HARDWARE AND OS SUPPORT FOR ACHIEVING ISOLATION/PROTECTION

1. Motivation

As discussed in class, real-world experience with computing systems running in batch mode in the 1950s and concurrent/multiprogramming systems in the late 60s (until today), reliable (or robust, fault-tolerant which for us mean the same) operation of computing systems became an important practical concern. At the heart of this concern is the protection of computing system resources from processes (i.e., apps) that may misbehave due to bugs or by intent (malware in today's jargon). Protecting hardware, OS, and other processes from misbehaving processes is achieved through isolating (or sandboxing, firewalling) a process so that the impact remains localized/contained (e.g., a misbehaving process is terminated). Providing isolation/protection is a key requirement of modern computing systems and their operating systems, and achieving isolation/protection efficiently (i.e., low overhead) requires specialized hardware and software (i.e., kernel) support.

2. Hardware support

We identified three hardware support features.

a) Privileged v. non-privileged instructions. Subdivide the instruction set of a CPU into privileged and non-privileged instructions. Privileged instructions are those that are needed to access shared resources (e.g., memory, I/O devices). For example, disabling all interrupts, read/write to an I/O device, halting a CPU are examples of operations that entail executing privileged instructions.

b) Kernel mode v. user mode. A process has a binary run-time state that we will call kernel mode and user mode. App processes (and most system processes) are started in user mode. Switching the state of a process to kernel mode requires executing a special hardware instruction, called a system call trap (e.g., sysenter in today's x86 CPUs). A process running in user mode that attempts to execute privileged instructions will trigger an interrupt by hardware whose default disposition is to terminate the process. Hence running in user mode low-level device driver code that contain privileged instructions to gain direct access to devices will fail.

System calls -- the API exported by an OS to apps to invoke pre-defined kernel services (such as read/write file I/O which entails device I/O) -- are "special" in that they result in a system call trap (hardware feature) that transforms the process that makes a system call from user mode to kernel mode (hardware feature). System calls, as we discussed, are function calls (i.e., branches) to kernel functions that were written by kernel programmers to carry out specific tasks (such as performing device I/O). Assuming these kernel functions do not contain bugs or hide malware, access to shared resources that require execution of privileged instructions can only be done through these pre-written kernel functions, hence protection of shared resources through isolation is affected.

Thus: An app that is started as a user mode process runs non-privileged instructions until it makes a system call to access shared resources. The system call toggles the process from user mode to kernel mode, runs kernel code containing privileged instructions that access shared resources on behalf of the requesting process, then returns to the process when the

requested task is completed (successfully or unsuccessfully) at which point the process is switched back to user mode from kernel mode.

c) As with system call trap (special hardware instruction) to switch from user mode to kernel mode, a special return instruction (return from system call) is provided to switch back from kernel mode to user mode after completion of a system call. Hence a typical time-evolution of a process entails toggling back and forth between user mode/kernel mode whenever system calls are invoked.

Keep in mind that kernel code that implement a system call are executed by the calling process, not by a separate special "kernel process." We say that kernel code is executed in the context of the calling process, albeit running in kernel mode.

Not all system calls need contain privileged instructions. For example, a system call such as getpid() in UNIX/Linux that requests the process ID of the calling process only requires look-up of the process table, a kernel data structure where this information is stored. Hence no privileged instructions come into play. However, a switch from user mode to kernel mode is still required since direct access of kernel data structures by apps running in user mode must be guarded.

d) Memory protection. Kernel memory (called kernel space) must be protected from app processes (memory belonging to apps is called user space). One app process's memory must be protected from another process unless explicit permission is granted. The sandboxing of memory (i.e., RAM) must be done in hardware since every memory reference (e.g., loc1 and loc2 in mov hardware instruction "mov loc1 loc2") of hardware instructions (which are operands of instruction opcodes) must be checked for validity. That is, the memory location to be accessed falls within a process's allocated memory area. For example, segmentation violation may be triggered by hardware (default disposition terminating the offending process) if a process references memory outside its allocated space.

In conjunction, hardware supported features a), b), c), and d) are critical for achieving isolation/protection in modern (and old) computing systems and their operating systems.

3. Software support

Software support in the sense of kernel support is needed, in conjunction with hardware features a), b), c), and d) to guarantee isolation/protection. The kernel feature is a per-process (i.e., by default) run-time stack that resides in kernel memory (kernel space). This is in addition to the per-process run-time stack of a process that resides in the process's user space memory. Since system calls invoke kernel functions (some invoking other kernel functions), a run-time stack is needed to perform efficient book-keeping of pushing/popping operations of stack frame chores. Since data utilized by kernel functions should not be accessible to app processes, we cannot use the per-process user-space run-time stack to do the function call book-keeping of kernel functions invoked by system calls. Thus when a system call is invoked by a process, its run-time stack switches from its user space run-time stack to a separate kernel space run-time stack.

These and other overhead render system calls more inefficient than regular function calls within a process that only use the process's user space run-time stack and do not trigger a switch from user mode to kernel mode. Thus efficient app code tends to be economic in its use of system calls to the extent possible.