10.2 Implementing Isolation for Processes

If we ignore the possibility of hardware attacks, then the environment for a program executing on a given computer comprises (i) a predefined instruction set for that computer and (ii) the state of that computer (which likely includes installed system software the program can invoke). Programs that execute together on a single computer can affect each other's environments whenever one of those programs updates state that can be read or executed by another. A *process* is an executing program whose state can be affected by other processes only in certain limited ways. So processes bring a form isolation. This isolation turns out to be crucial for implementing security in a computer that multiple programs are sharing—especially when the programs execute on behalf of different principals.

Two general approaches are typically combined in order to enforce the isolation required for processes. Main memory is protected by using *address translation*; the *processor state*, which comprises general-purpose registers as well as other processor registers (e.g., program counter), is protected by *time multiplexing*. The discussion that follows first explores each approach individually and then their combination.

10.2.1 Address Translation for Main Memory Isolation

Process isolation for main memory can be achieved by ensuring that each process uses a disjoint sets of names for its main memory locations. The usual approach is to employ address translation mappings from a single set N of memory location names all processes use. A distinct address translation mapping $Mmap_P$ is used by each executing process P. $Mmap_P$ maps elements of its domain N to elements of its range M, where each $m \in M$ is the address of a word in memory. The value that process P accesses using a name $n \in N$ is the value stored by memory word $Mmap_P(n)$. For example, when P executes `x := x + y`, the computer would store in memory word $Mmap_P(\texttt{x})$ the sum of the values stored by memory words $Mmap_P(\texttt{x})$ and $Mmap_P(\texttt{y})$.

Process isolation is achieved if the address translation mappings used by different processes have disjoint ranges—that is, location $Mmap_P(n)$ that a process P associates with a name n is different from the location that any other process P' can associate with any name. For any names n and n' in N:

$$(P \neq P') \Rightarrow (Mmap_P(n) \neq Mmap_{P'}(n'))$$

We would expect also that, for each process P, $Mmap_P$ designates distinct memory words for different names n and n':

$$(n \neq n') \Rightarrow (Mmap_P(n) \neq Mmap_P(n'))$$

In typical implementations of address translation, a register `MmapReg`, which is part of the processor state, designates the mapping in use. The value in `MmapReg` is changed whenever the computer switches from executing instructions from one process to executing instructions from another.

Address-Translation Process-Switch Protocol. Between executing an instruction from a process P and an instruction from a different process P', register `MmapReg` is updated to designate $Mmap_{P'}$. □

The cost of switching from one process to another is thus the cost of updating `MmapReg`. That update might also require flushing pipelines and purging caches and, thus, could have a performance penalty.

CPU Support for Address Translation

Any mapping μ from (finite) sets N to M can be represented using a (finite) table. In that table, the row for $n \in N$ would contain $\mu(n)$. If N is the set of all addresses for a computer's main memory, then the naive approach to storing this table would consume all of that main memory, leaving no room for programs or data. Smaller tables can be used to describe mappings, however, if each row maps many elements from N rather than mapping only a single element.

Such a scheme is employed by virtual memory architectures that interpret a virtual address as comprising two parts: a segment name and an offset into that segment.[6] The elements of N and virtual addresses; a segment is stored in main memory by using consecutive elements from M.[7] A *segment descriptor* is a data structure that specifies (among other things) a triple $\langle n_i, len_i, m_i \rangle$, with $0 \le len_i$. It relates subsets

$$\{n_i, n_i + 1, \ldots, n_i + len_i - 1\} \subseteq N$$
$$\{m_i, m_i + 1, \ldots, m_i + len_i - 1\} \subseteq M$$

by mapping every virtual memory address n satisfying $n_i \le n < n_i + len_i$ to address $m_i + (n - n_i)$ in main memory. A set *Segs* of segment descriptors for disjoint blocks of N thus specifies a mapping *Mmap* from names n in N to addresses in main memory, as follows.

$$Mmap(n): \;\; m_i + (n - n_i) \quad \text{if } \langle n_i, len_i, m_i \rangle \in Segs \text{ and } n_i \le n < n_i + len_i$$

The representation of set *Segs* is determined by a computer's instruction-set architecture, as are the details of how `MmapReg` designates that mapping. On some computers, `MmapReg` is called the *segment table register*; it contains the memory address for a table of segment descriptors (with that table itself a segment). Other computers eliminate a level of indirection by providing a small set (typically 2 or 4) of registers `MmapReg`$[i]$, each storing a segment descriptor. Here, *Segs* is the (small) set of segment descriptors contained in those registers.

