Operating Systems Functions

Steven Hand

8 lectures for CST Ib and Diploma

Lent Term 2000

Handout 4

Protection

Require protection against unauthorised:

- release of information
  - reading or leaking data
  - violating privacy legislation
  - using proprietary software
  - covert channels
- modification of information
  - changing access rights
  - can do sabotage without reading information
- denial of service
  - causing a crash
  - causing high load (e.g. processes or packets)
  - changing access rights

Also wish to protect against the effects of errors:

- isolate for debugging
- isolate for damage control

Protection mechanisms impose controls on access

by	SUBJECTS	to	OBJECTS
	active		passive
	users		memory
	processes		files
	programs		peripherals
			discs
			tapes
	a thread		a domain

Protection and sharing

Single user machine, no network connection, in a locked room - protection is easy.

But want to:

- share facilities - for economic reasons
- share and exchange data - applications requirement

Some mechanisms we have already come across:

- user and supervisor levels
  - usually one of each
  - could have several (e.g. MULTICS rings)
- memory management hardware
  - protection keys
  - relocation hardware
  - bounds checking
  - separate address spaces
- files
  - access control list
  - groups etc

Design of protection system

Some others:

- lock the computer room (stop tampering with hardware)
- restrict access to system software
- de-skill systems operating staff
- keep designers away from final system!
- legislate
- passwords (in general challenge / response)
- encryption (shared secret)

ref: Saltzer + Schroeder Proc. IEEE 634 pp1278-1308 Sept 75

- design should be public
- default should be no access
- check for current authority
- give each process minimum possible authority

- mechanisms should be simple, uniform and built in to lowest layers
- should be psychologically acceptable
- cost of circumvention should be high
- minimize shared access

Authentication of User to System

Passwords currently widely used:

- want a long sequence of random characters issued by system, but:
  - user would write it down
  - allow user selection - will use dictionary word, car registration, name of spouse
  - encourage use of an algorithm to remember password
- don't reflect on terminal, or overprint
- need to use encryption if line suspect
- security of password file?
  - only accessible to login program, e.g. CAP, TITAN
  - hold scrambled, e.g. UNIX
    * only need to write protect file
    * need irreversible function (without password)
    * maybe 'one-way' function
    * however, off line attack – use shadow passwords

Authentication of User to System

Passwords in UNIX:

- simple for user to remember

  `arachnid`

- sensible user applies an algorithm

  `!r!chn#d`

- use password for 'DES' like encryption of well known text

  `IML.DVMcz6Sh2`

Really require unforgeable evidence of identity that system can check:

- password
- id card inserted into slot
- fingerprint, voiceprint, face recognition
- smart cards

Authentication of System to User

User wants to avoid:

- talking to wrong computer
- right computer, but not the login program

Partial solution in olden days for directly connected terminals ...

- make login character same as terminal attention
- or, always do a terminal attention before trying login

But, micros used as terminals:

- local software may have been changed
- so carry your own copy of the terminal program
- but hardware / firmware in public machine may have been modified

Anyway, still have the problem of communication lines:

- simple wiretap

- workstation can see all packets on (certain types of) network

- must use encryption and trust encryption device - need smart cards

Mutual suspicion

- System of user
- Users of each other
- User of system

Called programs should be suspicious of caller (e.g. OS calls)

Caller should be suspicious of called program
(e.g. Trojan horse or Virus)

Trojan horse:

- 'useful' looking program - a game perhaps
- when called by user (in many systems) inherits all users privileges
- copy files
- modify files
- change password
- send mail

e.g. Multics editor Trojan horse, editor copied files as well as edited.

Virus:

- usually starts off as Trojan horse
- self-replicating

Access matrix

Matrix of subjects v. objects

Subject or principal:

- users e.g. by uid
- executing process in a protection domain (UNIX 2, MULTICS 8 rings)
- combinations

Objects:

- files
- devices
- domains / processes
- message ports

Matrix is large and sparse, two common representations:

- by object: store list of subjects and rights with each object
  access control list
- by subject: store list of objects and rights with each subject
  capabilities

Access Control Lists

Often used in storage systems:

- system naming scheme provides for ACL to be inserted in naming path, e.g. files
- if ACLs stored on disk, check is made in software $\Rightarrow$ must only use on low duty cycle
- for higher duty cycle must cache results of check
- e.g. Multics: open file = memory segment;
  On first reference to segment:
  - interrupt
  - check ACL
  - set up segment descriptor in segment table
  - place segment information in cache
- most systems check ACL
  - when file opened for read or write
  - when code file is to be executed
- access control by program, e.g. Unix
  - exam prog, RWX by examiner, X by student
  - data file, A by exam program, RW by examiner
- allows arbitrary policies ...

