SOFTWARE INTEGRITY LEVEL PARTITIONING

A short tutorial on the basic architectural principles of integrity level partitioning

M. Squair

www.criticaluncertainties.com

This tutorial is drawn from a number of sources, whose assistance I gratefully acknowledge

Except for images whose sources are specifically identified, this copyright work is licensed under a Creative Commons Attribution-Noncommercial, No-derivatives 4.0 International licence. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/
Partitioning

- A short tutorial on the basic architectural principles of integrity partitioning
Outline

- Overview
- Definitions
- Architecture
- Separation design methodology
- Spatial partitioning methods
- Temporal partitioning methods
- Operating system partitioning methods
- Distributed system partitioning methods
- Partitioning violation fault tolerance methods
- Assurance requirements and partitioning
- Conclusions
Overview
Overview - Architecture and the Lifecycle

■ Software architecture may outlast the system (DOS, Unix)

■ Fundamental paradigm shift from encapsulated SW to layered and/or partitioned software

■ Should software architecting be treated as purely an issue for software or as an integral part of system architectural design?
Overview - Architecture and the Lifecycle

- System architecture - from federated to integrated

Space shuttle avionics circa 1980’s

**Federated** – Every function has its own processor

**Integrated** – Backbone network with modular electronics and smart sensor/actuators connected by own communication buses

NASA Spider FT/IMA circa 2003
Overview – The Challenge of Integrating an Architecture

- Ideally we would like to keep the robustness of the federated system and its advantages in managing complexity while gaining the hardware and functional integration advantages of a modular integrated system.

- Or to put it another way, reduce costs while maintaining safety.

- We need to achieve high levels of safety i.e. if an aircraft has a maximum aircraft accident rate of $10^{-7}$ per hr then for a 100 systems architecture each system must achieve $10^{-9}$ per hr.

- This will not ‘just happen’ it needs to be designed in across the system development.
Overview – Software Hazards

■ Types of software hazards
  – Direct software component hazard – i.e. incorrect output due to internal design or requirement fault
  – Interaction style hazards – incorrect output because of inadvertent interactions between components – i.e. unbounded priority inversion

■ We would like to eliminate (if possible) interaction style hazards

■ One way is to partition/separate the components *

  If You Can Keep Them Separate (Partitioning)  
  Then You Can Bring Them Together (Composition)

■ The DO-178B view
  – Section 2.3.1 “If protection by partitioning is provided …”.
  – Partitioning is a means to protect components by separating them

* As a side benefit partitioning can also limit the consequences of transient hardware faults
Definitions
Definitions

- **Architecture**
  - A fundamental or unifying structure of thing, usually a building

- **Systems Architecture**
  - The fundamental and unifying system structure defined in terms of system elements, interfaces, processes, constraints, and behaviors.
    - INCOSE SAWG
  
  - The highest-level concept (or conception) of a system in its environment
    - IEE P1471 AWG
  
  - The structure of components, their relationships, and the principles and guidelines governing their design and evolution over time
    - Perry-Garlan, software, NOT IEEE 610.12
  
  - The organisational structure of a system or component
    - IEEE 610.12
Definitions

■ **Strict Protection**
  – Component X can be said to be strictly protected from Y if any behavior of Y has no effect on the operation of X

■ **Safety Protection**
  – Component X can be said to be safely protected from Y if any behavior of Y has no effect on the safety properties of X

■ **Two-way (symmetric) protection**
  – Component X is protected from Y, and Y is protected from X

■ **One-way (asymmetric) protection**
  – Component X is protected from Y, but component Y is not protected from X
### Definitions

<table>
<thead>
<tr>
<th></th>
<th>ONE WAY</th>
<th>TWO WAY</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STRICT</strong></td>
<td>Un-directional opaque interface</td>
<td>Separate hardware</td>
</tr>
<tr>
<td></td>
<td>CRC checks on received safety</td>
<td>Data received validity flags</td>
</tr>
<tr>
<td><strong>SAFETY</strong></td>
<td>critical messages*</td>
<td></td>
</tr>
</tbody>
</table>

