# CS 6210 Spring 2023 Test 1 (120 min Canvas Quiz)



## Table of Contents

## OS Structure [35 points]

### I. SPIN [11 points]

1. [3 points] [True/False with justification] Distinct protection domains require distinct hardware address spaces.

False.

A specific implementation of an OS may choose to pack multiple protection domains in the same hardware address space. E.g.: In the SPIN OS, all the logical protection domains are contained in the same hardware address space.

+1 – False with incorrect justification
+3 - False with correct justification

2. [4 points] SPIN promotes implementing an entire OS in a strongly typed high level language like Modula-3. Give 2 examples scenarios in which the OS may have to go outside the boundaries of language enforced protection model.

Modifying architectural features(e.g hardware registers in the CPU, memory mapped registers in device controllers) may require a service extension to step outside logical protection domain.

A significant chunk of the OS codebase is from third party vendors and it is difficult to force everyone to use a specific programming language.

+2 - For every correct example scenario with justification
+1 - For every partially correct example / not enough reasoning

3. [4 points] Give any two techniques that underlie SPIN's ability to achieve their goals of extensibility, safety, and good performance.

- Co-location: Extensions are dynamically linked into the kernel's hardware address space ensuring performance.
- Enforced modularity: Using a strongly typed language (in this case Modula-3) that provides compile time checks and runtime verification for the safety of the extensions.
- Logical protection domains: Extensions being bound in Modula-3 objects have the necessary guarantees of memory isolation from other subsystems, yet the resolve mechanism in SPIN dynamically stitches these extensions together to allow performant cross-domain communication.

Events and event handlers: Ability of extensions to associate handlers with the events which are software interrupts (or upcalls) from SPIN into the extensions.+2 - For every correct technique

## II. Exokernel [11 points]

1. [6 points] A library OS implements a paged virtual memory on top of Exokernel. An application running in this OS encounters a page fault. List the steps from the time the page fault occurs to the resumption of this application. For this problem assume the following:
   - The processor architecture has only a TLB for address translation (i.e., a TLB miss triggers a page fault)

- The library OS has already acquired all the necessary capabilities from Exokernel (writing to the memory mapped device controller registers to start the DMA from the disk to a memory buffer)
- The library OS has a pool of free page frames already pre-allocated to it by Exokernel

i. Page fault is first received by the exokernel which in turn invokes the handler registered for this event by the library OS in the PE data structure. (+1)
ii. Library OS finds a free page frame, running page replacement algorithm if necessary. (+1)
iii. Library OS calls its hard disk driver to initiate I/O from disk to a physical page frame. (+1)
iv. Device driver presents the capability for the hard disk controller and page frame to Exokernel to perform the DMA. Once DMA is complete, Exokernel uses PE to upcall the library OS. (+1)
v. Once Library OS gets the interrupt, it calls the TLB update primitive available in Exokernel to put (VPN, PFN) mapping into the TLB passing along the capability it has for the PFN. (+1)
vi. If the capability is valid, Exokernel installs the mapping and resumes the faulting process. (+1)

2. [2 points] [True/False with justification] When exokernel receives an interrupt and the library OS to which the interrupt is to be delivered is not currently running, exokernel ignores the interrupt.

+1 - False.

+1 for either (a) or (b) - (a) Either exokernel automatically executes the handler for that interrupt if the library OS has downloaded the handler code into exokernel, (b) or exokernel delivers the interrupt to the library OS when that library OS is scheduled to run on the CPU during its time slice.

3. [3 points] Give ONE similarity and ONE difference between the approaches to extensibility of SPIN and Exokernel

Similarity:
Ability to download code into Exokernel is functionally equivalent to extending the reach

of SPIN to include system services into the same hardware address space.

Difference:
The library OS can be coded in any language in Exokernel; SPIN requires the library OS to be coded in Modula-3 since it enforces protection using language-enforced logical protection domains.

+1.5 for Any VALID similarity
+1.5 for Any VALID difference

## III. L3 Microkernel [6 points]

1. [2 points] [True/False with justification] In designing the L3 microkernel, Liedtke further strengthens the conclusions of SPIN/Exokernel. Justify your answer

False.
In his paper, Liedtke demonstrates that microkernel architecture itself isn't inherently bad for performance and the fact that SPIN/Exokernel use Mach as an exemplar for microkernel performance isn't justified since Mach focuses on portability and not on performance.

2. [2 points] If a CPU architecture does not support address-space tagged TLB, a TLB flush is unavoidable when moving from one user-level protection domain to another. Justify your answer.

False.

There may be other architectural features that would allow packing multiple logical protection domains into the same hardware address space. E.g., segment registers in x86 architecture.