[6] Figure 7.8 (page 119) gives details for a typical segmented virtual memory.

[7] By further dividing each segment into fixed size pages, a single block of contiguous main memory is no longer needed to hold the entire segment. A virtual address now comprises three parts: a segment name, a page name, and an offset into the page. And an additional mapping, supported by hardware, translates each page name to the main memory address of the page frame that holds this page. The existence of paging should be transparent, and thus it is ignored in this chapter.

Early computers supported name mapping where *Segs* is defined using two registers: a *base register* `Base` and a *limit register* `Lim`. These registers defined a set *Segs* that contained the single segment descriptor $\langle 0, L, B \rangle$, where L is the value contained in `Lim` and B is the value contained in `Base`. Any address between 0 and the value in `Lim` was relocated by adding the value in `Base`.

10.2.2 Time multiplexing

A resource that is time multiplexed is, by definition, assigned exclusively to one task at a time, with that assignment switching between tasks periodically. We can use time multiplexing to isolate processes that are sharing a processor. The resource being time multiplexed is the processor, the tasks are processes, and whenever the processor is about to switch from executing a process P to executing another process, the current processor state is saved for use when execution of P is later resumed.

To implement time multiplexing for a processor, a separate, isolated memory region $Ptbl[P]$ is set aside to hold the processor state associated with each process P. The following protocol then ensures that the appropriate processor state has been loaded whenever a process resumes execution.

> **Time-Multiplexing Context-Switch Protocol.** Between executing an instruction from a process P and an instruction from a different process P', the processor performs a *context switch* by
>
> (i) storing the current processor state in $Ptbl[P]$
>
> (ii) loading a new processor state from $Ptbl[P']$. □

Notice that Address-Translation Process-Switch Protocol (page 270) is implemented by the above protocol, because `MmapReg` (or whatever register(s) implement its functionality) is part of the processor state.

CPU Support for Time Multiplexing

Implementations of Time-Multiplexing Context-Switch Protocol generally combine hardware support with additional *system software*.[8] We sketch a design below. In this design, memory that stores code and state associated with each process is isolated by using the address translation mappings described above. And memory regions that store code and state associated with system software is isolated by employing address translation mappings that allow no process P to reference those regions.

Processor Modes. Hardware support for time multiplexing and address translation is typically controlled by using special instructions. The result is two instruction sets: $Inst_U$ and $Inst_S$.

[8] Software is involved for good reason. It can be changed long after the hardware is produced, it can differ across system instances without increasing hardware-design costs (which tend to be formidable), and it is the better medium for implementing complicated policies.

- $Inst_U$ is the set of *user-mode instructions*. These instructions can access only a subset of the processor state and can access only those regions of main memory that address translation mappings make available.

- $Inst_S$ adds *system-mode instructions* to $Inst_U$. System-mode instructions can read and write all of the processor state and main memory; they therefore can control hardware support for time multiplexing, address translation, and input/output.

Notice that $Inst_U \subset Inst_S$ holds.

System-mode instructions, because they are able to change all processor state and main memory, can breach process isolation and also can facilitate breaches by subsequent execution of user-mode instructions (e.g., by changing an address translation mapping so that a `store` instruction executed subsequently by one process updates memory used by another process). Therefore, a processor will restrict which programs are able to execute system-mode instructions. System software must be trusted for a variety of reasons, so it typically would be able to execute system-mode instructions; other software is not trusted and, therefore, it typically is not able to execute system-mode instructions.

