This midterm test consists of 10 questions on 14 pages (including this one). When you receive the signal to start, please make sure that your copy of the examination is complete.

If you need more space for one of your solutions, use the last page of the test and indicate clearly the part of your work that should be marked. We have been careful to leave enough space for your answers.

In your written answers, be as specific as possible and explain your reasoning. Clear, concise answers will be given higher marks than vague, wordy answers. Please make your handwriting legible!

Please write in pen. Answers written in pencil will not be re-graded.

Marking Guide

# 1: _____/ 8
# 2: _____/ 8
# 3: _____/12
# 4: _____/ 6
# 5: _____/ 8
# 6: _____/12
# 7: _____/10
# 8: _____/10
# 9: _____/ 8
# 10: _____/ 8

TOTAL: _____/90
Question 1. Definitions [8 marks]

Define the following terms in the context of this course.

Part (a) [2 marks] Emergent property
Solution:
From the lecture slides, “surprises”, or an unexpected consequence of a change. More formally, a property of a collection of components (or modules) that make up a system that cannot be predicted by examining/analysing the components individually.

Part (b) [2 marks] Open system
Solution:
OS design in which all components (user processes, OS services, device drivers) execute in a single address space.

Part (c) [2 marks] Quiescent state
Solution:
A quiescent state for a thread T is a state in which T holds no references to shared data.

Part (d) [2 marks] False sharing
Solution:
Performance issue caused by cache coherence protocol when threads executing on different CPUs access (specifically, modify) different data items that are stored in the same cache line. [The cache coherence traffic is the same as if the threads were using a shared variable, even though they are using different items.]
Question 2. Systems Design [8 marks]

Part (a) [4 marks] In the first week, we discussed four general techniques for coping with complexity: modularity, abstraction, layering and hierarchy. Which of those four techniques is most evident in the design of the Unix timesharing system? Use at least one specific example to explain your answer.

Solution:
Of the four, abstraction is the most evident in Unix. For example, all I/O is handled via the file abstraction, including devices and other special files; programs in execution and all their associated resources are abstracted as a process; inter-process communication uses a simple pipe abstraction. [Any one example is enough]

Part (b) [4 marks] What was the guiding design philosophy of the exokernel? How did the authors of the exokernel paper apply the “end-to-end” argument to motivate this design philosophy?

Solution:
The exokernel design philosophy was to eliminate OS abstractions, separating protection of resources from abstraction and management. [i.e., securely multiplex bare hardware among user processes, imposing no OS abstractions on the end applications.] The end-to-end argument says functionality should not be implemented in lower layers unless it can be completely and correctly implemented there. Since some resource management decisions can be improved with the help of the application, or specialized for certain applications, they should not be implemented in the kernel, but rather left to the applications.
Question 3. OS Structure [12 marks]

Part (a) [4 marks] Mach and L4 provide the same four basic abstractions and/or mechanisms (although with different names). Identify them and explain why these must be implemented by a microkernel.

Solution:
1. Threads (unit of execution) to execute requires allocation of physical CPU resource, which is responsibility of microkernel (privileged). Actual scheduling policies can be implemented in user space but microkernel needs to provide support mechanisms and overall control.
2. Address spaces/Tasks (unit of protection) represents access to resources that are allocated exclusively to some process and requires privileged operations to create and modify the permissions, and prevent unauthorized access.
3. Mappings/Memory objects+devices microkernel provides privileged control over the allocation of physical resources, most importantly memory (which is needed by all running code) and devices.
4. IPC (ports+messages in Mach) since kernel is split into multiple servers, support must be provided for communication between them, and this must be in the microkernel for protection/correctness.

Part (b) [4 marks] Explain the difference between Type I and Type II Virtual Machine Monitors, and give one specific example of each.

Solution:
A Type I (aka “classic”) VMM runs on the bare hardware as the most privileged software on the machine; all other software runs in a less privileged mode on Virtual Machines created by the VMM. Examples are Xen, VMware Server, Hyper-V. A Type II (aka “hosted”) VMM is installed on top of an existing host OS and must use features of the host OS to access hardware. Examples include VMware Workstation, VirtualBox. [Reasonable arguments can be made for KVM as either Type I or Type II.]