*Assumes that loss of message is not a safety issue.
Definitions

■ Interaction Hazard
  – A component can effect the operation of other components by effecting the temporal (time) behavior or the data (space) of the other components
  – Similar to the ‘covert channels’ concept used by the software security engineering

■ Thread
  – A lightweight unit of program execution

■ Process
  – A heavyweight unit of program execution consisting primarily of a distinct address space, in which one or more threads may operate

■ Kernel
  – That part of the OS providing core system services such as scheduling, thread synchronisation and inter-process communication
Separation (Partitioning) Design Methodology
Separation Protection (Partitioning) Design Strategies

■ Separation protection strategy
  – Either one way or two way strict protection in time or space
  – Traditionally achieved via a single processor/program LRU and loose coupling between processors i.e. a federated architecture
  – This is very expensive in system resources and development cost*

■ Safety protection strategy
  – The protection strategy only concerns itself with safety properties
  – Identify all the time/space safety properties which could be affected
  – Demonstrate that the safety properties have not been violated

■ In partitioning, choose the elements so that they are as independent as possible, that is, elements with low external complexity and high internal complexity

*It can also reflect a contractual relationships i.e. aircraft to LRU vendors
Partitioning Rigor

- Can be software or hardware protection, in some cases a combination of both mechanisms

- Less rigorously for lower integrity levels use only compiler and linker mechanisms to implement partitioning
  - Gives us partitioning by design
  - But we rely on the correctness of these tools

- More rigorously
  - Mutually exclusive hardware to achieve isolation and protection
  - Hardware memory management units are an example for space protection

- The acceptability of selection of the partitioning solution should be determined relative to the software integrity level
Separation/Partitioning Design Process

- Hazard identification
- Risk assessment and management

System hazard analysis

Identify fault propagation channels
- Direct interaction (data exchange)
- Use of shared resources

Establish fault protection boundaries
- Hard partition (no interaction)
  - Containment strategy
    - Virtual Machine
    - Memory MMU
    - Cache management
    - Execution time monitors
    - Data wrappers
    - Run time evaluation
  - Soft partition (interaction)
  - Mediation strategy
    - Watchdog monitors
    - Degraded mode
    - Procedural control
    - Protect critical code (Bankers algorithm)
    - Scheduling analysis (RMA)

Establish ‘containment Violation’ fault recovery strategies
- MMU detection
- Default jump for unused memory
- From/to programming
- Task timeouts

How do we?

Mediation presumes there is some level of acceptable interference

Risk derived integrity requirements
Identifying Direct (Data) Interaction Hazards

- Hazards caused by the direct exchange of corrupted data
- Unexpected (illegal) inputs in time or value
  - Timing/value related – i.e. input received before start-up, out of range value
  - Specify complete behaviour i.e. a response to unexpected values
  - See the work by Leveson, Jaffe inter alia
- Expected (legal) inputs that are wrong
  - Much more difficult
  - Reasonableness checks

- **Ask the questions**
  - What is the worst thing that other elements could do to you across an interface?
  - What mechanism will detect and stop such a behaviour?
Identifying Shared Resources Interaction Hazards

- Two approaches, categorisation or resource usage

- Categorisation (top down)
  - Categorise as either space/time all resources
  - Develop a set of properties for each class that would assure partitioning (guarantee protection boundaries)
  - The problem is that the lists are not orthogonal i.e. an incorrect data value may cause a timing error, how would we categorise this?
  - In practice a list is usually used for each category to develop controls

- Resource usage list (bottom up)
  - Identify and list resources used by components
  - Identify the potential ways in which resources can form ‘covert channels’
  - Requires intimate knowledge (design) of hardware and software low and middleware

- Both approaches suffer from the completeness problem
Timing Interaction Style Hazards

■ Consequences (failure mode)
  – Loss of input or output data
  – Corruption of input or output data
  – Corruption of internal data
  – Delayed data

■ Causes
  – table overruns* or incorrect links
  – Program overlays
  – Buffer sequence (double jeopardy)
  – External device interaction (e.g. displays) i.e. protocol halts
  – Control Flow defects (space aspects) such as jump table corruption