Capabilities

Capabilities associated with active subjects, so:

- store in address space of subject
- must make sure subject can't forge capabilities
- easily accessible to hardware
- can be used with high duty cycle
  e.g. as part of addressing hardware
  - Plessey PP250
  - CAP I, II, III
  - IBM system/38
  - Intel iAPX432
- have special machine instructions to modify (restrict) capabilities
- support passing of capabilities on procedure (program) call

Can also use *software* capabilities:

- checked by encryption
- nice for distributed systems

Implementations

Tagged Architectures (e.g. IBM system/38):

- all words in memory and the processor registers are tagged as containing either data or a capability
- tag stays with contents on all copy operations
- system checks ALU operations for validity

Capability segments (e.g. CAP):

- capabilities for code segment held in special capability segment
- only a restricted set of operations are allowed on capability segments
- provide a cache of entries in capability segments in special capability registers
- use associative store, per domain capability list, central capability list
- add *enter* capability

Software schemes (e.g. EROS)

- require capabilities for all system services
- fake out `enter` via IPC.

Password Capabilities

- Capabilities nice for distributed systems but:
  - messy for application, and
  - revocation is tricky.
- Could use timeouts (e.g. Amoeba).
- Alternatively: combine passwords and capabilities.
- Store ACL with object, but key it on capability (not implicit concept of "principal" from OS).
- Advantages: revocation possible, multiple "roles" available.
- Disadvantages: still messy (use 'implicit' cache?).

Covert channels

Information leakage by side-effects.
At the hardware level:

- wire tapping
- monitor signals in machine
- modification to hardware
- electromagnetic radiation of devices

By software:

- leak a bit stream as:

file exists	page fault	compute for a while	1
no file	no page fault	sleep for a while	0

- system may provide statistics
  e.g. TENEX password cracker using system provided count of page faults

In general, guarding against covert channels is prohibitively expensive.

Extensibility

What's it about?

- Fixing mistakes.
- Supporting new features (or hardware).
- Efficiency, e.g.
  - packet filters
  - run-time specialisation
- Individualism, e.g.
  - per-process thread scheduling algorithms.
  - customizing replacement schemes.
  - avoiding "shadow paging" (DBMS).

How can we do it?

- give everyone their own machine.
- allow people to modify the OS.
- allow some of the OS to run outside.
- reify separation between protection and abstraction.

Low-Level Techniques

Give everyone their own [virtual] machine:

- lowest level s/w does:
  - virtual h/w
  - (simple) secure multiplexing.

$\Rightarrow$ get N pieces of h/w from one.

Then simply run OS on each of these N:

✔ can pick and choose operating system.

✔ can even recompile and "reboot" OS without logging off.

✘ how big is a sensible value for N?

✘ layer violations...

- Examples: VM 370, VMWare, [SimOS?]
- Can also get N from M, e.g. Disco.

Kernel-Level Schemes (1)

Often don't require entirely new OS:

- Just want to replace/modify some small part.
- Allow portions of OS to be dynamically [un]loaded.
- e.g. linux kernel modules
  - requires dynamic relocation and linking.
  - once loaded must *register*.
  - support for [un]loading on demand.
- e.g. NT services and device drivers
  - well-defined entry / exit routines.
  - can control load time & behaviour.
- However there are some problems, e.g.
  - requires clean [stable?] interfaces
  - specificity: usually rather indiscriminate.
- ... and the big one: security.
  - who can you trust?
  - who do you rate?

Kernel-Level Schemes (2)

Various schemes exist to try to avoid security problems:

- Trusted compiler [or CA] + digital signature.
- Proof carrying code.
- Sandboxing:
  - limit [absolute] memory references to per-module [software] segments.
  - use *trampolines* for other memory references.
  - may also check for certain instructions.
- e.g. *SPIN* (U. Washington)
  - based around Modula-3 & trusted compiler
  - allows "handlers" for any event.
- Still problems with dynamic behaviour (consider handler `while(1);`) $\Rightarrow$ need more.
- e.g. Vino (Harvard)
  - uses "grafts" = sandboxed C/C++ code.
  - timeouts protect CPU hoarding.
  - in addition supports per-graft resource limits and transactional "undo" facility.
- Lots of work ...

User-Level Schemes

Can avoid complexity by putting extensions in user-space:

- e.g. μ-kernels + IDL (Mach, Spring)
- still need to handle timeouts / resource hoarding.

Alternatively reconsider split between *protection* and *abstraction* : only former need be trusted.

- e.g. Exokernel:
  - run most of OS in user-space library.
  - leverage DSL/packet filters for customization.
  - can get into a mess (e.g. UDFs).
- e.g. Nemesis:
  - guarantee each application share of *physical* resources in both space and time.
  - use IDL to allow user-space extensibility.
  - still requires careful design ...
- Is this the ultimate solution?

Summary & Outlook

- An operating system must:
  1. securely multiplex resources.
  2. provide / allow abstractions.
- Major aspect of OS design is choosing trade-offs.
  - e.g. protection vs. performance vs. portability
  - e.g. prettiness vs. power.
- Future systems bring new challenges:
  - scalability (multi-processing/computing)
  - reliability (computing infrastructure)
  - ubiquity (heterogeneity/security)
- Lots of work remains ...