Part (c) [4 marks] Explain briefly how OS extensions are supported by Mach and Linux, and identify the primary advantage that each approach has over the other.

Solution:
In Mach, extensions are provided by new user-level servers. In Linux, extensions are provided by kernel modules. Mach’s primary advantage is safety - extension code runs at user-level and is isolated from the microkernel and other servers. Linux’s primary advantage is performance - modules execute in the kernel address space and can directly invoke other kernel functions or read/write kernel data structures.
**Question 4.** Performance Evaluation and Benchmarking [6 marks]

**Part (a) [4 marks]** Suppose you need to measure the time for some function \( f() \) in an ordinary user-space process. You can use either (i) the hardware cycle counter (continuously incremented on every cycle, without regard to context switches or mode switches), (ii) an interval timer with 10ms resolution that records user time per process, or (iii) some combination of both. The system clock ticks at the same resolution as the interval timer. You expect the runtime of \( f() \) to be less than 5ms. Describe how you would measure the time of \( f() \) as accurately as possible. (Point form answers are fine)

**Solution:**
Since the system clock ticks at 10ms, we can expect timeslice to be at least 10ms, so it should be possible for \( f() \) to run in a single timeslice.

- Run \( f() \) once to warm the cache [1 point]
- Read initial interval timer, then read initial cycle counter, run \( f() \), and read final cycle counter and interval timer [1 point]
- If cycle counter has changed, possible context switch, discard measurement and repeat previous step [1 mark]
- Calculate total cycles = final cycle counter - initial cycle counter, optionally converting to time using CPU frequency. [1 mark; anything along the lines of computing the actual time based on the measurements is acceptable]

[Alternate answer involves using interval timer alone, warming cache [1 mark], running \( f() \) long enough to reduce the error due to timer resolution below some threshold [1 mark], dividing total time by the number of iterations [1 mark], and using k-best to discard runs that appear to have been interrupted because of higher times [1 mark].]

**Part (b) [2 marks]** After reading some old papers on file system benchmarking, you are inspired to benchmark the file system on your own (modern, high-end) desktop. You download the filesystem benchmark used in the research papers and run it with the default configuration. The benchmark reports that your system has a sequential read bandwidth of 1 GB/s. The spec sheet for your hard drive indicates a sequential read bandwidth of 200 MB/s however. **Explain** the most likely cause of the discrepancy.

**Solution:**
Most likely the benchmark was designed for older systems with smaller memory, and on the current desktop the entire test data fits in memory so you are measuring memory bandwidth (plus software overhead), not disk.

Total Pages = 14
Question 5. Optimization [8 marks]

A system runs a workload that spends nearly all its time on disk I/O consisting of roughly 80% random read and 20% sequential write operations (by time). You are asked to upgrade the system from a single hard disk to two disks in a RAID-0 configuration (data is striped across the two disks) with the following properties:

- Reads become slower, at 0.8x of the original (bottlenecked on the slowest drive seek).
- Writes become 2x faster, utilizing the sequential bandwidth of both disks.

Part (a) [4 marks] Explain why this upgrade is a bad idea. Show your work. (4 marks)

\[
\frac{80\%}{0.8} + \frac{20\%}{2} = 110\%, \text{ not worth the upgrade.}
\]

Part (b) [4 marks] Instead of the RAID-0 configuration, you propose adding an SSD to act as a read cache. This will achieve 5x faster read performance for the workload, but the time spent on cache management will add 10% overhead (of the original time). What overall speedup can you expect? Show your work.

\[
\frac{80\%}{5} + 20\% + 10\% = 46\% \text{ of original}
\]
Question 6. Signals and IPC [12 marks]

Part (a) [4 marks] Explain when and how a user-level signal handler is invoked, including the purpose of the trampoline code placed on the user stack.