*The Phobos I spacecraft was lost in this fashion when a keyboard buffer overflowed into the memory of a critical flight control function
Timing Interaction Style Hazards

Causes continued

- Interrupts and interrupt inhibits (software and hardware)
- Loops (e.g. infinite loops)
- Real time correspondence (frame overruns etc)
- Control Flow defects (timing aspects)
- Memory, I/O contention
- Data flags
- Software traps (divide by zero, un-recognised instructions etc)
- Recursion termination
- Indirect non terminating call loops
- Holdup commands (performance hedges)
Spatial Partitioning Methods
Separation In Space

■ Two broad sets of techniques
  – Memory partitioning
  – Software Fault Isolation

■ DO-248B identified space partitioning issues
  – Protection of code memory, data memory, registers, and input/output buffers
  – Persistent storage locations (e.g., data memory), assigned to a software partition, write-able only by that partition
  – Context data (e.g., processor registers, CPU-caches) used by a task preserved or flushed as appropriate when control is transferred to another partition
  – Data flow and communications between partitions
Virtual Machines

The virtual machine concept

- Partition above a kernel (basic executive) rather than above the OS layer

- Current pure VMs have significant performance (and cost) penalties
- Partial VM can provide abstract services and reduces analysis overhead
Virtual Machines

- Partial VMs
  - For non instruction related services such as I/O an API can be used
  - Abstracts the interface
  - API design can include partitioning of components
  - API component integrity can be formally/rigorously argued

- Memory management can be developed in a similar fashion
  - Enforces spatial separation
  - Pre-emptive management of stack/heap memory allocation by kernel
  - Threads required to allocate memory from within own budget to spawned threads (avoids Unix ‘fork bomb’ style hazard)
Memory Quota Methodology

**BAD**

Fault driven dynamic process creation can still exhaust memory

**BETTER**

Threads must use own memory allocation for spawned thread

**BETTER STILL**

Thread A'
Hardware Memory Management Units (MMU)

- Spatial partitioning can be provided by HW MMU
  - All accesses to memory addresses are either checked or translated using tables held in the MMU
  - Attempts to access illegal addresses can be blocked or initiate fault recovery
  - Kernel ensures that memory locations that can be read and written in each partition are disjoint (except for designated IPC channel memory)
  - Kernel saves all registers on context switch*
  - Kernel uses the MMU to protect itself from modification by applications**

- Context overhead issues
  - Restoration (restore registers) versus restart (initialise registers) of task
  - Assumption of ‘dirty’ registers on restart
  - Are hidden registers (pipeline & caches) really hidden?

* The assumption of atomic read/writes to a register needs to be confirmed.
**Entry/exit from kernel needs to be handled very carefully to ensure that an application does not gain supervisor mode, some processors have design flaws that allow this
Hardware Memory Management Units (MMU)

Unmapped page at end of stack will catch thread overflows by forcing a HW memory overflow fault.

Catches NULL pointer dereferences.
Hardware Memory Management Units (MMU)

■ MMU design considerations
  – MMU are often more complex than required for embedded applications
  – Large look up tables makes them vulnerable to bit flip SEUs
  – Heavily optimised for speed (speculative, out of sequence execution)
    • Pre-emptive page read and roll back
    • Use of multiple lookup tables which require consistency checking
    • Timing uncertainty is introduced by these optimisations
  – SEU in the MMU can be dangerous
    • Likelihood of an SEU is increased as the size of the MMU increases
    • Need to address MMU fault tolerance in the face of SEU

■ MMU detection of memory reference errors during design
  – Major source of Heisenbugs
  – Subtle and hard to isolate during testing
Inter-partition Communication

- Pass data *that is authorised for the system configuration* using buffers

**Kernel managed copy of buffer**

If operating in unprotected mode,Kernel must check for overflows.

**Shared common buffer address space**

Receiver must assume arbitrary writing of data Anywhere in common buffer

**Unidirectional buffer is simplest**

Using a single buffer could cause partition C to overwrite B’s data intended for A
Inter-Partition Communication

- Partitions should send/receive data only if that communication is authorised in the system specification.