A common means for specifying to a processor whether system-mode instructions are available to the currently executing program is to employ a register `mode` (say), where either `mode` = U or `mode` = S holds. The processor allows execution of an instruction ι to proceed only if `mode` = m and $\iota \in Inst_m$ hold. A system-mode instruction `chMode` (say) is provided for updating `mode`. Therefore, a process executing with `mode` = U is unable to execute `chMode` and alter `mode`. When a process is not executing then the value of `mode` being stored for it cannot be altered by a process executing with `mode` = U, because of the isolation enforced by time multiplexing (for processor state) and by address translation (for memory).

Interrupts. Hardware support to instigate and handle *interrupts* allows efficient implementation of periodic context switches that time multiplexing a CPU requires. We arrange for interrupts to occur regularly, and we configure an *interrupt handler* to invoke system software that implements Time-Multiplexing Context-Switch Protocol. Details are described below for a hypothetical CPU, which includes key features of real hardware.

Interrupts on our hypothetical CPU are grouped into a fixed number of classes (as is typical) and are used for time multiplexing as well as for supporting other functionality. Processor state is associated with each class Int_i of interrupts. See Figure 10.1. Interrupt processing proceeds as follows.

> **Hardware Interrupt Processing.** If $enabled_i$ is *false* and an interrupt of class Int_i is signalled then that interrupt remains *pending* until $enabled_i$ becomes *true*. If $enabled_i$ is *true* and there are pending interrupts of class Int_i or a new interrupt of class Int_i is signalled then the hardware selects one of those interrupts and *delivers* that interrupt:

Name	Description
$enabled_i$	Boolean that defines whether a new interrupt of class Int_i will be delivered immediately or Int_i interrupts are *disabled* and will remain pending.
$Intrpt[i].old$	Region of memory where the processor state is stored when an interrupt of class Int_i is delivered.
$Intrpt[i].new$	Region of memory from which a new processor state is loaded when an interrupt of class Int_i is delivered.

Figure 10.1: State for Interrupt Class Int_i

(i) the current processor state is copied into $Intrpt[i].old$

(ii) a new processor state is loaded from $Intrpt[i].new$. □

For our hypothetical CPU, $enabled_i$ is bit i of a register `Enabled`, and there is an *interrupt vector* register `IntVector`$[i]$ containing the real memory address of $Intrpt[i]$ for each interrupt class Int_i. These registers along with associated $Intrpt[i].old$ and $Intrpt[i].new$ memory regions are part of the processor state, and they may be updated only by executing system-mode instructions.

Some interrupts are signalled in response to asynchronous events (e.g., completing an input/output operation). Other interrupts, called *traps*, are associated with execution of the current instruction (e.g., attempting to execute a system-mode instruction while `mode` = U, referencing inaccessible memory, or causing arithmetic overflow, underflow, or division by zero). The traps are grouped, forming one or more of the interrupt classes. Because it makes no sense to disable a trap, the value of `Enabled`$[i]$ is considered equal to *true* for any class Int_i of traps—no matter what value `Enabled`$[i]$ actually stores.

We arrange for an interrupt handler $Hndlr_i$ to be executed in response to an interrupt of class Int_i by initializing $Intrpt[i].new$ to a processor state in which the program counter (register `pc`) denotes an entry point for $Hndlr_i$. To ensure that each invocation of $Hndlr_i$ is guaranteed to complete before another interrupt of class Int_i causes $Hndlr_i$ to be invoked again,

(i) $Intrpt[i].new$.`Enabled`$[i]$ (i.e., $enabled_i$) is initialized to *false*,

(ii) execution of $Hndlr_i$ does not change `Enabled`, and

(iii) $Hndlr_i$ is written in a way that no instruction can cause a trap.

By initializing $Intrpt[i].new$.`mode` = S, we ensure that $Hndlr_i$ executes with `mode` = S and can execute system-mode instructions to change the processor state. And we initialize $Intrpt[i].new$.`MmapReg` with a mapping that provides the interrupt handler with access to any state it needs.

