - » Security, fairness (why?)
  - Manipulate memory management state
    - » Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - » Kernel mode, interrupt level
  - Halt instruction (why?)

Privilege Mode

- Hardware restricts privileged instructions to OS
- Q: How does the HW know if the executed program is OS?
  - HW must support (at least) two execution modes: OS (kernel) mode and user mode
- Mode kept in a status bit in a protected control register
  - User programs execute in user mode
  - OS executes in kernel mode (OS == “kernel”)
  - CPU checks mode bit when protected instruction executes
  - Attempts to execute in user mode trap to OS

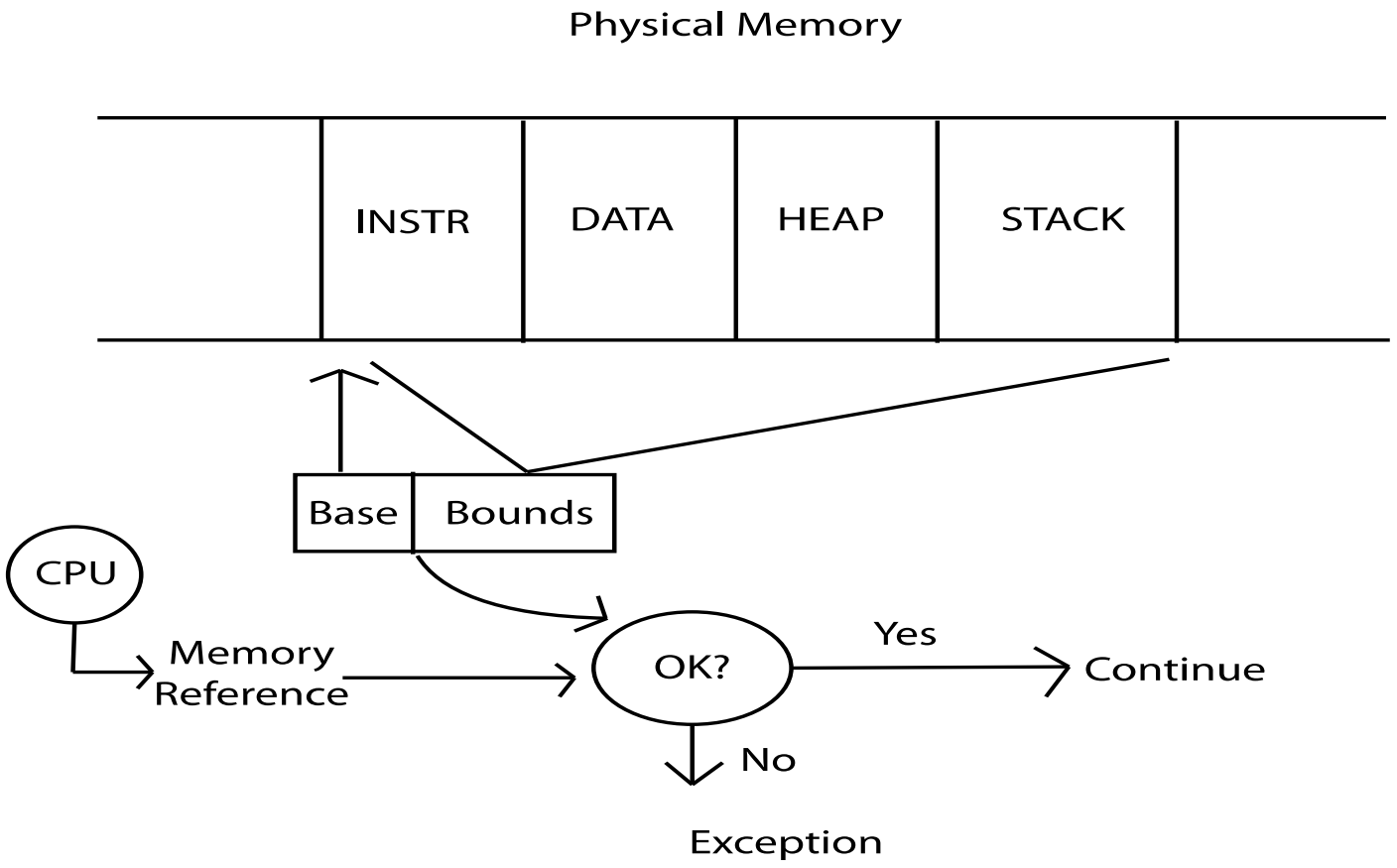
Switching back and forth

- Going from higher privilege to lower privilege
  - Easy: can directly modify the mode register to drop privilege
- But how do we escalate privilege?
  - Special instructions to change mode
    - » System calls (int 0x80, syscall, svc)
    - » Saves context and invokes designated handler
      - You jump to the privileged code; you cannot execute your own
    - » OS checks your syscall request and honors it only if safe
  - Or, some kind of event happens in the system

