Today's plan

- Minix:
  - reading and writing files
  - pipes
  - linking and unlinking
  - slow system calls
- distributed computing overview
- security overview
- multics security

Minix Reading and Writing

- reading and writing are the almost the same (`read_write`, p. 973):
  - check protections, sizes
  - find the correct block to read or write (perhaps allocating a new block)
  - copy the data to or from the block
- both reading and writing only access up to the next block boundary in a single iteration of the main loop
- `rw_chunk` (p. 977) does the actual (partial) block transfer, using `read_map` (p. 978) to obtain the address of an existing block
- `rw_chunk` may have to call `new_block` (p. 985) if writing to an as-yet-unwritten part of the file
- `new_block` uses `write_map` (p. 983) to do the hard work of allocating new zones in an inode, including the indirect and double-indirect blocks

Minix Pipes

- pipes are treated almost like files, except:
- the maximum size is limited to PIPE_SIZE (7 blocks, 28KB) -- all directly accessible from the inode
- when the file has been read completely, the write position is reset to the beginning (line 25202, p. 976)
- on a read, wake up any sleeping writers (`pipe_check`, p. 988)
- on a write, wake up any sleeping readers
- the "file" has an inode and a filp, but no directory entries (unless it is a "named pipe", which has a directory entry

Minix Linking and Unlinking

- linking adds a new link to an existing file
- unlinking removes an existing link to a file or directory
- `rmdir` is a slightly safer (more error checking) version of unlink
- when unliking a directory, must remove the "." and ".." entries and update the parent's link count
- renaming is almost like link followed by unlink, but slightly optimized to still work even when the disk is full
- renaming also allows renaming directories, whereas linking directories is only allowed to the superuser
- linking directories is dangerous because it introduces the risk of having loops in the file system hierarchy -- this could lead to infinite loops in the file system, which might be bad, e.g. line 27197 on p. 1007

Minix FS system call retry

- any process making a "slow" system call (read or write on a pipe, read on a terminal) may be suspended, in which case the system call must be retried later
- `suspend` (p. 989) copies the system call number and parameters to the process table, and avoids replying to the caller
- `release` (p. 990) calls `revive` for any process waiting on a given system call (e.g. writing) on a given pipe
- `revive` (p. 991) either sets a flag (line 26170), or directly replies to the suspended process (lines 26175, 26177, and 26181)
- the main loop of the file system process calls `get_work` (p. 959), which checks to see if any processes are being revived (line 24109), and only if none are being revived, then calls receive
- if processes are being revived, the system call number and arguments are taken from the process table rather than from the message received

Distributed Computing: Concepts

- Do we really care where our computer lives?
  - no -- as long as we can access all our files and run all our programs
  - yes -- for security
  - sometimes -- for performance
- benefits of distributed computing: the right number of computers at the right time
- challenges of distributed computing:
  - providing fast access to computing power
  - providing fast access to data (files)
  - providing access to I/O devices
  - providing secure access and computing
- the computing **environment** may be as important as the computing itself -- fortunately, Unix-like systems have supported distributed environments well, e.g. remote window clients via X windows (1980s), secure logins such as SSH (late 1980s)
- Windows environments have supported a different subset of distributed access, mostly access to data (e.g. SMB)

Cloud Computing

- commercial availability of distributed computing
- leverage:
  - different costs of electricity in different locations (arbitrage)
  - availability of "green" electricity in some locations
  - more efficient for intermittent use
  - more cost-efficient IT expertise: a few experts can manage many systems more efficiently/effectively than a single expert can manage a few systems
- useful for low security applications, e.g. most web applications
- or as long as the provider is trusted, the IT expertise can provide excellent security
  - e.g. financial applications

Distributed Computing: Goals and Strategies

- use a collection of computers as a single computer
- sometimes, designed to enhance reliability (if one computer is down, can still use other computers)
- process migration for faster performance or for access to specialized devices
  - even on a single computer, `fork` can make programs faster, e.g. in `make`
- remote procedure call to execute code that must be executed on a given machine
- paging across a network (to another system's memory or to a fast disk), may be faster than paging to a slow local disk
- most distributed computers support a distributed file system, such that all processors "see" the same file system

Architectures

- Commercial cloud provider:
  - data center with cheap electricity and/or cooling
  - many compute nodes connected by high-bandwidth, low-latency network
  - 24/7 technical support – nodes will fail, have a strategy to deal with that
- homebrew architectures:
  - closely coupled processors with a single shared memory (but separate caches) and devices shared among all CPUs: MP, multiprocessor, often now handled by traditional operating systems (Linux, Windows)
  - Network of Workstations (NoW), general-purpose workstations that agree to cooperate to get work done
  - Beowulf cluster, similar to an NoW but processors not intended for general-purpose use (maybe no graphics card, maybe no local disk), interconnected for high performance, and often in larger numbers than a typical NoW
  - grid computing: workstations, scattered across an internet, that agree to perform requested work while idle