Solution:
- Threads check for pending signals once per kernel entry, usually just before leaving kernel to return to user-space. If a signal has been marked in the thread’s signal list, the kernel checks to see if the thread has a user-level signal handler registered for that signal.
- If so, signal state (also known as signal context) is saved on the user stack along with trampoline code. Registers (e.g. pc register, stack register) are manipulated to begin execution in the trampoline upon return to user-space.
- The kernel does a normal return to user-space, and the trampoline code calls the registered signal handler function. When it is done, the signal handler function returns to the trampoline, which executes a special “signal return” system call.
- The OS handles this system call using the saved signal context to restore stack and registers and return to the original point in user-space.

Part (b) [4 marks] A monitoring and control process needs to take some action a1 (e.g. read a temperature sensor and adjust fan speed) every 15ms of real time. It must perform another action a2 every 1.5s. In between these actions, the program has other compute work to do. The OS provides interval timers with 1ms resolution. The process sets up an interval timer with an initial value of 15ms. The OS decrements the timer in real time, posts SIGALRM to the process when it reaches 0, and resets the timer to the initial value (repeatedly). Thus, the OS will post SIGALRM to the process every 15ms. The process uses a signal handler to catch SIGALRM and (i) performs a1, (ii) increments a counter, and (iii) when the counter reaches 100, performs a2 and resets the counter to 0. SIGALRM is an ordinary POSIX signal. Explain what can go wrong in a heavily loaded system.

Solution:
Ordinary POSIX signals are not queued, so multiple occurrences of the same signal before the previous one is handled cannot be detected (signals can be lost). On a heavily loaded system, there may be more than 15ms of delay between posting SIGALRM when the timer expires and delivery/handling of that signal by our control process. The process will still perform a1 as often as possible (it may be more than 15ms between checks and adjustments, but it will happen as soon as the process gets scheduled) – a1 is not affected by the loss of some SIGALRMs, only by the delay in acting upon the signal. But a2 can be delayed even further, as the internal counter only ticks once per signal delivery (for instance, if 2 signals are lost for every 1 that is delivered, it could take 4.5s for 100 signals to be delivered). The problem is not that the process has no opportunity to perform a2 (it is getting scheduled often enough) but instead that it has lost track of the real passage of time.

Part (c) [4 marks] Describe two features of the mbuf data structure used for socket and network communication that are designed to support efficient message handling.

Solution:
- mbufs in a packet are linked together and packets in a message are linked together with pointers in the first mbuf of the packet. This allows entire messages to be moved between queues by moving just the first mbuf.
- mbufs allow a pointer to external data so that large data doesn’t need to be broken down into multiple mbufs (particularly good for local IPC).
- mbuf header includes pointer to data and data length, so additional headers for other protocol layers can be easily added and removed.
Question 7. Event Notification [10 marks]

Part (a) [2 marks] Why is an event notification mechanism like select() or poll() needed?
Solution:
Traditionally, read() and write() are blocking operations, but a process often has other work it can do if a particular input or output channel is not ready. For example, a server handling a large number of client connections should not block waiting to receive a request from one client while other clients have submitted requests and are waiting for the server.

Part (b) [4 marks] Briefly explain how kqueue provides efficient and scalable event notification, including the use of the knote data structure.

Solution:
The kqueue solution separates the expression of interest in an event from the notification of that event. Processes can express interest in a large number of events once, but notification only reports on events that have actually happened, which makes it more scalable and efficient than traditional poll() or select().
The knote data structure represents events of interest. On registration of interest in an event, a knote is created and attached to the event source (e.g., open file object). The knote includes a filter function that executes whenever there is some activity at the event source, and decides if the event should be reported to the application. If so, the knote structure is linked to the active list for the kqueue. When a process checks for events of interest, the OS only needs to scan the active list and report all knotes that are linked there.

Part (c) [4 marks] The data in the kqueue paper showed that it was roughly 1.5X more costly to delete an event than to disable an event on 400 descriptors, and that deletion was less scalable as the number of descriptors increased. Suppose you have previously registered an event on some (larger than 400) set of descriptors, and you are no longer interested in that event. Should you delete the event or just disable it? What extra information, if any, would you need before making this decision?

