# CS 6210 Spring 2024 Test 1

# (120 min Canvas Quiz)

# [Total Points: 102]

## Table of Contents

# OS Structure [21 points]

## SPIN [7 points]

1. (3 points) Do you agree with an assertion that SPIN has convincingly demonstrated the feasibility of developing an entire operating system using a high-level language that ensures strong type safety? No credit without justification.

   No, accessing some hardware resources (e.g., device registers in controllers) will require stepping outside the boundaries of Modula-3.

   +1 No only if there is justification
   +2 Strong reasoning as to why
   +1 Reasoning given, but it's incomplete or not clear

2. (4 points) In the context of OS structure, what is one similarity and one difference between the extensibility approaches 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.

   +2 for Any VALID similarity
   +2 for Any VALID difference; the above are just examples

## Exokernel [4 points]

1. [4 points]

Exokernel has a mechanism to repossess hardware resources allocated to a library OS. Let's say you are implementing a library OS on top of Exokernel. Specifically, you are implementing the virtual memory manager. How would you ensure the integrity of the processes running in your OS when Exokernel could at any time exercise its repossession mechanism?

Ans.

(+4) for either of the two options mentioned below

You have two options:

1) Exokernel gives you an opportunity to take the necessary actions (if any) for the hardware resources mentioned in the "repossession vector". In your case, for the page frames mentioned in the vector, you could "page out" the affected virtual pages of the processes.

2) You can seed Exokernel ahead of time with specific actions for each of the page frames mentioned in the repossession vector, so you don't have to do any additional work (such as I/O) at the time of revocation (except to take actions internal to your library OS such as fixing up page tables).

## Microkernel [10 points]

You are building an OS using a microkernel-based approach following the principles of the L3 microkernel. The processor architecture you are building this OS for has the following features:

- A byte-addressable 32-bit hardware address space.
- Paged virtual memory system (8KB pages) with a processor register called PTBR that points to the page table in memory to enable hardware address translation.
- A TLB which DOES NOT support Address space IDs and requires a flush on address space switching.
- A pair of hardware-enforced segment registers (lower and upper bound of virtual addresses) which limit the virtual address space that can be accessed by a process running on the processor.
- A virtually-indexed virtually-tagged processor cache (ignore potential coherence issues for the scope of this question).

You end up with the following subsystems that each need to be in a separate protection domain.

A: Requires 2^32 bytes virtual address space.

B: Requires 2^30 bytes virtual address space.

C: Requires 2^30 bytes virtual address space.

D: Requires 200*2^20 bytes virtual address space.

E: Requires 300*2^20 bytes virtual address space.

F: Requires 500*2^20 bytes virtual address space.

1. [6 points] Design the most efficient way of packing these domains into hardware address spaces and list out the potential lower and upper segment register values for your packing.

Soln:

[+2] Make sure each set take less than 2^32): Two hardware address spaces: one for A and (B, C, D, E, F).

Segment Register Values:

[+4] Segment register values can be:

For (A):

LB - 0, UB - (2^32 - 1)

For (B,C,D,E,F):

For B; LB – 0, UB - (2^30-1)

For C; LB – 2^30, UB - (2^31-1)

For D; LB – 2^31, UB - (2^31+200*2^20)-1

For E; LB – (2^31+200*2^20), UB - (2^31+500*2^20) - 1

For F; LB – (2^31+500*2^20), UB - (2^31+1000*2^20) - 1

2. [4 points] Assume that the following subsystems execute on the processor one after the other: A->B->C->D->E->F. What is the minimum number of TLB flushes and Cache flushes required (if any) in your design?
   What changes could you make to the processor architecture features to reduce the number of flushes required?

[+1] #TLB Flushes = 1

[+1] #Cache Flushes = 1

[+1] Address space ID supported TLB

[+1] Virtually Indexed Physically tagged cache.

# Virtualization [28 points]

## Paravirtualization [8 points]

[+8 points]

Imagine a Linux OS atop Xen (XenoLinux). A user launches an application process called "foo" on top of XenoLinux. Walk through the steps before "foo" actually starts running. The following hypercalls are available.