- How do we reference and identify data sink/sources?
  - Absolute reference (send to partition A) – Fragile/rigid system vulnerable to system changes
  - Functional (output to auto-throttle) – Making assumptions about system architecture and limiting reuse
  - Relative addressing (my port I) – Allows late binding but requires a DB to provide a lookup of what data is on what port
  - Service/data (send air_data) – Allows late binding to IPC channels, provides a proto-publish/subscribe style architecture
Inter-Partition Communication

- **Timing issues**
  - Applications should not make assumptions as to when other partition applications are scheduled
  - Communication may be asynchronous
  - Introduces data latency issues, traditionally handled with a time stamp

- We don’t wish to export complexity across partitions to handle latency issues (so we should handle at the source)

- **Temporal firewall (Kopetz)**
  - Phase-insensitive data – provide a time and a guarantee of accuracy up-to that time
  - Phase-sensitive data – where state estimation must be used provide both the value and additional state data that can be used to estimate the value
Handling Devices

■ Three key spatial partitioning issues for devices
  – Protection against access by the wrong partition
  – They must not themselves become covert channels for violating partitions
  – They may need to be partitioned

■ Sensors/actuators
  – Normally I/O is memory mapped to specific registers which can be accessed like normal memory

■ Partitioning strategies
  – If possible treat as same as other memory for partitioning
  – If not create a device handler partition (separation within partition will still need to be verified)
  – For devices on a bus or attached to a device concentrator use a device handler partition especially for multicast services
  – Handled by kernel (but this makes the kernel more complex)
Sharing Device Across Partitions

- Sometimes devices must be shared
  - Data input is fairly straightforward
    - For example a safety shutdown function and a control function in separate partitions both monitor the same sensor
    - In this case access can be shared by both
  - Where commands can be issued sharing is more problematic
    - A fault in one partition can lead to erroneous data being interpreted as a valid command*
    - Protection by a device handler partition may be necessary in this instance
  - Mass storage devices pose particular problems
    - Device handler would generate a ‘file service’ for each partition mapped to separate physical memory
    - DMA of mass storage devices may be able to circumvent MMU
    - Physically limit DMA device to memory of handler partition or isolate device to private bus with a dual port bridge to main bus

*The Clementine spacecraft was lost when a software fault caused garbage to be sent over an unmediated bus, where it was interpreted by an attached device as a command to fire all the thrusters without limit.
Still requires argument that Device separation is maintained
Software Fault Isolation (SFI)

- Similar to array bounds checking in HLL
- Applied to all memory not just memory indexed into arrays
- Static analysis technique
- Examine machine code of partition
  - Determine if jump destinations or memory references are safe
  - Indirect references via registers are checked at runtime (added code)
  - Can optimise runtime overhead by only checking updated registers
  - Reported 4% additional overhead

- Can greatly reduce the cost of controlled references or procedure calls across partitions – i.e. no context switch
- Does impose additional cost of implementation vs hardware mediation
- But can be automated i.e. ‘proof carrying code’
Software Fault Isolation (SFI)

- Possible uses
  - Asymmetric application i.e. safety function + some nice to have functions in a single partition, use SFI to isolate the two
  - Provide assurance between to Level ‘A’ functions
  - Provide additional assurance between threads in a single hardware mediated partition

- A small increase in likelihood of hardware generated SEU faults
Temporal Partitioning Methods
Partitioning in Time

- Two levels of scheduling
  - Scheduling at the partition level
  - Scheduling at the task level

- DO-248B identified time partitioning issues
  - Protection of the processing and communication assigned to a partition
  - The consistent order of execution between communicating partitions
  - Deterministic scheduling (processor and communication)
  - Guaranteed access for each software partition to a prescribed set of hardware resources
    - for a prescribed period of time,
    - at a prescribed rate, and
    - at a prescribed point in time (if necessary)
Gross Timing Problems