Modern CPUs typically offer a class of interrupts designed specifically to support processor time-multiplexing. A *timer interrupt* is generated whenever the *interval timer*—a register that is automatically decremented by the hardware as time advances—reaches 0. System-mode instruction `loadtmr` loads the


```
var ProcessTable[ 1 .. NumProcs ]: record
                                     ps: processor state
                                     elgbl: boolean
                                   end record
    LastRun: 1 .. NumProcs

                    ⋮

Hndlr_IT: procedure
            ProcessTable[LastRun].ps := Intrpt[IT].old
            call Dispatcher
          end Hndlr_IT

Dispatcher: procedure
              p := Scheduler()  {ProcessTable[p].elgbl = true}
              LastRun := p
              loadtmr τ {loads interval timer with τ}
              Load processor state from ProcessTable[p].ps
            end Dispatcher
```

Figure 10.2: Code for Processor Multiplexing

interval timer, thereby scheduling a timer interrupt for the future. Timer interrupts thus can be scheduled to instigate regular context switches.

Figure 10.2 contains the sketch of an interrupt handler $Hndlr_{IT}$ for timer interrupts. This code shares *ProcessTable* and *LastRun* with all other interrupt handlers, where *ProcessTable* and *LastRun* are not directly accessible to any process. *ProcessTable*$[p]$.*ps* stores the processor state when process p was last interrupted, and *ProcessTable*$[p]$.*elgbl* indicates whether process p is eligible to run (versus being blocked because, say, the process is waiting for an I/O operation to complete). *LastRun* stores the index for the *ProcessTable* entry that corresponds to the process most recently executed; this information is needed by an Int_i handler for moving the processor state from *Intrpt*$[i]$.*old* into the correct entry of *ProcessTable*.

Dispatcher, which is invoked at the end of every interrupt hander, resumes some previously executing process. Its invocation of *Scheduler* is assumed to return a value p that satisfies *ProcessTable*$[p]$.*elgbl* = *true* and, thus, identifies some process that can execute. The last step ("Load processor state ...") resumes that selected process at the last interruption of it's execution, because the saved processor state included the program counter value at the time that interruption occurred. By loading τ into the interval timer just before the process is resumed, *Dispatcher* schedules a timer interrupt for τ seconds in the future.

Extending the Instruction Set. System-mode instructions provide functionality that could be needed by user-mode processes. Yet these instructions are not available to user-mode processes—with good reason. For example, input is typically obtained by executing a system-mode instruction that transfers data from some device to main memory; operands specify the device and specify a real (not mapped) address in main memory where the data will be copied. If that instruction were user-mode then user-mode processes could initiate data transfers that breach process isolation by changing any location in memory.

System software, however, can be programmed to invoke system-mode instructions on behalf of user-mode software and in ways that cannot compromise isolation. So it can provide a safe way for user-mode processes to access to system-mode functionality. A mechanism for user-mode processes to invoke system software completes the picture. Such a mechanism would

- transfer control from a user-mode process to the entry point of some specified system software, which would execute with `mode` = S, and later
- transfer control from that system software back to the invoker, which would execute again as a user-mode process.

The control transfer we seek—to change not only the program counter but also the processor mode—is realized by a mechanism that loads the processor state. Provided that user-mode processes are unable to change the memory from which the new processor state is loaded, we can have assurance that predetermined *gates* are the sole entry points to which control can be transferred.[9]

Interrupts cause a control transfer and, by design, $Intrpt[i].new$ regions of memory cannot be changed by a user-mode process. So traps could be used to effect the control transfers that we seek. CPUs thus typically provide a user-mode *supervisor call* instruction `svc` that causes a trap. Our hypothetical CPU follows the conventions that (i) prior to executing an `svc`, general-purpose register `r` will contain an integer to indicate what function is being requested of system software, and (ii) operands are placed in other registers. Because the contents of all registers are automatically stored in $Intrpt[SVC].old$ by CPU hardware whenever an `svc` is executed, handler $Hndlr_{SVC}$ can take different actions by executing different code according to the value found in $Intrpt[SVC].old.\texttt{r}$.