- create-PT(page-frame)
- switch-PT(page-frame)
- update-PT(page-frame, vpn, ppn)
- read(disk-block, page-frame)- moves the contents of the disk block into the named page frame

Ans:

1. Xenolinux picks up an empty page frame, (say PageFrame1) and executes the create-PT hypercall by passing the PageFrame1 to register it as the pagetable with the Xen hypervisor for the newly created process. [+2 points]

2. Switch-PT(PageFrame1) is executed next, to change PTBR to point to the pagetable of "foo". [+2 points]

3. XenoLinux picks up another empty page frame, (say PageFrame2) and reads a page from foo's binary on the disk into PageFrame2, using the read() hypercall. [+2 points]

4. It then executes Update-PT(PageFrame1, VPN0, PageFrame2) to update the PT mapping. [+2 points]

## Full Virtualization [8 points]

Given the following:

- Full virtualization
- Two VMs are running on the hypervisor
  - VM1 is running 10 processes (P1 to P10) started up in that order
  - VM2 is running 20 processes (P1 to P20) started up in that order
- Page table for each process requires 250 entries
- Page table for each of VM1 and VM2 require 1000 entries
- Each page table entry is 4 bytes
- Starting address for VM1's shadow page table (SPT) is 10000. Starting address for VM2's SPT is 50000.
- The processor has a Page Table Base Register (PTBR) which points to the memory address for the start of the page table for the currently running process

(a) (4 points) What is the size of the SPT for VM1 and VM2?

Ans.

- Per process PT size = 250*4 = 1000 bytes
- Per VM PT size = 1000*4 = 4000 bytes
- SPT for VM1 = 4000+10*1000 = 14000 bytes
- SPT for VM2 = 4000+20*1000 = 24000 bytes

(b) (2 points) At some point in time, the hypervisor is currently running P6 of VM1. What is the content of PTBR?

Ans.

- PTBR points to
  - Start of P6's PT in VM1's SPT
  - = Starting address of SPT + VM1 PT size + P1-P5 PT sizes
  - = 10000+4000+5*1000 = 19000

(c) (2 points) At another point in time, the hypervisor is running P11 in VM2. What is the content of PTBR?

Ans.

- PTBR points to
  - Start of P11's PT in VM2's SPT
  - = Starting address of SPT + VM2 PT size + P1-P10 PT sizes
  - = 50000+4000+10*1000 = 64000

## Memory Management [12 points]

1. In a datacenter, the hypervisor is using VM-oblivious page sharing. Virtual machine VM1 that is currently running experiences a page fault on VPN1. MPN1 is the machine page assigned to bring this missing page from the disk. The hypervisor notices that the page being brought in from the disk is a potential match to MPN2 that maps to VPN2 of another virtual machine VM2.

(a) (2 points) How does the hypervisor recognize that there is a potential match?

Ans.

The hash generated in parallel to the I/O for MPN1 (call it hash1) matches that of the entry for MPN2 (call it hash2) in the hashtable table data structure of the hypervisor.

(b) (4 points) List the steps taken by the hypervisor to share pages across VM1 and VM2 using this specific example.

Ans.

1. Hypervisor performs a full comparison (when cycles are available on the server) of MPN1 and MPN2.
2. Once the hypervisor determines that pages are identical, VPN1 is made to point to MPN2. MPN1 is freed.
3. The hash table entry for VPN2->MPN2 is updated to indicate that VPN1->MPN1 is a match by increasing the refcount for this entry.
4. Both the entries for VPN1 and VPN2 are marked copy-on-write

2. (6 points) A datacenter uses ballooning mechanism to alleviate memory pressure experienced by a VM. There is a covert channel that exists between the hypervisor and the balloon driver on each VM to get/put machine memory back and forth.

The policy used by the hypervisor is to tax 20% on the idle memory of VMs at a time. The hypervisor may exercise this tax in multiple rounds if need be, to meet the memory request of a given VM.

VM1, which is experiencing a memory pressure asks for 180 MB of additional memory.
Assume the following:

- The hypervisor has no free machine memory to give presently
- VM2 has an idle memory of 200 MB
- VM3 has an idle memory of 300 MB

List the steps taken by the hypervisor to meet VM1's request.

Ans.

a) Inflate balloon driver on VM2 to get 40 MB using the covert channel (20% of 200 MB)
b) Inflate balloon driver on VM3 to get 60 MB using the covert channel (20% of 300 MB)
c) Inflate balloon driver on VM2 to get 32 MB using the covert channel (20% of 160 MB idle memory remaining in VM2)
d) Inflate balloon driver on VM3 to get 48 MB using the covert channel (20% of 240 MB idle memory remaining in VM3)
e) Use the covert channel to give 180 MB machine memory to VM1
f) Deflate balloon driver on VM1 to release 180 MB

# Parallel systems [27 points]

## Shared Memory Machines [4 points]

1. You are designing the virtual memory manager for a shared memory multiprocessor. The workload you have to cater for include single-threaded processes as well as multi-threaded processes.

You have taken the following design decisions:

- You have a page fault handler on each processor.
- You have a page table for each process.

