Paper title: Exokernel: An Operating System Architecture for Application-Level Resource Management
Paper authors: Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr.
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
This paper looks at enhancing the flexibility and performance of applications by replacing the way traditional operating systems abstract the hardware resources. This high-level abstraction is immutable which hampers the ability of the applications to optimize, inhibits them from modifying the implementation of existing abstractions and limits their flexibility. Discouraging domain-specific implementation of abstractions results in applications paying overhead costs rooting from the trade-offs that are decided by the OS without considering the requirements of the application. This also results in a limited view of the system information to the applications thereby making it difficult for them to apply any customized resource management abstractions.

Previously, several endeavors have been made into developing flexible kernels. ***Hydra*** separated kernel policy and mechanism - a principle extended by exokernel which eliminates mechanism altogether. ***VM/370*** offers virtualization of the base-machine - an expensive and complicated affair that still doesn't address the lack of control for the applications. ***Microkernels*** provided some flexibility over monolithic kernels by distributing the functionality of the kernel over several sub-processes known as servers, running in the user space. However, these microkernels restricted applications from modifying the high-level abstractions which again raises the same kind of limitations as discussed above. ***Cache Kernel*** provides a low-level kernel supporting concurrent application-level OS, however it focuses on reliability and thus limits the extent of concurrency. Exokernel circumvents this by safely exporting the hardware resources and providing abundant freedom to untrusted applications.

What are the key contributions of the paper?
The paper designs an exokernel architecture which pushes the OS interface very close to the hardware and transfers all hardware resources to library operating systems. For this, the exokernel aims to isolate protection from management using 3 mechanisms - secure bindings, visible resource revocations and abort protocol.

Secure binding aims to protect individual resources by enforcing authorization at bind time and then implementing access checks at access time. This mechanism is implemented using - hardware mechanisms, software caching and downloading application code. ***Visible resource revocation*** allows the involvement of the applications and notifies the application when the resources are drained. This allows the library OS to save state information or relocate its physical names, thus enhancing performance. ***Abort Protocol*** is a second stage of the revocation protocol to forcefully break the secure bindings and free up resources, which is recorded in a *repossession vector*. This information is passed to the library OS so that it can update the mappings of the lost resource. Exokernel can preserve the state information of the revoked resource by writing it to a memory resource - one which can be pre-specified by the library OS itself.

The paper offers two software implementations - ***Aegis***, an exokernel and ***ExOS***, a library OS. These prototypes demonstrate the four hypotheses - exokernels are very efficient, secure low-level multiplexing of hardware resources can be implemented efficiently, traditional OS abstractions can be implemented efficiently at the application-level & applications can create special-purpose implementations of these abstractions.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
Aegis showcases good performance by monitoring ownership, maintaining a small kernel, caching secure bindings and downloading packet filters along with dynamic code generation for efficient secure binding to the network. It maintains the simplicity of basic primitives, which beats the general primitives of Ultrix in performance, along with superior implementation of exceptions dispatch and control transfer primitives, efficient multiplexing of the processor, memory and the network.

ExOS is able to efficiently implement interprocess communication, virtual memory and remote communication at the application level. It also creates special purpose implementations of these abstractions, which offer improvements in functionality and performance.

Write down one question you may want to ask the authors.
While the concept of exokernels is seemingly better than any of the kernels at that time, what could be some limitations preventing its mass adoption that the authors could have foreseen? This is intriguing to me, because unlike most research papers, this one does not discuss any scenarios where exokernels might be at a loss.

Paper title: The Multikernel: A New OS Architecture for Scalable Multicore Systems
Paper authors: A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, A. Singhania
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
The prime focus of the paper is to address the scalability issues of the operating system with advances in the hardware domain. OS optimizations are not portable across hardware and cannot easily adapt to developments in hardware. Further, the diversity in hardware is not only across machines - cores within a machine tend to be diverse. In this case, cores cannot share a single kernel instance since the tradeoffs in performance are different or the cores are inherently different in terms of their ISA. It is also evident from an experiment that cores using shared memory for updating a data structure spend extra cycles in the order of magnitude 3 waiting for an update. In contrast, message passing using asynchronous or pipelined RPC implementation avoids stalling the processors and frees them to perform other tasks. With increase in the number of cores, the traditional cache-coherence protocols also become excessively expensive and need to be replaced.

Multikernel architecture is based on several previous works in the domain. **Auspex** and **IBM System/360** had operating systems resembling distributed systems; multikernel improves the parallelism and deals with increased diversity of underlying machines. **Tornado and K42** presented clustered objects which used partitioning and replication, although they were based on shared data itself. **Microkernel** also employs message-passing, but maintains a shared memory kernel. Lastly, several works in the domain of **distributed operating systems** come close to the concept of multikernel. This paper showcases how multikernel leverages the best parts of all these systems to build a distributed system over a collection of cores.