- Gross timing problems
  - System crashes, hangs or is halted by partition

- Usually handled by spatial partitioning mechanisms
  - In User mode ‘halt’ commands are trapped by the kernel
  - But for some processors there may be user-mode instructions that can hang when supplied with certain parameter (see 80X86 µp)
  - May have to insert run-time checks for dangerous commands

- If all else fails a watchdog timer can be used
  - Basic kernel design has to be correct*
  - Hardware resident

---

*A fault in the Magellan spacecraft led to a runaway execution in which a program sat in a loop that did nothing but reset the watchdog timer*
**Gross Timing Recovery Issues**

■ Seizing control from a partition may not guarantee continuation
  – Partition may have locked the resources
  – Even if the kernel seizes the resources they may have been left in an inconsistent state

■ Reinforces the design principle that resources should not be directly shared across partitions

■ Use of a mediating partition is preferred

■ Fault tolerance by invoking partitions is also required

■ Kernel fault recovery action may be required
  – Force a partition restart on next cycle
  – Notify partition and allow partitions OS to take action
Temporal Partitioning and Scheduling Policy

■ Tasks need to get access at the right time and with high dependability
  – Tasks may be cyclic (iterative) or sporadic
  – Cyclic timing may have jitter constraints
  – Sporadic timing may have deadlines or latency constraints

■ Scheduling of tasks
  – Static – established before runtime but rates may be mode dependent
  – Dynamic – established at runtime based on task priority and interruption scheme

■ Dynamic scheduling is dominated by Rate Monotonic Analysis
  – Introduced by Liu and Layland in 1973
  – Original restrictions have been eased to allow a-periodic events and arbitrary deadlines
Temporal Partitioning and Scheduling Policy

■ Rate Monotonic Analysis (RMA)
  – Highest priority goes to highest rates
  – As long as memory usage does not exceed 69% will meet deadlines
  – Can go higher if other restrictions are met
  – Best of all provable by analysis, Project Managers love this…

■ There are arguments for and against static or dynamic scheduling

■ But partitioning affects both scheduling policies
  – Need to ensure that fault assumptions about temporal behaviour in one partition don’t affect another
  – Therefore (logically) need to schedule time to partitioned tasks
Scheduling Policy in a Partitioned Environment

- Two strategies

- Scheduling by partition
  - Two level approach
  - The kernel guarantees each partition a certain amount of time
  - Partition arbitrarily budgets time internally

- Task quota
  - Time is budgeted to tasks
  - Kernel uses a quota system to limit faulty task impacts to containing partition
Partition Level (Static) Scheduling

- **Partition scheduling**
  - Kernel statically schedules a partition
  - Partition internally schedules tasks in any arbitrary fashion
  - Can allocate tasks running at different rates to different partitions if using a restoration strategy
  - Rigid scheduling limits ability to add new tasks i.e a 10Hz task can only be added to a 10 Hz partition
  - Partition swap overhead grows as swap frequency increases
  - Need to address a-periodic event handling

- **Ensures predictability through temporal determinism**

- **This determinism imposes strong restrictions on event-driven activations**
  - Kernel time taken to latch an interrupt may affect current partition
  - Adding margin to partition schedule makes one partition dependent on the attributes of another (expected interrupt arrival rate)
Partition Level (Static) Scheduling

- The fundamental design principle of temporal partitioning is that
  - resources in each partition have access to the resources of the system at guaranteed intervals and
  - That those resources provide their expected performance

- To contain faulty partitions
  - No other partition should be able to initiate any activity that will compete with the partition scheduled to access the resource
  - This can be violated by a partition conducting an I/O action when terminated or by a spontaneous external interrupt arriving

- To contain these uncertainties
  - Poll or mask external interrupts (CPU dependent)
  - Mask interrupts generated by other partition I/O
  - Lock out I/O registers in last portion of partition
Partition Level (Static) Scheduling

- IPC must maintain independence of partitions
  - Asynchronous tasks
  - Mailbox or ‘message under the rock’
  - Limits on service request rates to handle ‘faulty’ processes requesting services at an unexpected rate