Figure 10.3 illustrates the overall structure of a supervisor call trap handler. It also sketches code for handling a hypothetical supervisor call "`svc` 23" (an `svc` invoked when"23" is loaded in general-purpose register `r`) that initiates input/output.[10] Notice, any process that executes "`svc` 23" gets marked by $Hndlr_{SVC}$ as being ineligible to be run. That process presumably would be

[9] Contrast this control transfer with a procedure call, which is usually implemented by a user-mode code fragment that ultimately loads a new value into the program counter (thereby effecting a branch to a procedure's entry point). With a procedure call, nothing prevents an attacker from modifying that calling code so that it loads a different address and, thus, transfers control elsewhere. In addition, the code fragment for implementing a procedure call does not change the processor mode from U to S, nor could this mode change be accomplished by instructions that a user-mode process could have executed.

[10] *Dispatcher*, called at the end, was given in Figure 10.2.


```
Hndlr_SVC: procedure
             ProcessTable[LastRun].ps := Intrpt[SVC].old
             case Intrpt[SVC].old.r
                 ⋮
               when 23: {svc 23 to instigate input/output}
                        Start input/output operation
                        ProcessTable[LastRun].elgbl := false
                        call Dispatcher
                        end when 23
                 ⋮
             end case
           end Hndlr_svc
```

Figure 10.3: Code for Supervisor Call Interrupt Handler

later marked eligible to be run by the handler for input/output interrupts—specifically, when that handler gets the interrupt to signal completion of the requested input/output.

An `svc` not only provides a way for system-mode instructions to be executed on behalf of user-mode processes, but an `svc` also can be used to invoke other functionality we implement as system software. So `svc`'s extend the functionality available in $Inst_U$. Such extensions, however, are a mixed blessing for process isolation. We illustrate by giving examples of functionality that operating systems typically make available though `svc`'s.

- *Interprocess communication and synchronization primitives.* These primitives allow an application to be structured as a set of cooperating processes rather than as a monolithic program executing in a single process. The added structure facilitates human understanding, and process isolation makes it easier to localize problems within the application's code. But an `svc` that allows processes to communicate and/or synchronize necessarily violates process isolation, since actions by one process now influence the execution of another process.

- *Dynamic allocation primitives.* Primitives for requesting a resource from a shared pool and for later releasing it enable higher utilization than could be achieved through a fixed allocation to each process. Yet delays may now arise from contention. And actions one process takes become visible to other processes through those delays. Isolation is thus compromised (albeit, only slightly) when processes share resources and operating system primitives provide support for dynamic allocation.

How much does any given `svc` compromise process isolation? The answer requires analyzing the code for the `svc` as well as the code for other system

software that reads state the `svc` updates. That body of code would include all of the interrupt handlers since, for example, there exists state (e.g., *ProcessTable*) that is read and updated by all. A formal analysis of such a large body of code is unlikely to be feasible; an informal analysis might overlook things. So determining what isolation potentially is eroded by a given `svc` can be quite difficult.

Also, system code we add to extend the functionality of $Inst_U$ increases the risk of bugs. Of concern would be bugs that allow an `svc` invocation by one process to contaminate *ProcessTable* or some other interrupt handler state, ultimately corrupting the states of other processes. The risk of compromised process isolation is offset, though, if an `svc` provides functionality that simplifies, hence increase our basis for trusting, user-mode processes. Moreover, developers of system programs serve large user communities, which creates incentives for investments in assurance for system programs.

10.3 Hardware Rings of Protection

Systems are often structured hierarchically. In such an architecture, each *layer* maintains state and exports operations for use by higher layers but not by lower layers. The lowest layer might be implemented by hardware; it exports machine language instructions. Other layers are implemented in software. They hide, redefine, and/or augment operations exported by lower layers.

A defining characteristic of a hierarchical system is that higher layers trust lower layers: if a layer L invokes an operation exported by some lower layer L' then correct operation of L depends on correct operation of L'. This trust does not rule out higher layers from attempting to subvert lower layers—an isolation mechanism must be provided to defend against that.