What are the key contributions of the paper?
Based on the shortcomings of the traditional operating systems, this paper proposes a novel operating system that treats the machine as a network of cores that communicate through message passing and refrain from sharing memory. This idea is based on three principles - explicit communication among cores, hardware-independent design and using state replication over state sharing.

Firstly, except the messaging channels, no shared memory exists between the cores. Using explicit communication over implicit allows information about the shared state to be revealed. Further, this also allows the operating system to leverage common networking optimizations like pipelining or batching. Via message passing, operations can separate requests from responses and thus be productive while waiting for a response. Lastly, as they communicate through well-defined interfaces, there is scope for refining and tolerance to fault.

Secondly, decoupling OS from the hardware has several advantages. Adapting OS to new hardware becomes less cumbersome, while allowing communication algorithms to run independently of the underlying hardware implementation.

Lastly, state replication bolsters OS scalability by reducing load on the system interconnect, dispute for memory and the hassle of synchronizing. The paper also suggests an optimization based on limited sharing between a set of associated cores or threads.

Based on these design principles, the authors present **Barrelfish**, a multikernel OS implementation that proves the desired scalability and performance.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
The Barrelfish implements the multikernel OS by separating the CPU driver from the monitor components. The CPU driver runs in privileged mode while the monitor process runs in a distinguished user-mode. No state sharing policy allows CPU drivers to be event-driven, single threaded and event-driven. This in turn results in an easily developed small-sized kernel. In contrast to CPU drivers, monitors are independent of the processor and are schedulable. The Barrelfish also uses **System Knowledge Base (SKB)** to maintain knowledge of the underlying hardware which can be utilized for optimization.

Thus, the Barrelfish demonstrates satisfactory performance and meets all the goals set by the authors.

Write down one question you may want to ask the authors.

The authors consistently condemn memory sharing, yet suggest a limited amount of sharing as an optimization on top of message passing. What would be the specific use-cases where this kind of optimization could yield better performance?

Paper title: FlexSC: Flexible System Call Scheduling with Exception-Less System Calls
Paper authors: Livio Soares, Michael Stumm
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
This paper focuses on the drawbacks of synchronous system calls. Generally, the system call invocation involves raising a synchronous exception that surrenders the user-mode execution to a kernel-mode exception handler. The application expects the completion of the system call before continuing execution. These system calls have two kinds of overhead costs associated with them. Direct cost is associated with mode switching where the raised exception flushes the processor pipeline. There is also an indirect cost from the system call pollution of the processor structures which refers to writing of kernel-mode processor state on the L1 caches, TLB and other processor structures. High frequency system call invocation causes user-mode IPC degradation mainly due to the direct cost, whereas for medium frequency, indirect cost is the prime source of this degradation. There is also a negative impact on the kernel-mode IPC due to frequent mode switches.

System call batching: **Cassyopia** assembled independent system calls as a single multi-call. **Xen** and **VMware** also bear this feature. However, multi-calls do not support parallel execution or deal with blocking system calls.

Locality of Execution and Multicores: **Cohort scheduling, Soft timers, Lazy receiver processing** are some of the proposed techniques to improve locality of execution. **Computation Spreading** introduces migration of threads and specialization of cores. **Corey** and **Factored Operating System** proposed core specialization but required micro-kernel OS and were unable to dynamically adapt the core utilization to the workload.

Non-blocking Execution: **Capriccio** proposed improvements to user-level thread libraries for server applications. **Behren** demonstrated efficient management of thread stacks and server performance upgrade using resource aware scheduling. **Asynchronous calls** is a mechanism that implements non-blocking system calls. However, FlexSC differs from these techniques as it calls for complete separation of the invocation and execution of the system calls.

What are the key contributions of the paper?
In lieu of the drawbacks of the synchronous system calls, the authors introduce **exception-less** system calls which improves processor flexibility along with temporal and spatial locality of execution. In this mechanism, whenever a system call occurs, a kernel request is written to a reserved **syscall page**. This system call is then executed asynchronously by **syscall threads** and the result is written back to the syscall page. This method offers flexibility in two ways - batch execution of system calls, which increases temporal locality & provision of executing system calls on a different core, parallel to the user-mode threads, offers spatial locality.