- What happens if there are no threads running in active partition?
  - Use a ‘background’ partition for low priority non time critical tasks (BIST)
  - Run background partition when current partition has no runnable threads
Task Level (Dynamic) Scheduling

- If we schedule partitions dynamically we may be able to relax some restrictions
- Devolve to the task as a unit of scheduling
- Temporal partitioning will drive scheduling
- The problem with scheduling theories
  - The most well understood and proven scheduling theories make a lot of simplifying assumptions (i.e. zero context switch overhead)
  - More complex theories account for these issues but are less well understood
  - Faulty partitions may violate the theorems assumptions
- A quota system is one way to address these problems
  - Each partition is allocated a kernel enforced percentage of time
  - Need to account for context switching overhead as part of this budget
Task Level (Dynamic) Scheduling – Temporal Quota

- Temporal quota’s can guarantee that critical threads get the required processor time
- Still vulnerable to faults in thread scheduling

Scheduling on priority level

Critical thread ‘A’ starved

Scheduling using a quota system

Critical thread ‘A’ time guaranteed
Task Level (Dynamic) Scheduling

- We still need to address inter-partition interference
  - Cycle-stealing DMA transfers
  - Masking/latching of interrupts for a partition that has exceeded its quota
  - Locks and semaphores

- Temporal deadlines must still be assured
  - Handle low jitter events using highest priority task* or statically scheduled partitions
  - Other tasks (such as control law computations) would require only guarantee of resources sufficient to process at the appropriate rate

- Thread scheduling faults
  - What if thread ‘A’ lowers its own priority (due to a design fault)?
  - What if thread ‘B’ raises its own priority (due to a design fault)?

*This would require a formal and complex scheduling analysis to justify
Operating Systems and Kernel Partitioning Methods
Kernel and Operating System Issues

- Where an OS exists on top of a kernel and provides services to a number of partitions
  - The services provided by the Operating System may be used by different tasks in different partitions
  - This may occur deliberately in the case where a task using a system service is interrupted (e.g., software partition deadline expires)
  - It may also occur when a system service is not released correctly

- Partitions must use system services without knowledge of other partitions and their service utilisation

- Therefore, there should be a means to protect and recover the system services or provide for degraded operations for all software partitions resident on a single processing resource
Kernel and Operating System Issues - System Call Hazards

- Many kernels return the actual pointer to new kernel objects
  - When pointer passed back to kernel it may be directly de-referenced
- Potentially dangerous as threads may
  - modify the kernel object directly or
  - Overwrite kernel handle with reference to some other memory
- Bad system calls should not be able to crash the kernel
- Drives requirement for opaque handles for all kernel objects
- Kernel should automatically validate the parameters to all system calls
Kernel and Operating System Issues – Interrupt Handling

Kernel provides some of the services of an OS, at a minimum service interrupts

Most disable interrupts while manipulating internal data
  – Avoids context switching when internal kernel data is being changed
  – Interrupt latency is proportional to length of critical section in the kernel
  – Timer tick interrupt is usually the reason this policy is applied

Handling interrupts for partitions can be challenging
  – Vector direct to a partition if statically scheduled (simple kernel)
  – But if they require supervisor mode introduces potential for partition to clobber the kernel
  – Alternatively kernel handles and passes opaquely to the partition (complex kernel)
Kernel and Operating System Issues – Device Handling

- Handling devices is ‘really’ a traditional OS service
  - Allocate OS services to each partition
  - Tailor to functions of each partition
  - If a partition encapsulates a shared service (file server) use with care
  - These are extensions of the kernel and must be shown to partition their services appropriately adding an OS makes this a more complex task

- Access rules for certain devices must be mandated to assure safety (both inclusive and exclusive)

- Kernel must enforce these rules to ensure that application does not break them
Kernel and Operating System Issues

- **Partition A**
  - Safety monitor & shutdown
  - Mini-OS (Formally verified)

- **Partition B**
  - Plant Display Generator
  - COTS OS

- **Partition B**
  - Devices Handler
  - Tailored OS Services for Device handling

- **Mass Storage**
  - Highest integrity level (formal argument)
  - Mid-range integrity level (rigorous argument)