Security Overview

- a *policy* determines who has access to what **data** and **resources**
- a *mechanism* allows the specification of a policy and implements the policy
  - e.g. file ownership and permissions are a mechanism, assignment of specific permissions (and/or ownership) to a file are policy
- an *audit trail* keeps track of who reads or modifies what data or code. It is best if audit trails are **write-once** or **append-only**
- users execute programs, but it is sometimes hard for a user to know what a program will do (even if the user has authored the program!), so programs should not trusted more than necessary

Multics Overview

- Multics was an early time-sharing system
  - and Multics inspired Unix
- designed for interactive use
- much of Multics was programmed in a high-level language (PL/1)
- potentially multiprocessor

Multics Security

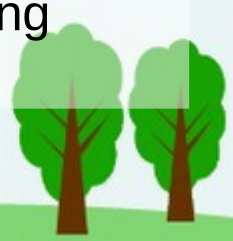
- Multics designers didn't want to repeat the insecurities of previous time-sharing systems
- hardware segmentation (with paging), with each segment having access protection bits
- segments are persistent and are stored in directories which have Access Control Lists (ACLs)
- hardware rings -- 8 concentric rings of protection, implemented as a separate descriptor table for each ring
- rings were designed to be implemented in hardware, but were actually implemented by having "master mode" be ring 0 and providing facilities to cross from one ring to another. When a program tries to add a segment to its space, ring 0 checks the ACL
  - similar to the supervisory bit in modern processors
- process table entries contain a user ID and are only accessible from ring 0
- passwords are stored encrypted, but the passwords file is also inaccessible to users
- users are told when they last tried to log in
- all system code is reloaded from tape when rebooting

Multics attack

- classic paper, e.g. at http://csrc.nist.gov/publications/history/karg74.pdf by Karger and Schell
- conclusion: not secure enough to have untrusted users share the computer with classified documents
- attacks designed to break hardware, software, and procedures, to the point of being able to access or modify "protected" data
- attackers had access to software and to hardware design (no "security by obscurity")
- attacks designed on a computer on an Air Force Base, then reproduced on a similar computer at MIT
- attacks should be undetectable, e.g. not cause anomalous crashes
- procedural vulnerabilities included physical access, the ability to use the software attacks to masquerade as a different user or change arbitrary memory or files, modifying the software at the development facility or enroute
- overall result: securing a system takes much more work than breaking in to a system!
- suggestion: a small (understandable) security kernel might do the trick

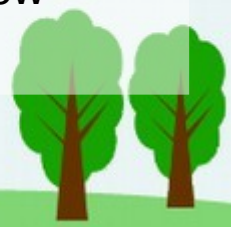
Multics Attack Specifics

- random tests of illegal operations or parameters to try and find unexpected behavior
- random tests uncovered a hardware bug that allowed some accesses to go unchecked (the bug had been introduced after the original design by an "upgrade")
- gate 0 code uses a routine to determine if argument lists are valid, but an autoincrement feature could lead the check to succeed but the actually accessed word to not be permitted
- a procedure (the "signaller", to deliver signals to user processes) was for efficiency placed in user space ("user ring") but ran in master mode, allowing a careful user to transfer to arbitrary code while still in master mode
- a change in usage of the stack segment register allowed user programs to store the registers in an arbitrary segment on a context switch, e.g. allowing the installation of a trap door into user code
- trap doors include modifying the TTY driver to recognize a special string, and modifying the compiler to recognize itself (and produce a trapdoor-generating compiler) and a given piece of code



Multics 30 years later

- new paper, e.g. at http://www.acsac.org/2002/papers/classic-multics.pdf, also by Karger and Schell
- Multics had security as its goal from the start, other systems don't (and therefore are weaker)
- modifying other systems to be more secure may interfere with "normal" usage, so the code base forks
- Multics did not suffer from buffer overflows, since PL/1 is safer than C and the segments prevented memory misuse
- Multics stacks grow upwards, so an overflow does not overwrite the return address
- Multics is simpler than most current systems, e.g. Multics ring 0 is much smaller than the SELinux security module
- intrusion detection often does not deal with the possibility of professional attackers who refine the attack on their system before attacking the target
- the GEMSOS OS certified to "Class A1" security standards, so the technology exists
- why aren't we doing better?
- chicken and egg problem: makers don't think customers want it, customers don't know makers can do it



Security in 2019

- Systems are routinely tested in real life
  - powerful actors have incentives to break systems
  - financial rewards for successful break-ins
    - both legitimate and not
- buffer overflow is still a problem (heartbleed)
- new side-channel attacks
  - e.g. meltdown and spectre (but no known exploits)
- users are often the weakest link (still!)
- zero-day attacks still appear