Solution:
Recall that disabling an event only prevents the kevent from being returned to the user - the filter will still be executed and so disabling rather than deleting will have an ongoing cost whenever the event occurs. Also, other operations (adding new kevents, enabling kevents) grow more costly with increasing numbers of descriptors. In the immediate term, you can save time by just disabling the event, but deletion may be cheaper in the long run if you expect to continue to add kevents on new descriptors, or if you expect the descriptor will continue to have a lot of activity (but you don’t want to be notified).
Question 8. Advanced Locking [10 marks]

Part (a) [2 marks] Why is lock acquisition expensive on modern multiprocessors, even if there is no contention for the lock?

Solution:

- Requires some atomic memory operation, so it will happen at memory speed and not at CPU or cache speed. [As CPUs have become faster relative to memory access time, the cost of these atomic memory operations has increased.]
- Requires explicit fences or barriers to prevent memory operations from being re-ordered across the lock, which stalls processor pipelines and prevents many optimizations. [Processor pipelines have become deeper and rely more on out-of-order execution to keep them filled, so the barriers become more costly.]

Part (b) [2 marks] Suppose we have a process with many more threads than there are CPUs on the system where it executes. What problem can occur with ticket locks that would not occur with ordinary spinlocks?

Solution:

Threads can be executing on all CPUs waiting for a lock that is available, because it is not their turn to get the lock yet, and the thread that should get the lock next has not been scheduled to run.

This question continues on the following page.
Part (c) [6 marks] Code for the MCS lock, as shown in lecture, is reproduced below. When a thread attempts to acquire a lock that is already held, it first sets `locked` of its own node to `TRUE` (line 9), and then redirects `next` of its predecessor to its own node (line 10). Give a scenario to show that the MCS queue lock could deadlock if this order were reversed.

```c
struct qnode {
    int locked;
    struct qnode *next;
}

void acquire(struct qnode **tail, struct qnode *my_node) {
    my_node->next = NULL;
    struct qnode *pred = fetch_and_store(tail, my_node);
    if (pred != NULL) { // queue not empty
        my_node->locked = TRUE;
        pred->next = my_node;
        while (my_node->locked); //spin
    }
}

void release(struct qnode **tail, struct qnode *my_node) {
    if (my_node->next == NULL) {
        if (compare_and_swap(tail, my_node, NULL))
            return;
        while (my_node->next == NULL); //spin
    }
    my_node->next->locked = FALSE; // release next waiter
}
```

Solution:

- **T1 holds the lock** (lock points to T1's qnode) and is about to call `release()`.
- **T2 calls acquire and gets T1's qnode as pred**, then sets `pred->next = my_node`.
- **T1 executes release**, sees that `my_node->next` is not NULL (its T2's node) and sets `my_node->next->locked = FALSE`.
- **T2 executes**, setting `my_node->locked = TRUE`, and then spins forever since predecessor has already released the lock.

Since MCS locks are FIFO, and T2 is supposed to enter `next` (lock points to T2 already), no other thread will be able to acquire the lock either. Deadlock results.
Question 9. Transactional Memory [8 marks]

Suppose we have the binary search tree shown below, which has been implemented with transactional memory as shown in the code boxes. Assume the TM system tracks conflicts at a word granularity.

![Binary Search Tree Diagram]

(Trees and values not shown.)

```c
typedef struct treenode_s {
    int key;
    int value;
    struct treenode_s *left;
    struct treenode_s *right;
} tnode_t;

int tree_search ( tnode_t *root, int k, int *val ) {
    int found = 0;
    tnode_t *n;
    atomic {
        n = root;
        while (n != NULL && k != n->key) {
            if (k < n->key) 
                n = n->left;
            else
                n = n->right;
        }
        if (n && k == n->key) {
            *val = n->value;
            found = 1;
        }
    }
    return found;
}

void tree_insert ( tnode_t **root, tnode_t *newnode ) {
    tnode_t *cur;
    tnode_t *parent;
    atomic {
        cur = *root;
        while (cur != NULL) {
            parent = cur;
            if (newnode->key < cur->key) 
                cur = cur->left;
            else
                cur = cur->right;
        }
        if (parent == NULL)
            *root = newnode;
        else {
            if (parent->key < newnode->key)
                parent->left = newnode;
            else
                parent->right = newnode;
        }
    }
}
```

\(T_1\) is executing \texttt{tree_search(root, 20, &result)}. Concurrently, \(T_2\) is executing \texttt{tree_insert(&root, mynewnode)} for a new node with key 16. \(T_2\) reaches the commit point first and successfully commits its transaction.