The implementation of processes sketched in §10.2.2 is structured hierarchically. The lowest layer is hardware. A next layer comprises interrupt handlers and other systems programs. User-mode processes are layered above that. User-mode processes must trust the interrupt handlers, but the interrupt handlers need not trust user-mode processes. Because user-mode processes trust interrupt handlers, the interrupt handlers can have direct read/write access to the memory for all processes. Such direct access, for example, allows an `svc` to copy directly to/from user-mode process's memory, which enables input/output to be performed efficiently. But layering user-mode processes above interrupt handlers does mean that state used by interrupt handlers should be protected from accesses by user-mode processes. The defense here is to employ name mappings (described next) that user-mode processes cannot control.

Enforcing Isolation. With lower layers in a hierarchically structured system trusted by higher layers, execution in a given layer is (i) authorized to read and write state being maintained by higher layers but (ii) not authorized to read or

write state maintained by lower layers. That is, for any layers L and L',

$$L' \prec L \;\Rightarrow\; (\mathit{read}(L') \supseteq \mathit{read}(L) \;\wedge\; \mathit{write}(L') \supseteq \mathit{write}(L)) \tag{10.1}$$

where relation, $L' \prec L$ on layers holds[11] when layer L' sits below (hence, is trusted by) layer L, set $\mathit{read}(L)$ enumerates state components that code in layer L is authorized to read, and set $\mathit{write}(L)$ enumerates state components that code in layer L is authorized to write.

CPU hardware for address translation often will directly enforce access restrictions (10.1) implied by layering. A metaphor of nested or concentric *rings* is adopted, with hardware providing

- a register `curRing` to associate a ring with current execution, and
- a means to assign access restrictions for execution in each given ring.

The ordering of the concentric rings corresponds to the ordering of layers. Specifically, each layer L is associated with a non-negative integer $\mathit{ring}(L)$ that satisfies

$$L' \prec L \;\Rightarrow\; 0 \le \mathit{ring}(L') < \mathit{ring}(L). \tag{10.2}$$

Access restrictions on the code being executed are imposed according to the value of `curRing`. The access restrictions are specified in segment descriptors. A segment descriptor $\langle n_i, \mathit{len}_i, m_i \rangle$ for mapping names n satisfying $n_i \le n < n_i + \mathit{len}_i$ (as discussed on page 270) is extended with fields

- rb_i an integer specifying a *read bracket* comprising set of layers L satisfying $0 \le \mathit{ring}(L) \le rb_i$, and
- wb_i an integer specifying a *write bracket* comprising set of layers L satisfying $0 \le \mathit{ring}(L) \le wb_i$.

A read access to a name in segment $\langle n_i, \mathit{len}_i, m_i, rb_i, wb_i \rangle$ is authorized only if $0 \le$ `curRing` $\le rb_i$ holds and, therefore, the currently executing layer is in the read bracket for the segment being accessed; an access violation occurs otherwise. If rb_i is negative then, reading from this segment is never allowed. Writes are analogous, but as restricted by the write bracket.[12] Typically, wb_i and rb_i are defined in such way that $wb_i \le rb_i$ holds, so that code can read what it has written.

Notice that (10.1) is satisfied by a segment $\langle n_i, \mathit{len}_i, m_i, rb_i, wb_i \rangle$ no matter what values are used to define the read and write brackets. Here is a proof. Consider a name n satisfying $n_i \le n < n_i + \mathit{len}_i$, and suppose that $L' \prec L$ holds

[11] In a hierarchically structured system, relation $\prec$ is total, irreflexive, asymmetric, and transitive. So it is impossible to have both $L' \prec L$ and $L \prec L'$ hold, and for every pair of layers L and L' either $L' \prec L$ or $L \prec L'$ holds.