- The page fault handlers on the processors, share the shared page table of multi-threaded processes.

(a) (2 points) (Answer True/False with justification) There are concurrent page faults for single-threaded processes running on the multiprocessor. Your design will serialize the page fault handling for the concurrent requests.

Ans.

False.

Since each process has its own page table, the handlers will be able to process the requests simultaneously.

(b) (2 points) (Answer True/False with justification) There are concurrent page faults for a multi-threaded process running on the multiprocessor. Your design will serialize the page fault handling for the concurrent requests.

Ans.

True.

Even though the page fault handlers can run concurrently on all the processors, the single shared page table will force the handlers to obtain mutual exclusion on the shared page table to service the requests.

## M.E.Lock [6 points]

You are implementing a linked-list based lock algorithm on your multiprocessor (based on MCS lock) that provides fairness for threads competing for the same lock. (The below code may not be syntactically correct which should be ignored).

The basic data structure is as shown below:

```
q_node{
      boolean gotit; // initialized to FALSE
      next; // points to the next lock requestor
}
```

Every new lock one wishes to declare in the program will do the following:

```
L = new(q_node); // allocate a q_node
L->next = nil; // nobody is using the lock
```

A lock requestor for Lock L would do the following to execute a critical section:

```
My_lock = new(q_node);

My_lock->next = nil;

My_lock->gotit = FALSE;

LOCK(L, my_lock); // get lock

      // execute critical section

UNLOCK(L, my_lock); // release lock
```

You have implemented the lock/unlock algorithm as follows:

```
LOCK(L, new_request)
{
     if (L->next == nil)
     {
       L->next = new_request;  // points to the last requestor
        new_request->gotit = TRUE;
     }
     else {
       L->next->next = new_request; // add new requestor to the
     queue
       L->next = new_request; // new requestor is the last
     requestor
       While (new_request->gotit == FALSE); // spin waiting for
     lock
     }
}

UNLOCK(L, current_request)
{
  if (current_request->next == nil)
  {
    L->next = nil;  // no one is waiting
  } else {
    current_request->next->gotit = TRUE; // signal the next
requestor
  }
}
```

(a) (1 point) Does your LOCK implementation work correctly? No credit without a clear reasoning.

Ans.

No.

Reason:  Concurrent lock requests cannot be handled by this implementation

(b) (2 points) Write the code snippet to fix the LOCK algorithm.

Ans.

In the LOCK function, we need an atomic function that would fetch L->next and would link L->next = new_request

```
last_request = fetch_and_store(L, new_request);
If (last_request != nil)
{
  last_request->next = new_request;
  while(new_request->got_it == FALSE);
}
Else {
  new_request->got_it = TRUE
}
```

(c) (1 point) Does your UNLOCK implementation work correctly? No credit without a clear reasoning.

Ans.

No.

A new LOCK request is forming but the requestor has not yet executed fetch-and-store (L, new_request) in the LOCK algorithm leading to a race condition between the current request trying to release the lock (which would set L to nil thinking there is no other lock request waiting).

(d) (2 points) Write the code snippet to fix the UNLOCK algorithm.

```
// use a new atomic operation compare_and_swap
If (compare_and_swap(L->next, current_request, nil) == false)
      {
        while(current_request->next != nil); /spin for the next requestor completes the LOCK algorithm
        current_request->next->got_it = TRUE;
      }
```

## Barriers [8 points]

1. [2 points] Describe under what circumstance you may want to use a tournament barrier over an MCS barrier. Explain your answer.

Ans:

(+1) A cluster with no shared memory as opposed to a CC shared memory parallel machine

(+1) The tournament barrier would be advantageous if the processors are in a cluster (no shared memory) because the communication between two nodes can be represented as messages sent from one node to another. MCS barrier assumes shared memory machine for the construction of the MCS tree.