**Part (a) [4 marks]** What items are in \(T_1\)’s read set and write set when it attempts to commit the transaction?

**Part (b) [4 marks]** Will \(T_1\)’s transaction commit successfully? Explain your answer making reference to \(T_1\)’s read set and write set, and \(T_2\)’s transaction.

Write your answers on the following page. (Part (a) and (b) are repeated for your convenience.)
Question 9. Transactional Memory (continued)

Part (a) [4 marks] What items are in T₁’s read set and write set when it attempts to commit the transaction?

Solution:
Nodes are identified by their key number in this answer, for instance, the node with key 15 will be referred to as Node 15.

Read set: { (Node 15)→key, (Node 15)→right, (Node 18)→key, (Node 18)→right, (Node 23)→key, (Node 23)→left, (Node 20)→key, (Node 20)→value }

Write set: {} (empty, T₁ does not write any shared data)

(Notes:
1. We will accept the read set either with, or without, 'root'. As written in the question, 'root' is a private copy (passed to the function) of the address of the root node in the tree. As such, it is not a shared variable and does not belong in the root set. As written, however, this code is incorrect (if the global shared root pointer changed after root was passed to tree_search, the searching thread would not see the change) - the function should have been passed the address of the root pointer, and the value of that address should be read as part of the transaction that searches the tree, meaning that '*root' should be in the read set.
2. It is not specified if the “result” parameter passed into tree_search() by T₁ is shared or local, so putting either “*val” or “result” into the write set is acceptable.)

Part (b) [4 marks] Will T₁’s transaction commit successfully? Explain your answer making reference to T₁’s read set and write set, and T₂’s transaction.

Solution:
Yes. The transaction would only fail to commit if the read set validation failed, which means that there is a conflict between the T₁’s read set and T₂’s write set. T₂ inserted key 16, which would have involved a write only to (Node 18)→left. This location is not in T₁’s read set, so there is no conflict.
Question 10.  RCU and Scalability [8 marks]

Part (a)  [4 marks] Why is memory reclamation challenging for RCU-protected data structures?

Solution:
The memory allocated to the object can only be reclaimed when there are no more references to the object. But, because readers do not perform any synchronization operations in RCU (rcu_read_lock / rcu_read_unlock may be no-ops in some environments), it is challenging to figure out when there can be no more references to an object that has been removed from a shared data structure. The requirement to keep the read-side critical sections lightweight constrains the possible solutions - for instance, reference counting on the objects introduces shared data updates (the reference counter on each object) that destroy the scalability benefits of RCU.

Part (b)  [2 marks] RCU can be applied to synchronize access to data structures like linked lists. Should all linked lists in a multiprocessor OS kernel be protected by RCU? Briefly justify your answer.

Solution:
No. RCU trades higher read performance for reduced write performance, so it is only suitable for read-mostly data structures. Many linked lists in an OS kernel will are not read-mostly. [Other reasons: Some data structures may need stronger guarantees than RCU provides.]

Part (c)  [2 marks] In lecture, we looked at a shared counter that was implemented as an array of per-CPU counters with each element padded to occupy a full cache line. Should all counters in a multiprocessor OS use this implementation? Briefly justify your answer.

Solution:
No. The padded per-core counter trades faster updates for slower reads, and is thus most suited for counters that are frequently written but rarely read; not every counter used in the OS has this property. [Other reasons: the padded per-core counter also trades space for time. OS may have a lot of reference-counted objects and the machine may have a lot of cores - for example, if a memory page frame has a reference counter, a system with 64 cores and a 64 byte cache line size would need 1 page of memory just to store the padded per-core counter!]