True.

If the architecture has no mechanism other than the hardware address space to provide memory isolation for a logical protection domain, then the TLB flush is inevitable.

3. [2 points] To implement a high-performance microkernel, the microkernel abstractions should be architecture independent. Justify your answer.

+1.0 As Liedtke argues in his paper, to achieve high performance, the microkernel implementation should fully exploit the hardware features available in the processor architecture.

+1.0 Abstractions, by their very nature, should ideally be architecture independent. With this said, having abstractions that are

dependent on the architecture would not necessarily impact the performance of the microkernel negatively.

# Virtualization [35 points]

## IV. Paravirtualization [14 points]

1. An architecture supports a TLB that is entirely software-managed in kernel mode. The size of the TLB is an architectural parameter that is published and known to the system software. The ISA of the processor provides the following instructions:

- Set-AS(AS-ID): Stores the AS-ID in a register called Address Space Register (ASR) that can be accessed only in kernel mode.
- Enter(AS-ID, VPN, PFN): Stores the tuple <AS-ID, VPN, PFN> in the TLB. If the TLB is full, this instruction will fail silently.
- Delete(AS-ID, VPN, PFN): Deletes the tuple <AS-ID, VPN, PFN> from the TLB. If the entry is non-existent the instruction is a NOP.

You are designing a hypervisor for supporting para-virtualized guest-OSes on top of this architecture.

i. [2 points] What advantage does this architecture have over x86-style architecture (that has a page-table plus hardware managed TLB) for implementing a para-virtualized guest OS?

+2 No TLB flush needed when switching from one guest OS to another

ii. [2 points] How would you provide memory isolation/independence/integrity for each guest OS?

+ 2 When a guest OS starts up, the hypervisor gives it a unique Address-Space ID.

iii. [6 points] How can each guest-OS provide memory isolation/independence/integrity for processes created within it?

- Provide a Create-AS() API in the hypervisor. This call will result in the hypervisor creating a new Address-space ID (AS-ID) and returning it to the guest-OS. The hypervisor will remember ALL the AS-IDs given to a specific guest-OS. [+2 points]

  + 2 For mentioning that ASIDs will be unique per process, and the hypervisor will have a role in ensuring uniqueness.

- Provide a Switch-AS(AS-ID) API in the hypervisor. The guest-OS will use this call to context to this process. The hypervisor will first check if this is a valid AS-ID that belongs to this guest-OS. If ye, it will facilitate the

context switch by calling the Set-AS(AS-ID) instruction in the ISA.

+ 2 For mentioning that the Guest OS will use a hypercall that internally uses Set-AS when context switching between processes.

- Provide a Insert-mapping(AS-ID, VPN, PFN) API in the hypervisor.
  - The hypervisor will first check if the AS-ID is a valid AS-ID that belongs to this guest-OS. If yes it will do the following. If not the API call will return an error to the guest-OS.
  - The hypervisor will first check if the TLB is full (it has to do this book-keeping to make sure that there is still space in the TLB since it knows the architectural limit of the TLB). If it is full, it will call Delete(victim-AS-ID, VPN, PFN) choosing an appropriate victim-AS-ID to delete that entry from the TLB (victim selection is left to the hypervisor, a simple one will be a random choice).
  - The hypervisor will use the ENTER(AS-ID, VPN, PFN) instruction in the ISA to insert this mapping into the TLB.
  - The Guest OS will use this call to allocate a new page in the process update space

+2 For an explanation in which the guest OS uses a hypercall which internally uses Enter to create a TLB entry for allocating a new page for a process.

+1 (grace point) For demonstrating an understanding of how memory isolation is achieved in a hypervisor

2. [HIDDEN] Answer the following questions with reference to Xen's I/O ring data structure. The I/O ring is a circular buffer used by Guest Operating systems for enqueuing requests for the Hypervisor, and collecting its responses.

i. [2 points] Why must the request producer and response consumer pointers be shared between the Guest OS and the Hypervisor? How can they be shared without mutual exclusion?

**(Editorial NOTE: we made an error in saying "request consumer" instead of "request producer". So we are ignoring student answers to the "request consumer" part (it is private to Xen). So the rubric for grading ONLY focuses on student answer to "request producer".)**

+1 The request producer marks the end of the enqueued requests that the hypervisor is supposed to process, and must be visible to the hypervisor.

+1 It can be shared without mutual exclusion, because only one entity writes to each pointer.

ii. [2 points] Explain why the hypervisor will never run out of slots for enqueuing responses in the ring buffer.