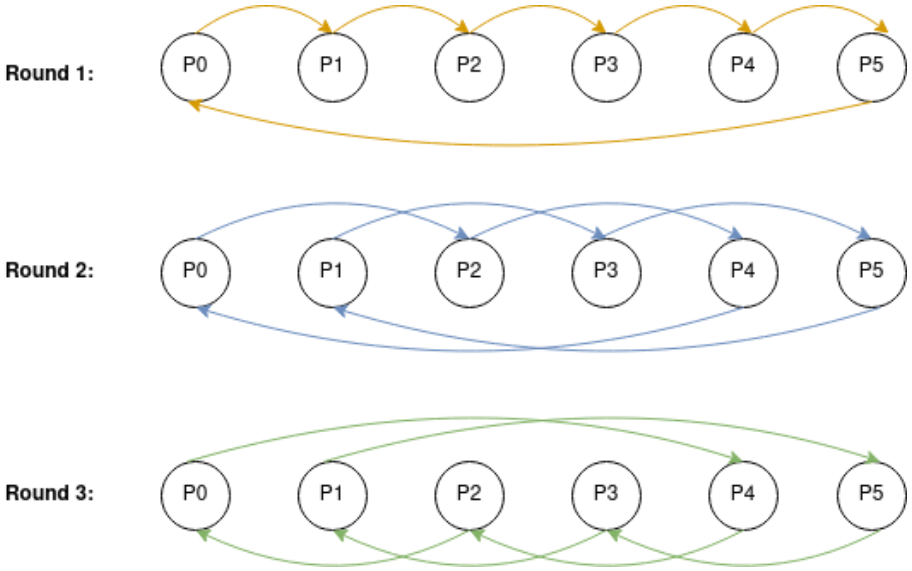
2. [2 points] Describe under what circumstance you may want to use an MCS barrier over a tournament barrier. Explain your answer.

Ans.

(+1) A CC shared memory machine as opposed to a cluster with no CC shared memory

(+1) The MCS Barrier could be ideal in a situation where we could exploit spatial locality by having multiple spin variables on the same cache line; the parent could spin on that location and each child could indicate there that they are done. The behaviour of the tournament barrier never leads to a situation where a node is talking to more than one other node, and therefore cannot benefit in the same way.

3. [4 points] Consider the dissemination algorithm on 6 processors.




(a) [2 points] After each round, write the list of processors that P0 knows have arrived at the barrier.

Ans.

a.

Round 1: P0, P5

Round 2: P0, P3, P4, P5

Round 3: P0, P1, P2, P3, P4, P5

(b) [1 point] How many additional processors should be participating in the barrier to warrant a fourth round?

Ans. 3 additional processors would need to be added for another round

(c) [1 point] How many less processors should be participating in the barrier to complete the barrier in two rounds?

Ans. 2 less processors to ensure barrier ends in 2 rounds

## LRPC [9 points]

1. Assuming a vanilla client/server RPC package on a shared memory machine,
a) [4 points] How many copies are needed before the server starts executing the server procedure? No credit without stating what the copies are and by whom

Ans.
(+1) client stack to RPC message by client stub
(+1) RPC message to kernel buffer by the kernel
(+1) kernel buffer to Server domain by the kernel
(+1) Server domain to server stack by Server stub

b) [2 points] What necessitates the kernel involvement?

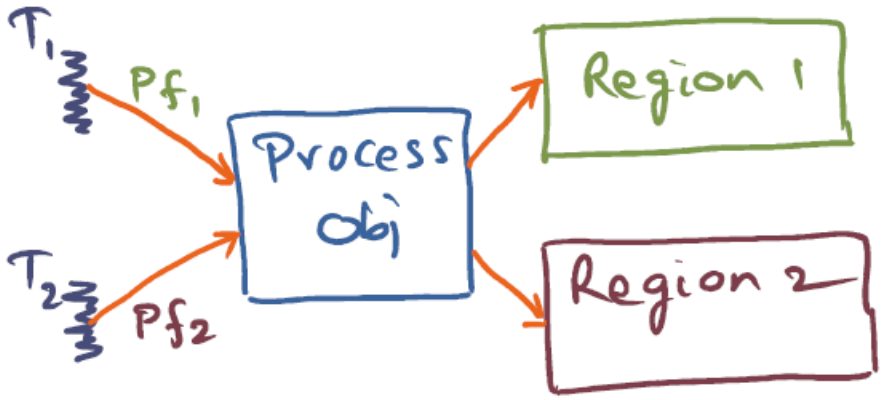
Ans.
(+2) Client and Server are in their own respective hardware address spaces and hence need mediation through the kernel.

c) [3 points] How does the "bind" mechanism in LRPC reduce the number of copies to two?

Ans.
(+1) A shared communication area (A-Stack) mapped into both the client and server address spaces is established by the kernel on the first call by the client.
(+1) The client stub simply copies the arguments into the A-Stack
(+1) The server stub simply copies the arguments from A-stack into the server's execution stack

## Parallel System OS [6 points]

1. (4 points) The figure below shows two threads of the same process running on different cores of the same processor simultaneously incurring page faults in Tornado OS. The page faults are to different regions of the address space as shown in the figure.




a. (2 points)
Why does Tornado suggest using reference count for the "process" object instead of mutual exclusion lock?

