Quiz2 Operating Systems	
Name: Zubair Abid	Roll No: 20171076

1. Briefly explain about the notion of para-virtualized architecture and its advantages.[5]

Ans : Paravirtualized architecture is a concept of virtual machine architecture built using Paravirtualization, a technique in which the hypervisor exposes a modified version of physical hardware interface (such as replacing features of an architecture that are difficult to virtualize with custom paravirtual operations) to the guest machine.

The idea behind paravirtualization is to replace complex computation emulating hardware and boost virtualization performance with a setup that replaces the non-virtualizable OS instructions with hypercalls to certain aspects of the physical hardware. The OS actually recognizes the presence of a hypervisor (the software/hardware that creates and runs virtual machines) and communicate directly with it to share the activity that would otherwise be complex and time-consuming for the hypervisor's VM manager to handle. In order for paravirtualization to work, the guest VM OS must be tailored to run on the given hypervisor

Advantages:

a. *With respect to Performance*

Performance is the most well-known advantage that paravirtualization has.

With paravirtualized device drivers in a fully virtualized OS this advantage is actually getting smaller over time. However, compared to traditional full virtualization (where the virtualization software emulates a complete computer and a completely unmodified guest operating system is run), for reasons elucidated above paravirtualization has very significant performance advantages.

b. *With respect to Hardware Support*

Paravirtualization imposes its changes in a very few aspects of the kernel, and allows many other aspects to continue working unmodified. In particular it uses the native device drivers of its "host" Operating System, thereby offering more flexibility in terms of hardware support.

2. Explain the data transfer mechanism in Xen [5]

Ans : The Xen Virtual Machine Monitor (VMM) presents clean and simple device abstractions allowing efficient I/O as well as good I/O related communication between the guest OS and the VMM. **Resource management** and **Event Notification** are two major factors shaping the design of the mechanism.

For each device used by each guest, there is a circular buffer shared by the guest and the VMM via shared memory. Read and write data are placed in this buffer. This ring buffer is a circular queue of “descriptors” allocated by a domain, but accessible only from within Xen. These descriptors do not directly contain I/O data, but instead, I/O data buffers are allocated out-of-band by the guest-OS and indirectly referenced by I/O descriptors. Access to each ring is based around two pairs of producer-consumer pointers: domains place requests on a ring, advancing a request producer pointer, and Xen removes these requests for handling, advancing an associated request consumer pointer. Responses are placed back on the ring similarly, save with Xen as the producer and the guest OS as the consumer. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows Xen to unambiguously reorder I/O operations due to scheduling or priority considerations.

3. Regarding the development of Multikernel OS, explain the following observation: Messages cost less than shared memory.[5]

Ans : To understand the observation, consider an experiment that compares the costs of updating a data structure using shared memory with the costs using message passing.

In the **shared memory** case, threads pinned to each core directly update the same small set of memory locations without locking and the cache-coherence mechanism migrates data between caches as necessary. The costs grow approximately linearly with number of threads and number of modified cache lines. It has been observed that a single core can perform the update operation in under 30 cycles, whereas 16 cores modifying the same data takes almost 12,000 extra cycles to perform each update. All of these extra cycles are spent with the core stalled on cache misses and therefore unable to do useful work while waiting for an update.

In the case of **message passing**, client threads issue a lightweight remote procedure call to a single server process that performs the update on their behalf. As expected, the cost varies little with the number of modified cache lines since they remain in the server's local cache. Because each request is likely to experience some queuing delay at the serve proportional to the number of clients, the elapsed time per operation grows linearly with the number of client threads. Despite this, for updates of four or more cache lines, the Remote Procedure Call (RPC) latency is lower than shared memory access. Furthermore, with an asynchronous or pipelined RPC implementation, the client processors can avoid stalling on cache misses and are free to perform other operations.

This above experiment shows scalability issues for cache coherent shared memory on even a small number of cores, which is why **message passing is preferred**. Current OSes have point-solutions for this problem, which work on specific platforms or software systems, but it is believed that the inherent lack of scalability of the shared memory model, combined with the rate of innovation seen in hardware, will create increasingly intractable software engineering problems for OS kernels.

4. Describe the process of inter-core communication in Barrelfish. [5]

Ans : All inter-core communication occurs with **messages** in a multikernel, . The only inter-core communication mechanism available on the current hardware platforms is cache-coherent memory. Barrelfish presently therefore uses a variant of user-level remote procedure calls (URPC) between cores: a region of shared memory is used as a channel to transfer cache-line-sized messages point-to-point between single writer and reader cores.