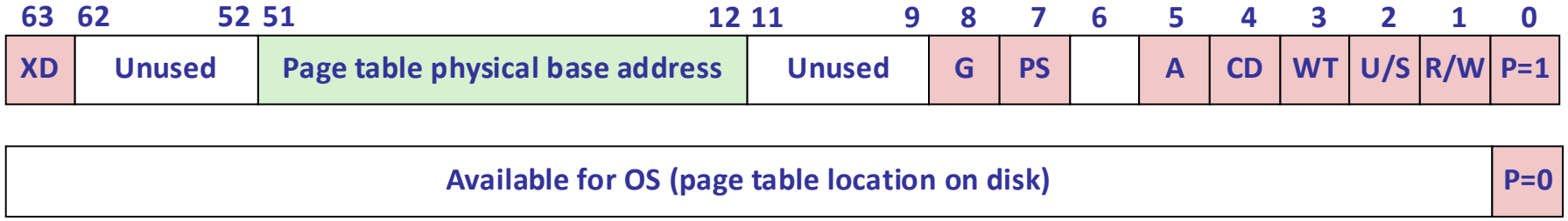
Memory Isolation

- OS must be able to protect programs from each other
- OS must protect itself from user programs
- OS may or may not protect user programs from itself
- Memory management unit (MMU)
  - Hardware unit provides memory protection mechanisms
  - Virtual memory
  - Segmentation
- Manipulating memory management hardware uses protected (privileged) operations

Example memory protection



Core i7 Level 1-3 Page Table Entries



P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 2 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

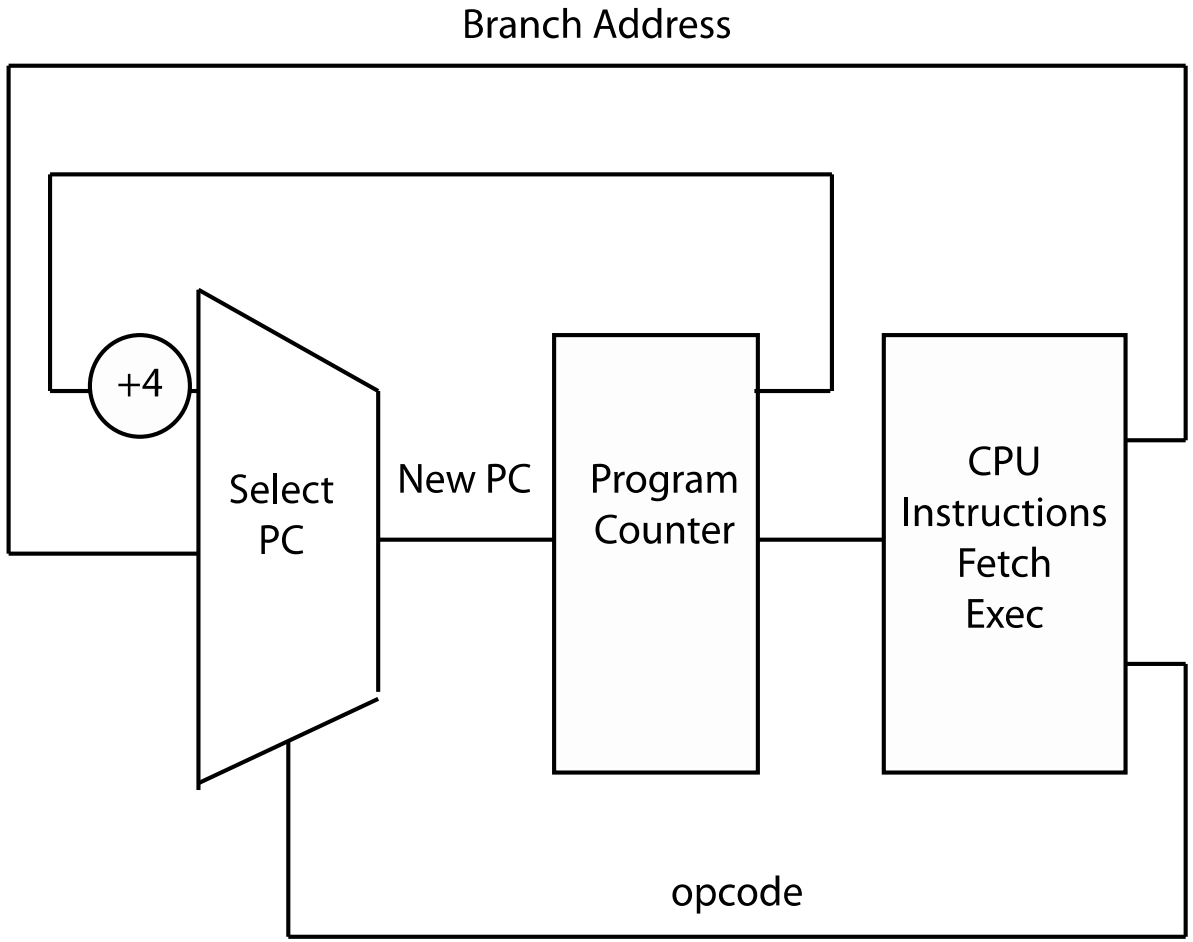
Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: Non-executable pages

Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access
  - Memory isolation and translation
- Generating and handling “events”
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

Review: Computer Organization



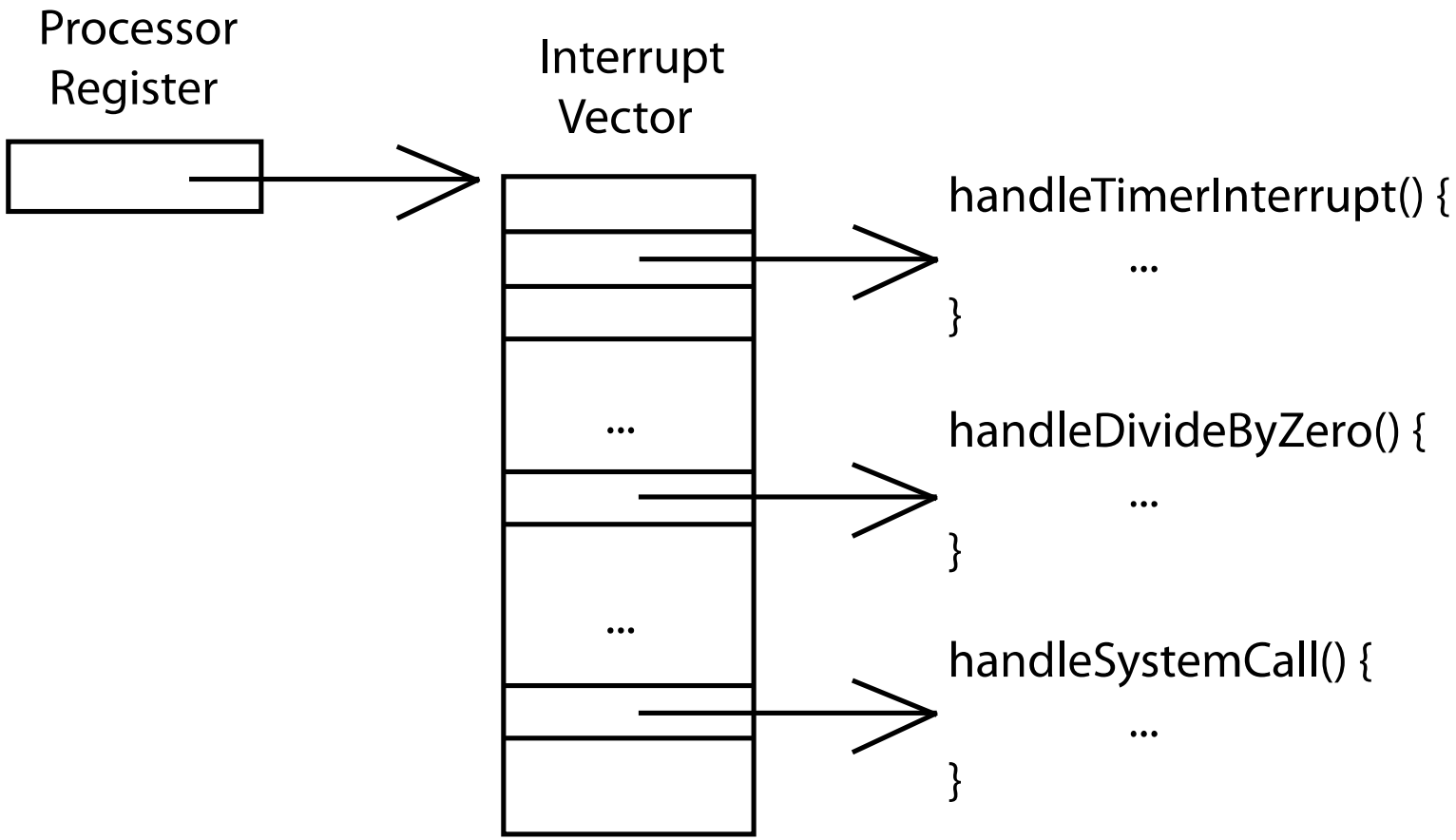
Events

- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both
- Most events are handled by the OS kernel, which defines a handler for each event type
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine
- Once the system is booted, OS is one big event handler
  - all entry to the kernel occurs as the result of an event

Context Switch

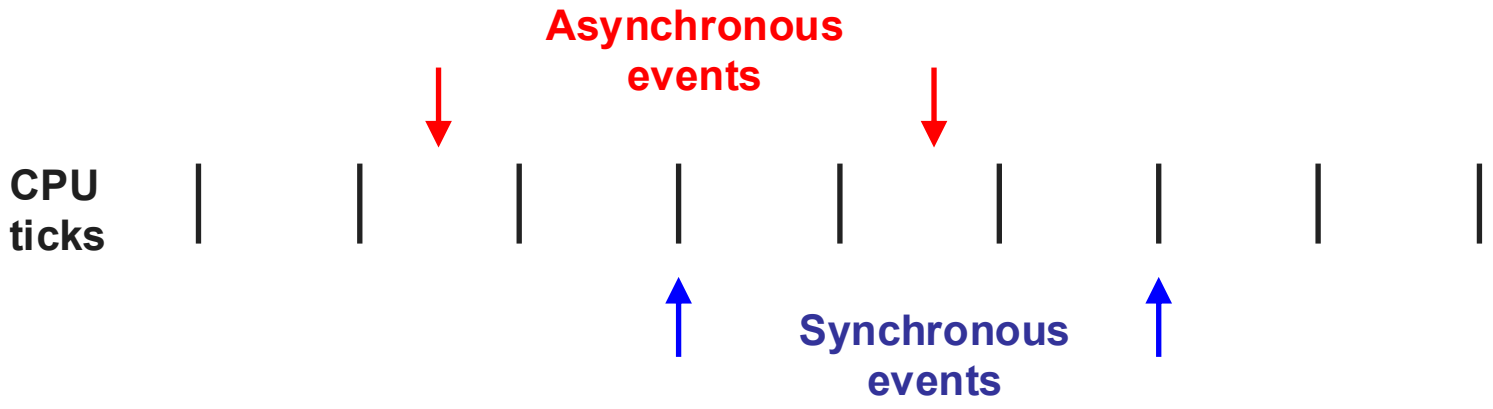
- An event is an “unnatural” change in execution, so there are a few important questions
  - How to make sure the event can be properly handled?
    - Who (what code) should handle the current event
    - What preparation is necessary to handle the current event
  - Once the event is done handling, how do we go back to previous execution?
  - Essentially, how to switch between different tasks?
- Context Switch is the key concept in understanding this, when event happens, the hardware will
  - Save the current execution context (CPU registers)
  - Setup the context for the handler (entry point of code, stack, privilege mode, etc.)
  - (after done) Restore the previous context

Handling events – Interrupt Vector Table



Categorizing Events

- Two *kinds* of events: synchronous and asynchronous
- Sync events are caused by executing instructions
  - Example?
- Async events are caused by an external event
  - Example?