Ans.
If mutual exclusion lock is used the page fault service will be serialized through the process object despite the fact that the page faults change the mappings in different region objects.

b. (2 points)
What purpose is served by the reference count mechanism in the process object?

Ans.
Process object is the equivalent or a process context block in a vanilla OS like Linux. Tornado may do load balancing by moving the threads executing on a node to a different node. In this case, the process object at the original node will be garbage collected. The reference count mechanism ensures that the process object will not be garbage collected prematurely if page fault service is in progress.

2. (2 points) Imagine 16 threads of an application are executing on 16 cores of a CPU running Corey OS. Two threads T1 and T2 (out of the 16) are sharing an OS data structure (call it "fd"). How does the "shares" concept in Corey help to make this application more performant?

Ans.

The application can explicitly convey to the OS that fd is shared ONLY by T1 and T2. Helps Corey to optimize the contention and coherence maintenance for fd to the two cores that T1 and T2 are executing.

## Potpourri [20 points]

1.[2 points] (Answer True/False with justification) After being extended, an operating system operating on top of SPIN exists in a distinct hardware address space separate from SPIN itself.

False.

SPIN and service extensions are co-located in the same hardware space. SPIN expects the developer to implement each service as a Modula-3 object using the APIs provided by SPIN, namely, "create", "resolve" and "combine" to extend SPIN into the desired functionality.

+2 for False with correct justification
+1 for False with incorrect justification

2. [6 points] A library OS implements a paged virtual memory on top of Exokernel. The processor architecture has only a TLB for address translation (i.e., a TLB miss triggers a page fault). A process executing in the library OS incurs a page fault. After the page fault gets serviced by the library OS, a mapping for the virtual page number (VPN) and physical frame number (PFN) is then installed into the TLB.

a. [2 points] Page frame is a hardware resource. How does the library OS get to use any page frame for servicing this request?

Ans. Exokernel exposes an API for allocating physical hardware resources (such as a page frame) to the library OS.

b. [2 points] (Answer True/False with justification) The library OS directly installs the {VPN, PFN} mapping into the TLB.

Ans.

False

The library OS has to use the API provided by Exokernel for writing into the TLB, and Exokernel does the rest.

c. [2 points] (Answer True/False with justification) When the process is executing, the library OS presents its key for using the mapped page frame to Exokernel on every memory access.

Ans: False.

The library OS incurred a one-time cost of presenting the key when the mapping was first installed into the TLB.

During the execution of the process, the hardware uses the mapping already present in the TLB to do the address translation (without involving either Exokernel or the library OS).

+1 False with incorrect/no justification

+2 False with correct justification

3. [2 points] (Answer True/False with justification) In a para-virtualized environment, ANY and ALL program discontinuities are locally handled within the hypervisor.

Ans.

False.

The hypervisor fields the discontinuity but delivers it as a software interrupt to the Guest OS that needs to service the discontinuity.

4. [2 points] (Answer True/False with justification) With VM oblivious page sharing, checking for an exact match of a page being brought from disk to an existing machine page happens in the critical path of page fault handling.

Ans.

False.

Only the hash creation happens in parallel with the I/O as the page is being brought in from the disk. The exact match is done by a daemon process in the hypervisor (not in the critical path).

5. [2 points] (Answer True/False with justification). The ballooning technique for increasing/decreasing the VM's machine memory allocation is applicable only for para-virtualization.

Ans.

False.

Balloon driver is like any other 3rd party device driver and can be installed on the fly into an OS. Thus, the technique can be used in either full or para virtualization.

6. [2 points](Answer True/False with justification) In a multiprocessor system, using a scheduling policy that is based on respecting cache affinity is **always** more efficient than having a fixed processor scheduling policy.

Ans.

False, in case of heavy workloads, due to different threads running on a processor may totally wipe out the working set of an old thread that is supposed to be rescheduled on that processor. A fixed processor policy might be better here.

7. [2 points](Answer True/False with justification) Tornado OS allows the service provider to create multiple replicas of a single logical object. Hardware-enforced cache coherence in a shared memory multiprocessor will keep the replicas of an object consistent.

[+1] False.

[+1] The replicas of a single logical object separate entities located in different physical memory addresses, and hence Protected Procedure Calls are used to maintain consistency between them. Hardware Cache Coherence only maintains consistency between cache blocks backed by the same physical address.

8. (2 points) (Answer True/False with Justification)
A multi-threaded process running on top of Tornado is fully aware of how its address space is split into regions to increase the concurrency for handling page faults incurred by the threads.

Ans.
False.

The clustered object mechanism in Tornado is completely internal to the OS. An application has no knowledge of the region concept within Tornado for memory management.