Inter-core messaging performance is critical for a multikernel, and the Barrelfish implementation is carefully tailored to the cache coherence protocol to minimize the number of interconnect messages used to send a message. As an example, on the fast path for a HyperTransport-based system, the sender writes the message sequentially into the cache line, while the receiver polls on the last word of the line, thus ensuring that in the (unlikely) case that it polls the line during the sender's write, it does not see a partial message. In the common case, this causes two round trips across the interconnect: one when the sender starts writing to invalidate the line in the receiver's cache, and one for the receiver to fetch the line from the sender's cache. The technique also performs well between cores with a shared cache.

As an optimization, pipelined URPC message throughput can be improved at the expense of single-message latency through the use of cache prefetching instructions. This can be selected at channel setup time for workloads likely to benefit from it. Receiving URPC messages is done by polling memory. Polling is cheap because the line is in the cache until invalidated; in addition, keeping the endpoints in an array can allow a hardware stride prefetcher to further improve performance.

5. List the characteristics of network sensors and the corresponding features required in Tiny OS [5]

Ans : TinyOS is primarily directed at Wireless Sensor Networks which refers to a group of sensors dedicated to monitoring and recording the physical conditions of the environment and organizing the collected data at a central location. Their characteristics are:
• The sensor networks have low power, limited memory and energy constraints due to their small size
• Limited Physical Parallelism and Controller Hierarchy: The number of independent controllers, the capabilities of the controllers, and the sophistication of the processor-memory-switch level interconnect are much lower than in conventional systems. Typically, the sensor provides a primitive interface directly to a single-chip micro-controller.
• These devices deal with heavy flow information from place to place with a modest amount of processing on-the-fly, rather than to accept a command, stop, think, and respond. Hence they are concurrency-intensive devices.
• Diversity in Design and Usage: Networked sensor devices will tend to be application specific, rather than general purpose, and carry only the available hardware support actually needed for the application. As there is a wide range of potential applications, the variation in physical devices is likely to be large.
• Robust Operation: These devices will be numerous, largely unattended, and expected to form an application which will be operational a large percentage of the time. Hence, enhancing the reliability of individual devices is essential. Additionally, we can increase the reliability of the application by tolerating individual device failures.

Due to the following characteristics of WSNs, the following requirements were defined for TinyOS:
• The OS must require very few resources, and must make efficient use of processor and memory while enabling low power communication.
• Space and power constraints and limited physical configurability on-chip are likely to drive the need for the operating system to support concurrency-intensive management of flows through the embedded microprocessor.
• Adapt to hardware evolution and support a diverse set of platforms & applications
• Robust design and modularity, since division into modules makes it easier for modification of code to suit a particular application, and also should facilitate the development of reliable distributed applications.

6. Briefly explain how the following aspects are designed in TinyOS: Memory allocation, command execution (blocking/non-blocking), mutual exclusion handling. [5]

Ans : **Memory Allocation** :

TinyOS has only static memory allocation. The memory requirements are determined at compile time itself, which in turn increases runtime efficiency, since it it prevents the overhead associated with dynamic allocation. The 'fixed size frames' of memory, due to static allocation, allows us to know the memory requirements of the system's component.

Command Execution :

TinyOS commands are fully non-blocking: Typically, a command will deposit request parameters into its frame (memory space) and conditionally post a task for later execution. It may also invoke lower commands, but it must not wait for long or indeterminate latency actions to take place. A command must provide feedback to its caller by returning status indicating whether it was successful or not. The lowest level components have event handlers connected directly to hardware interrupts, which may be external interrupts, timer events, or counter events. An event handler can deposit information into its frame, post tasks, signal higher level events or call lower level commands. A hardware event triggers a chain of processing events that goes upward, which in turn returns commands. In order to avoid cycles in the command/ event chain, commands cannot signal events. Both commands and events are intended to perform a small, fixed amount of work, which occurs within the context of their component's state.

Mutual Exclusion Handling :

Tasks are atomic processes that perform with respect to other tasks and run to completion, though they can be preempted by events. Tasks can call lower level commands, signal higher level events, and schedule other tasks within a component. The run-to-completion semantics of tasks make it possible to allocate a single stack that is assigned to the currently executing task. This is essential in memory constrained systems. Tasks allow us to simulate concurrency within each component, since they execute asynchronously with respect to events. However, tasks must never block or spin wait

or they will prevent progress in other components. While events and commands approximate instantaneous state transitions, task bundles provide a way to incorporate arbitrary computation into the event driven model. The task scheduler is currently a simple FIFO scheduler, utilizing a bounded size scheduling data structure.