+2 The hypervisor dequeues a request placed in the ring buffer by a guest-OS. When it is ready with a response it will place it in the same spot where it dequeued the request from.

+2 For any answer which demonstrates the understanding that a slot is automatically created for a response when a request is dequeued

## V. Full virtualization [8 points]

1. [2 points] [True/False with justification] A fully virtualized environment with multiple Guest OSes has a single shadow page table.

False.

The shadow page table is the data structure needed by the hypervisor in a fully virtualized environment to associate machine pages allocated to a specific guest-OS to the illusion of physical pages maintained by the guest-OS. Therefore, there will be as many shadow page tables as the number of guest-OSes that are currently running on top of the hypervisor.+2 False with valid justification for number of shadow page tables

+1 False with mention of there being as many shadow page tables as the number of guest OSes

+0 True or False without valid justification

2. Assume that a fully virtualized OS is running on top of x86 processor that has a page table and TLB. The guest-OS maps a process's virtual page number (VPN) to a physical page number (PPN). However, PPN is an illusion of the guest-OS which must be translated to the machine page number (MPN).
   i. [2 points] How is this illusion of physical memory being contiguous for the guest-OS not result in inefficiency in a fully virtualized setting?

      Whenever the Guest OS services a page fault of a process, it attempts to install the mapping from the VPN to the PPN in the process' page table (a privileged operation since the Guest OS

runs in user mode); this results in a trap to the hypervisor which then updates the appropriate entry in the shadow page table with the mapping of the VPN directly to the MPN.

+2 Any answer that conveys the above sense, mentions page faulting process or otherwise conveys how mappings are installed in the SPT

+1 Mentions shadow page table that stores the VPN->MPN mapping

+0 Incorrect

ii. [2 points] How does the processor translate the virtual page number generated by the currently running process to its corresponding machine page number in memory?

In a fully virtualized setting, the hypervisor directly maintains the mapping between a virtual page number and its corresponding machine page number in the shadow page table. The hypervisor will set the PTBR of the processor to the shadow page table offset for the currently scheduled process for the currently scheduled guest-OS.

+2 Any answer that conveys the above sense
+1 Mentions hypervisor maintaining the VPN->MPN mapping in the SPT
+1 Mentions hypervisor updating the PTBR to SPT offset
+0 Incorrect

3. [2 points] State the difference between fully virtualized and para virtualized environments when it comes to updating the process page table after servicing a page fault.
   - In a fully virtualized setting, when the Guest OS tries to update a process' page table, it implicitly generates a trap into the hypervisor, which then updates the shadow page table with the mapping.

In a para virtualized setting, the Guest OS makes an explicit hypercall to the hypervisor ("Update-PT") to install the mapping[+2 points] Any valid difference

## VI. Memory Management [13 points]

1. [2 points] Explain concisely, how can the ballooning mechanism be used with a fully virtualized OS?

All that the ballooning mechanism requires is an installation of a balloon driver in the guest-OS. Modern OSes already have the ability (thanks to all the research in extensibility of OSes) to incorporate a device driver into it at runtime. So this capability applies to the fully virtualized OS as well.

2. [2 points] [True/False with justification] In VM oblivious page sharing, if there is a hash match between the page (call it incoming page) being brought in from the disk and an existing hash entry in the hypervisor's hash table, then the machine page associated with the incoming page can be safely discarded.

False.

The hash match is only a hint not an absolute. A full match of the two machine pages is needed to know that the contents are the same.

3. [2 points] Inflating the balloon driver of a Guest OS will result in more machine memory being allocated to that OS. True/False

False.

Inflating the balloon driver of a Guest OS will cause the Guest OS to page out unwanted pages of other processes to the disk to make room for the balloon driver's memory requests. This results in machine memory being deallocated from the Guest OS.

4. [HIDDEN] [5 points] There are two guest OSes (OS1 and OS2) running on top of the hypervisor. Assume that vpn1 of OS1 and vpn2 of OS2 are mapped to the same machine page (mpn1) through VM oblivious page sharing.

   How does the hypervisor handle modifications to vpn2?