[12] So `curRing` can be seen as a generalization of `mode`, with smaller values for `curRing` authorizing access to additional state rather than to additional instructions. The distinction between state and instructions can be ignored here, because the effect of executing any instruction is to perform reads and writes to main memory and processor state—restricting access to state is thus equivalent to disallowing execution of an instruction.

too. We can prove that $read(L') \supseteq read(L)$ in the consequent of (10.1) holds by showing that if $n \in read(L)$ holds then so does $n \in read(L')$. If a read to some name n succeeds while layer L is executing (so `curRing` $= ring(L)$ and $n \in read(L)$ hold) then $0 \leq ring(L) \leq rb_i$ must be satisfied. We have $0 \leq ring(L') < ring(L)$ from $L' \prec L$ and (10.2), so $0 \leq ring(L') \leq rb_i$ follows from $0 \leq ring(L) \leq rb_i$. Thus, reading n while executing in layer L' does not cause an access violation: $n \in read(L')$ holds, as we needed to show. The argument to prove conjunct $write(L') \supseteq write(L)$ in the consequent of (10.1) is analogous.

Operation Invocation. In hierarchically structured systems, a `call` instruction executed by layer L code is allowed to proceed only if (i) the destination is the gate for an operation Op exported by some lower layer L' and (ii) Op is not being hidden or redefined by any interposed layer L'' where $L' \prec L'' \prec L$. Processor support to enforce these restrictions can be controlled by incorporating information into descriptors. The descriptor for each segment i now would include:

- $nGates_i$, the number of gates that segment i contains. Entry points are enumerated in the first $nGates_i$ words of segment i: word 1 contains the address in segment i of the entry point for operation 1, word 2 contains the address in segment i of the entry point for operation 2, and so on through word $nGates_i$.

- xb_i, a non-negative integer that the processor loads into `curRing` whenever an instruction from segment i is being executed. So in order to specify that segment i stores code for a layer L then xb_i is set equal to $ring(L)$.

- cb_i, the largest value of `curRing` from which a **call** is permitted to a gate in segment i. By setting $cb_i = xb_i + 1$, operations defined by gates in segment i are hidden or redefined by the layer immediately above; and if $cb_i > xb_i + 1$ then higher layers can themselves directly invoke operations defined by gates in segment i.[13]

Execution of a "`call` *Op*" then proceeds as follows, where Op is presumed to be given as a segment name i and offset p in words. To start, the hardware checks that $p < nGates_i$ and $xb_i <$ `curRing` $\leq cb_i$ hold, thereby establishing that (i) Op identifies a gate in segment i and (ii) Op is visible to the layer executing the `call`. If these checks succeed then execution of the `call` stores a return address and the value of `curRing` into well known registers (which we call `retPC` and `retRing`, respectively), loads xb_i into `curRing`, and loads into the program counter the value found in word p of segment i (thereby branching to the entry point of Op). Code at that entry point is responsible for saving the contents of `retPC` and `retRing`, so that control can be returned to the caller when execution of the operation completes.

[13] This scheme does not offer the flexibility to specify that only certain specific operations that a layer exports should be hidden or redefined in higher layers. Such flexibility is attractive, but it is not part of the processor support modern CPUs typically provide for rings.

Note that care is required when referencing arguments—otherwise, confused deputy attacks (see page 101) are possible. Of concern is a caller that (i) is executing in some layer not in the read (write) bracket for the segment named in an argument *arg* and (ii) passes *arg* to an operation in some lower layer L that is. So the caller is not itself authorized to read (write) *arg* but, courtesy of a confused deputy in layer L, effects access to *arg* nevertheless. The obvious defense is for operations to check whether arguments are accessible to their callers. This check can be performed in software by inspecting the corresponding segment descriptor for each of the arguments. Some processors even offer a separate addressing mode to facilitate performing such checks. This addressing mode allows accesses to be made under a temporarily increased value for `curRing`, such as the value found in `retRing` when the operation started.

Layers versus Processes. Layers and processes are orthogonal constructs. Each provides a way to decompose a system. A process might be layered, or a layer might itself be implemented by a set of processes. The layer and process constructs differ in what isolation they enforce. However, each construct helps with the Principle of Least Privilege by enforcing isolation that restricts what parts of the overall system each individual component can access. So, in both cases, structure is being leveraged for defense.