Categorizing Events

- This gives us a convenient table:

	Unexpected	Scheduled
Synchronous	fault/exception	syscall
Asynchronous	interrupt	signal (and timer)

- Terms may be slightly different by OS and architecture
  - E.g., POSIX signals, async system traps, async or deferred procedure calls

Signals

- Software interrupts, from
  - Users (terminals, keyboard)
  - Other processes
  - OS (process' own execution)

Signal	ID	Description	Default Action
SIGINT	2	**Interrupt.** Sent when you press Ctrl+C. It politely tells the process to stop.	Terminate
SIGTERM	15	**Terminate.** The default signal sent by the kill command. It asks the process to clean up and exit.	Terminate
SIGKILL	9	**Kill.** The "nuclear option." It effectively pulls the power cord on the process. It cannot be ignored or handled.	Force Kill
SIGSEGV	11	**Segmentation Fault.** The process tried to read/write memory it doesn't own.	Dump Core
SIGHUP	1	**Hangup.** Sent when a terminal closes. Daemons often use this as a trigger to **reload configuration files** without stopping.	Terminate
SIGSTOP	19	**Stop.** Pauses the process (like pressing pause on a movie). Cannot be ignored.	Pause
SIGCONT	18	**Continue.** Resumes a process previously paused by SIGSTOP.	Resume

Other Staff

- Support for synchronization
  - How to build locks?
    - » Atomic operations
    - » Disabling interrupts
  - How to support memory consistency
    - » One CPU writes, visible to other CPUs
    - » How to keep caches consistent
  - How to communicate between CPU cores?

The TENEX System (Hardware/OS)

- Design Philosophy
  - The primary goal was to create a virtual machine that exceeded the physical limitations of the processor (PDP-10)
- The BBN Pager
  - Innovative hardware that provided individual mapping for 512-word pages → MMU
- Associative Registers
  - The pager used these to store mapping information, allowing for rapid address translation → TLB
- Large Address Space:
  - It provided a paged virtual address space equal to or greater than the processor's addressing capability with full protection and sharing

TENEX: Memory Management

- Copy-on-Write (COW)
  - TENEX introduced the "copy-on-write" facility, allowing processes to share large address spaces and only create private copies when a page is actually modified
- The Working Set Principle
  - Memory management was based on Peter Denning's "working set principle," aiming to identify the "balance set" of processes that could fit in core memory simultaneously to minimize thrashing
- Core Status Table
  - The hardware maintained a record of page activity (references and writes) in a core status table, which the software used to make informed page replacement decisions
- Demand Paging
  - No pages were preloaded; the system relied entirely on demand paging, making the paging traps invisible to the user process

TENEX: File System

- File-to-Memory Mapping
  - TENEX allowed a virtual memory slot to contain a pointer to a page from a file, effectively integrating the file system into the virtual address space → mmap
- Symbolic Naming and Versioning
  - It featured a multilevel symbolic directory structure and automatic version numbering for files. Every time a file was written, the system incremented the version number to prevent accidental data loss
- Uniform I/O
  - The system treated all external devices and data streams through a uniform interface, allowing I/O to be handled consistently regardless of the device → UNIX
- Thawed vs. Unthawed Access
  - The system managed file sharing through "thawed" access (allowing multiple writers/readers without consistency guarantees) or "unthawed" access (enforcing single-writer/multiple-reader consistency)

TENEX: User Interactions

- User-Centric Design
  - A major goal was "good human engineering," focusing on the TENEX Executive (EXEC) command interpreter → shell and coreutils
- Command Completion
  - The EXEC introduced the use of the ESC key for command and filename completion, as well as the "?" key for context-sensitive help
- Full-Duplex Interaction
  - The system was optimized for full-duplex terminals, allowing for "intimate interaction" where users could type input while the machine was still outputting data
- Pseudo-Interrupt System
  - This allowed processes to receive asynchronous signals from other processes or terminals, facilitating sophisticated terminal control → signals

Unix appears

- Ken Thompson, who worked on MULTICS, wanted to use an old PDP-7 laying around in Bell labs
- He and Dennis Richie built a system designed by programmers for programmers
- Originally in assembly. Rewritten in C
  - In their paper describing UNIX, they defend this decision!
  - However, this is a new and important advance: portable operating systems!
- Shared code with everyone (particularly universities)