7. Briefly explain the basic idea of "Cells: a virtual mobile smart phone architecture". [5]

Ans : Mobile phone users often carry multiple mobile phones to accommodate work, personal, and geographic mobility needs. Cells is a **virtualization architecture** targeted at running **multiple virtual smartphones** on the same physical mobile phone in an isolated, secure manner.

It puts one virtual phone in the foreground at a time. This model enables a new device namespace mechanism (which is completely different from the PID/UID namespace mechanism in Linux) and novel device proxies that integrate with lightweight operating system virtualization to multiplex phone hardware across multiple virtual phones while providing native hardware device performance (The presence of multiple virtual phones should not negatively affect the user experience). Cells does not require running multiple OS instances: by using lightweight OS virtualization it provides virtual namespaces that can run multiple VPs on a single OS instance. Cells isolates VPs from one another, and ensures that buggy or malicious applications running in one VP cannot adversely impact other VPs. Cells provides a novel file system layout to maximize sharing of common read-only code and data across VPs, minimizing memory consumption and again, enabling additional VPs to be instantiated without much overhead.

8. Summarize the power management strategies employed in Cells. [5]

Ans : **Principles -**

To provide Cells users the same power management experience as non-virtualized phones, two simple virtualization principles have been applied:

- Background VPs should not be able to put the device into a low power mode
- Background VPs should not prevent the foreground VP from putting the device into a low power mode.

Android -

These principles are applied to Android's custom power management, which is based on the premise that a mobile phone's preferred state should be suspended.

Android introduces three interfaces which attempt to extend the battery life of mobile devices through extremely aggressive power management: **early suspend**, **fbearlysuspend**, and **wake locks**, also known as suspend blockers.

Early Suspend

The early suspend subsystem is an ordered callback interface allowing drivers to receive notifications just before a device is suspended and after it resumes. Cells virtualizes this subsystem by disallowing background VPs from initiating suspend operations.

Frame buffer early suspend (fbearlysuspend) :

The fbearlysuspend driver exports display device suspend and resume state into user space, allowing user space to block all processes using the display while the display is off, and render the screen after the display is powered on. Power is saved since the overall device workload is lower and devices such as the GPU may be powered down or made quiescent when not necessary. Android implements this functionality with two sysfs files, *wait_for_fb_sleep* and *wait_for_fb_wake*. When a user process opens and reads from one of these files, the read blocks until the frame buffer device is either asleep or awake, respectively. Cells virtualizes fbearlysuspend by making it namespace aware, leveraging the kernel-level device namespace and foreground-background usage model. In the foreground VP, reads function exactly as it would in a non-virtualized system. Reads from a background VP always report the device as sleeping. When the foreground VP switches, all processes in all VPs blocked on either of the two files are unblocked, and the return values from the read calls are based on the new state of the VP in which the process is running. Hence, this forces background VPs to pause drawing or

rendering which reduces overall system load by reducing the number of processes using hardware drawing resources.

Wake Locks :

Wake locks are a special kind of OS kernel reference counter with two states: active and inactive. When a wake lock is “locked”, its state is changed to active; when “unlocked”, its state is changed to inactive. A wake lock can be locked multiple times, but only requires a single unlock to put it into the inactive state. The Android system will not enter suspend, or low power mode, until all wake locks are inactive. When all locks are inactive, a suspend timer is started. If it completes without an intervening lock then the device is powered down.

Cells virtualizes Android wake locks by allowing multiple device namespaces to independently lock and unlock the same wake lock. Power management operations are initiated based on the state of the set of locks associated with the foreground VP. The solution comprises the following set of rules:

- When a wake lock is locked, a namespace “token” is associated with the lock indicating the context in which the lock was taken. A wake lock token may contain references to multiple namespaces if the lock was taken from those namespaces.
- When a wake lock is unlocked from user context, remove the associated namespace token.
- When a wake lock is unlocked from interrupt context or the root namespace, remove all lock tokens.
- After a user context lock or unlock, adjust any suspend timeout value based only on locks acquired in the current device namespace
- After a root namespace lock or unlock, adjust the suspend timeout based on the foreground VP's device namespace.
- When the foreground VP changes, reset the suspend timeout based on locks acquired in the newly active namespace. This requires per-namespace bookkeeping of suspend timeout values.