Tailored OS for partition services
Kernel and Operating System Issues – Execution Times

- Time budgets for threads includes system calls to kernel
  - Ideally kernel can bound execution times for system calls
  - Two problem areas message transfer and mutex take operation times

- Message timings vary with message size
  - How are the times allocated sender or receiver or both?
  - Kernel should have capability to control such allocation
  - Kernel scheduler should treat messages as prioritised unit of execution

- Mutex take operations
  - Priority inversion problem, invalidates RMA guarantees of temporal partitioning
  - Priority inheritance can resolve this violation of temporal separation
Kernel and Operating System Issues – Execution Times

High priority task
Medium priority task
Low priority task

1. Low priority task
   Holds Mutex
   & high priority task cannot run

2. Low priority task
   cannot execute
   and passes Mutex
to medium task
   which can run

3. High priority task
   locked out by Mutex

BAD

2. High priority call
   for Mutex elevates
   Low priority task to High

3. Medium priority
   task blocked and
   high priority task
   gets Mutex

BETTER

Does not prevent chained blocking
where a ‘chain’ of priority elevation
occurs i.e. low priority elevated to medium
then medium to high by calling sequence

1. Low priority task
   priority drops
   when Mutex released

1. Low priority task
   Holds Mutex and is
   assigned high priority
   while it holds it

2. Low priority task
   priority drops
   when Mutex released

BETTER STILL

Priority ceiling implementation
Mutex assigned priority of
highest priority task that could
use it. When a thread takes
Mutex it inherits this priority
preventing chained blocking

(Ada’s protected objects
implement a priority ceiling)
Kernel and Operating System Issues – Context Switches

- Context switch time is overhead
  - Not available for mission functions
  - Decreases likelihood of meeting deadlines

- Hand optimise the context switching code to increase speed

- But, this may make formal proof more difficult

- Also makes code more idiosyncratic and dependent on application context
DO-178B CAST Guidelines for OS Partitioning

■ A software partition should not be allowed to contaminate another partition’s code, I/O, or data storage areas

■ A software partition should be allowed to consume shared processor resources only during its period of execution.

■ A software partition should be allowed to consume shared I/O resources only during its period of execution.

■ Failures of hardware unique to a software partition should not cause adverse effects on other software partitions.

■ Software providing partitioning should have the same or higher software level than the highest level of the partitioned software applications.
Partitioning Methods for Distributed Systems
Distributed Systems and Communication

If partitions could be point to point connected demonstrating partition would be straightforward.

However,

In practice we connect processors not partitions usually through a common data bus (channel).
Partitioning

- Need to consider both temporal and spatial partitioning

- A bus is a departure from strict partitioning
  - We must provide partitioning so that a fault in one partition or processor cannot affect others

- For a bus the critical partition or processor faults are
  - Babbling idiot, or
  - Failure to follow the access protocol
Temporal Partitioning and Babbling Idiot Faults

■ A transmitter sends data constantly
  – Overwhelms the receive
  – Denial of service to other transmitters

■ If it occurs at the partition level
  – A global schedule of partitions could solve this
  – Partition can only babble in its slot
  – Schedule to ensure that no two processors simultaneously schedule partitions that transmit to the same recipient
  – Implement a quota system within a device management partition

■ If it is a babbling processor then either
  – Tolerate it, or
  – Terminate it.

■ These are all temporal partitioning issues
Spatial Partitioning and Authorised Transmissions

- Also need to ensure that only authorised messages are sent and received
  - Correct addressee receives the message
  - Message is ‘legal’ for that configuration

- Recipient can be explicitly identified (event) or time (time sliced protocols)

- Fault tolerance on addressee ID is required to maintain spatial partitioning (checksums)

- Increase hamming distance on addresses to prevent right message wrong addressee type faults

- Timing based schemes may require bus mediation

- Alternatively use a ‘publish/subscribe’ architecture
Bus Mediator

- **Addresses the babbling processor problem**
  - Time versus event triggered protocols