Mappings to shared pages are always marked as copy-on-write. So, if OS2 makes a modification to the shared page, an access violation fault occurs, which shifts control to the hypervisor. [+2 points]

The hypervisor then creates another copy of the page (mpn2) and updates the mapping of vpn2 in the shadow page table to map it to mpn2. [+2 points]

The access right for vpn2 of OS2 is changed to r/w (removing the copy-on-write restriction). [+1 point]

5. [HIDDEN] [2 points] If a Guest OS has 2 processes running (in a fully virtualized setting), and each hardware page table has 1000 entries, then the size of the shadow page table for this Guest OS is (choose the right answer with justification):

(i) 1000 entries

(ii) 2000 entries

(iii) 3000 entries

(iv) None of the above

(ii), (iii)

2000 entries because there are 1000 entries in each hardware page table and 2 such hardware page tables (OS pages are mapped into each process's address spaces with appropriate protection through user/kernel mode). [+2 points]

OR

3000 entries, because there are 3 hardware page tables (two for the processes and one for the Guest OS). [+2 points]

# Parallel systems [32 points]

## VII. Atomicity [8 points]

1. [4 points] Consider a SMP with invalidation-based CC and whose Instruction-set Architecture (ISA) provides instructions for atomic read and atomic write. The OS supports multi-threaded applications and implements scheduling at the level of individual threads at available CPU cores. You have implemented the mutual exclusion algorithm as below:

```
// L is a shared variable
Lock(L):
  back:
  if (L == 0) L = 1; // success
   else
          While (L == 1); // spin
           go back;

Unlock(L):
    L = 0;
```

Will this lock algorithm work? If not, why not?

No, it will not work. Checking if L is 0 and then setting it to 1 needs to happen atomically. [+1]

Otherwise, a thread could find L is 0 (through the atomic read instruction), is yet to set it to 1 (using the atomic write instruction); but OS could context switch this thread to another one which also finds L to be 0 resulting in multiple threads eventually getting the lock. [+3]

2. [2 points][True/False with justification] An atomic read-modify-write instruction is not a necessity in the processor's ISA for implementing Anderson's mutual exclusion lock algorithm.

False. [+1]

To get a unique spot in the array to spin on, a thread needs to atomically read and increment the queuelast variable. [+1]

3. [2 points] [True/False with justification] An atomic read-modify-write instruction is a necessity in the processor's ISA for implementing any barrier synchronization algorithm.

False. [+1]

The tournament and MCS barrier algorithms do not need a RMW instruction.[+1]

## VIII. Barriers [11 points]

### A. Sense Reversing Barrier [3 points]

1. [3 points] Consider the sense reversing barrier implementation in C as follows where count, sense are shared variables which are initialized to N and False respectively. Will this code work? If yes explain why, if no, give brief reasoning and an example!

```
// global variables visible to all threads
bool sense = false;
int count = N;

// thread specific static variable
bool local_sense = false;

void barrier(){
      count--;
      if(count == 0){
            sense = !local_sense;
            count = N;
      }
      else{
            while(local_sense == sense);
      }
      local_sense = !local_sense;
}
```

No +1, with reasoning

+1 for each of the following reasons

REASON 1.

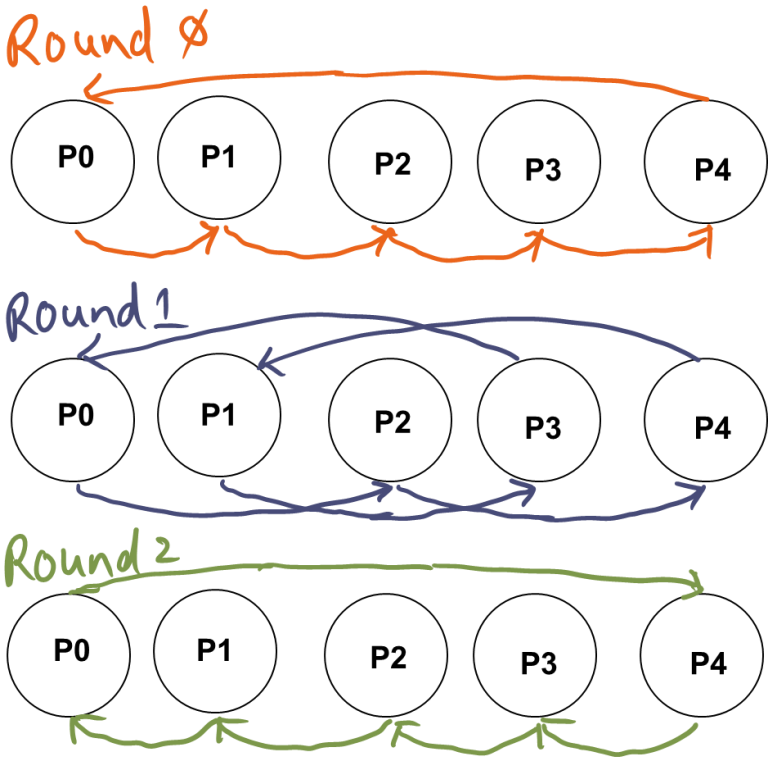
The correctness criterion for this algorithm is that "count--" should be atomically executed by each thread. Therefore, the correctness depends on how the C construct "count--" is compiled. If the underlying architecture has an atomic decrement instruction and "count--" is compiled with this instruction, then it will work. If not, it will not work since multiple instructions of the processor (read count, decrement count, write count) would be needed to compile "count--". Interleaved executions of these instructions across multiple threads will result in this algorithm breaking the necessity that "count--" should be executed atomically.