- **Time based mediation**
  - Mediator has a clock (synchronised with its processor) and lookup table
  - Checks and releases processor messages
  - Local versus global clock synchronisation (global is more complex)
  - If mediator is local but system is globally synchronised then if processor loses synch it will take its mediator with it
  - Add in a fail silent capability if mediator fails to receive ‘X’ messages

- **Event based mediation**
  - Much weaker
  - Can use a master/slave protocol (single a-priori bus manager)
  - Dual check processing
  - Timeouts (1553) on transmission, and limits on rate (ARINC 629)
Distributed Systems and Redundancy

- Communication of node failures elsewhere is required
- One method is to provide ‘heartbeat’ in mechanisms
- If the redundant node fails to receive a heartbeat then failure of the active node is assumed
- Need to consider bus reliability and start-up/shutdown exclusions
Partitioning Violation Fault Tolerance Methods
Partitioning/Protection Violations

- **DO-178B (Section 11.10j)**
  - A design must address the potential breaches of protection/partitioning
  - Adequate error control should be provided to contain breaching errors

- **DO-178B (Section 6.3.3f & Obj.13 of Table A-4)**
  - Verification of a partitioning/protection integrity requirements

- Some aspects of partitioning are very close to fault tolerance
  - For example the control of babbling partitions
  - Functions often need to be fault tolerant themselves and a partitioned architecture needs to support this

- **Functional allocation issue (architecture vs function)**
  - Sensor failure (function)
  - Babbling partitions (architecture)
  - Processor failure (both?)
Fault Tolerance and Architecture

- Fault tolerant architectures differ
  - Homogenous or diverse redundant replicas of computation
  - Synchronous (voting) or asynchronous states
- Some architecture choices are strongly contingent

- Design freedom is curtailed if partitioning is a design driver
Fault Tolerance and Architecture

■ General principles
  – Knowing that a failure has occurred is important
  – Plan for a fast recovery from a single failure
  – Remember that complexity increases (and difficulty) as the number of near coincident failures to be tolerated increases
    • Examples 777 FBW & LandSat-7
    • Permanent vs transient fault (quasi permanent state errors) tolerance
    • Massive coincident faults (lightning strike, HIRF) tolerance
    • Byzantine faults (often ignored)

■ Fault tolerance architecture
  – Employ supervisory thread in separate partition
  – Kernel detects thread faults and refers them to the supervisor
  – Supervisor embodies system safety policies and responds to thread fault
  – Supervisor can handle (software) watchdog timer monitoring as well
Assurance Requirements and Partitioning
Certification Basis

■ **Federated Systems:** Traditionally certification is of the whole
  – All Components considered together
  – Verify safe joint behaviour
  – “You don’t certify a single application, you certify an entire system”
  – Relies on low levels of apparent interaction

■ **Integrated Modular Systems:** Incremental certification
  – Functions verified once, independently and strictly to their criticality level
  – Composition of functions retains individual certification
  – **BUT**
    • Need to address functional interaction
    • Certification of the partitioning strategy achieves this
Assurance of Partitioning Integrity

■ Traditionally (i.e. DO-178B)
  – Apply a rigorous development method, lots of reviews, documents and testing
  – One weakness of this is that software is discontinuous consequently, extrapolation from tested to untested cases is of doubtful validity
  – Compares to physical systems where continuity can be assumed and extrapolation from small sets of data made

■ Alternatively or as a support to traditional test based assurance
  – Apply formal (i.e. formal deductive logic) methods to validate partition attributes of the architecture
  – Examine (model) all behaviour
  – Can develop correctness (strictest) or safety properties (weaker) proofs
  – Assist in design choices
  – Easier if we keep architecture elements (kernel etc) small and simple
Conclusions
Conclusions

- Partitioning is a strong system requirement
- Practical application of the separation of concerns design principle
- Implementation issues/assumptions can be critical
  - Assumption of atomic actions
  - The presence of hardware/microcode design faults
- Interacts with and imposes many restrictions on system design
  - Fault tolerance
  - Scheduling
  - Distributed system design
- But, there are many differing methods to achieve partitioning
  - Especially for temporal partitioning