REASON 2:

The final process at the barrier modifies the shared sense variable before resetting the count back to N. This may cause the waiting processes to race ahead the decrement the count further before the final process of the previous round is able to reset the count.

**B. Dissemination Barrier [8 points] [HIDDEN]**

1. [8 points] Consider the 5-processor dissemination algorithms shown pictorially below:




You are given the matrix below called MINROUNDS. The <i,j> element in this array represents the **earliest** round number (rounds start at 1) (one of 0, 1, 2, 3, etc.) in which a processor finds that one of its

peers has arrived at the barrier. For example, we have shown when each processor knows it has arrived at a barrier by filling in the diagonal elements. Fill the other elements of the array.

| | P0 | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|---|
| P0 | 0 | | | | | |
| P1 | | 0 | | | | |
| P2 | | | 0 | | | |
| P3 | | | | 0 | | |
| P4 | | | | | 0 | |
| P5 | | | | | | 0 |

+0.4 per non zero entry

| | P0 | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|---|
| P0 | 0 | 3 | 3 | 2 | 2 | 1 |
| P1 | 1 | 0 | 3 | 3 | 2 | 2 |
| P2 | 2 | 1 | 0 | 3 | 3 | 2 |
| P3 | 2 | 2 | 1 | 0 | 3 | 3 |
| P4 | 3 | 2 | 2 | 1 | 0 | 3 |
| P5 | 3 | 3 | 2 | 2 | 1 | 0 |

## IX. Potpourri (13 points)

1. [2 points] [True/False with justification] Each position in the array of flags defined in Anderson's queue lock is statically allocated to a designated processor to ensure fairness.

False.

Designating a specific location for each processor will result in unfairness. The algorithm dynamically assigns a new lock request to the next available spot in the array to ensure fairness.

[2 points]

False. With explanation for why dynamic allocation will ensure fairness/static allocation eliminates fairness in a queue algorithm

[otherwise, 0 points]

2. [2 points] [True/False with justification] MCS mutual exclusion lock can be implemented on a multi-processor that supports only a globally atomic T&S primitive.

True.

Though MCS algorithm requires atomic fetch-and-store and atomic compare-and-swap primitives, they can be simulated using a globally atomic T&S instruction.

[2 points]

True, globally atomic T&S is an appropriate replacement( is able to simulate) a for fetch and store

[otherwise, 0 points]

3. [3 points] Explain why the MCS barrier algorithm will work even on an NCC NUMA machine.

- The CN data structure which is used by the parent to know when the children are done will be housed in the local NUMA piece of the parent. All children will reach across the network with their respective pointers to write to their designated memory location in this data structure. (+1)
- While NCC-NUMA does not provide cache coherence across the entire NUMA machine, the memory hierarchy within each node is coherent. Thus, the writes into CN by the children will be reflected into the cache of the processor on which the parent is executing. (+1)
- Similarly, the CP data structure which is used by the parent to wakeup a child that the barrier is complete will be housed in the NUMA piece where the child is executing. For the same reason as in (2) above the child will get notified. (+1)

[adopted the ideas presented in the above rubric]

[+3 for mentioning all three points, +2 for mentioning that children nodes reach out over the interconnect and writes by children will reflect in the parent processors cache (but not mentioning CP updates, +1 for only mentioning point 1, 0 otherwise]

4. [2 points] [True/False with justification] In Tornado parallel OS, the representation of clustered object – namely, replication, partitioning across the processors – is decided at the design time of a given subsystem and does not change.

False.

Tornado incrementally optimizes the clustered object representation by monitoring the execution pattern of applications. For example, the memory management service will decide how to break up an address space into regions based on the execution pattern.

[2 points for the above mentioned answer(even without the example), 1 point for False with some explanation, 0 otherwise]

5. [4 points] Explain why in a vanilla client/server RPC package there are 4 message copies between the client's invocation and the server procedure starting to execute.

1) Copy actual args from Client stack and create RPC packet by client stub

2) Kernel copy of RPC packet from Client domain into the kernel

3) Copy RPC packet from the Kernel into the Server domain

4) Disaggregate the RPC packet into the actual args and place them on the execution stack of the server by the server stub

[1 point for each of the copy steps]