The authors implement a 64-byte syscall page table entry, consisting of a system call number, arguments, status and the result. The syscall threads execute the requests in a syscall page in kernel mode, and are woken up only when the user-space execution is blocked. They can also be scheduled on a different processor core. FlexSC uses two new system calls. **flexsc_register()** is invoked by processes who want to register for using the FlexSC mechanism. Explicit registration avoids initialization overheads for processes that do not use this mechanism and involves two steps - mapping of syscall pages to user-space virtual memory and creation of one syscall thread per syscall page entry. **flexsc_wait()** is invoked when the thread is waiting for a system call execution and cannot progress without it. On execution of this system call, the kernel will put the thread to sleep and wake it up when one of the system calls is executed. The paper also offers FlexSC-Threads, a package for converting synchronous system calls to exception-less calls.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
The paper measures the overhead of executing an exception-less system call, which is upto 130% faster than a synchronous call on single-core when batching 32 or more calls. In the case of Apache, FlexSC provides an 86% improvement by using batching while an exceptional 116% improvement is seen when performing dynamic core specialization. For MySQl, a throughput improvement of 40% was observed and a BIND DNS Server demonstrated improvements between 30% to 105% depending on the request concurrency. Thus, the implementation of FlexSC was able to prove the effectiveness of the mechanisms proposed by the authors

Write down one question you may want to ask the authors.
The authors have mentioned grouping of system call requests by request type as a possible optimization. How would this improve performance of the exception-less system calls?

Paper title: The Battle of the Schedulers: FreeBSD ULE vs Linux CFS
Paper authors: J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, J. Sopena
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
This paper considers the two open-source schedulers - FreeBSD's ULE and Linux's CFS scheduler. Each of these differs in terms of design and implementation and this difference affects application performance under different workloads. It is difficult to compare the application performance for the two as it is closely intertwined with the OS subsystems, which are different for FreeBSD and Linux.

There are several similarities as well as differences between the two schedulers. While ULE is simple while CFS is more complex. Load balancing is different too - ULE regularizes the number of threads per core while CFS balances the load across cores. FreeBSD uses FIFO runqueues while Linux uses priority for threads on its runqueues. These differences cause the schedulers to affect the performance in both per-core scheduling as well as load balancing scenarios.

Previously, **Abaffy** compared the waiting time of threads in scheduler runqueues, while **Schneider** compared the performance of the network stack of the two OS. However, instead of comparing the two OS, this paper attempts to independently study the two schedulers by porting ULE to Linux. There are several other works that highlight different aspects of these schedulers, but are not as comprehensive as this one.

What are the key contributions of the paper?
Given the differences between the two schedulers, there is a variance in the performance of various applications under varying workloads. This paper attempts to analyze these differences and outlines the specific scenarios where each scheduler, owing to their design, has an upper hand.

Firstly, to perform a fair comparison, the authors ported the ULE scheduler to Linux and used it as a default scheduler to run all threads on the machine. The functions of the ULE scheduler were changed to their corresponding commands in Linux. The authors demonstrate that ULE gives interactive threads a higher priority and this might lead to starvation of the non-interactive threads, even in single application workloads. However, this starvation has its own benefits in certain cases. In terms of load balancing, CFS is faster but doesn't achieve perfect balance while ULE attains perfect balance but is slower.

These observations made by the authors help in outlining particular scenarios where each scheduler can perform better.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.

When evaluating the per-core scheduling performance, it was observed that CFS prefers fairness for all threads, while the ULE prioritizes interactive threads over batch threads. The paper considers a multi-application workload where one of the applications is compute-intensive (fibo) while the other one is mostly asleep (sysbench). Under CFS, fairness is chosen and so both applications share the resources. This results in a longer completion time for sysbench as both applications share the resources. Under ULE, however, sysbench is classified as an interactive application resulting in no shared resources and early completion, after which the compute-intensive thread is allowed to run alone. Since applications can run alone, they are able to use the cache more efficiently and are faster on ULE as compared to CFS. This is particularly beneficial for latency-sensitive applications. Further, for single-application workloads, the starvation caused by ULE helps in avoiding oversubscription, and allows applications to perform better when all the threads are fighting for the same task. Applications whose threads do not sleep do not have any impact of starvation.

Next is the analysis of load balancing. In terms of the time taken to balance a static workload on all cores, ULE takes longer as the load balancer only migrates one thread at a time. In contrast, load balancing happens much faster on CFS although it never achieves perfect load balance as it only attempts to balance the load when the imbalance is significant. In terms of thread placement, ULE takes a long time because of a cascading barrier between the threads which prevents batch threads from waking up other threads. CFS is fair and quickly wakes up the threads, but fails to achieve perfect load balance. Altogether, ULE performs 2.75% better than CFS.

Write down one question you may want to ask the authors.
If CFS was programmed to attain perfect balance, what would be the extent in which it would positively affect multi-application workloads?

Paper title: Everything You Always Wanted To Know About Synchronization But Were Afraid To Ask
Paper authors: T. David, R. Guerraoui, V. Trigonakis
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
The paper offers a thorough insight into synchronization schemes and how it is dependent closely on the hardware, e.g. the cache-coherence protocols. The authors analyze the varying latencies of the cache-coherence protocols, the performance of various atomic operations, the locking and message passing techniques along with a concurrent hashtable, in-memory key-value store and transactional memory in software. Further, observations were made by varying the architecture, from single-socket uniform and non-uniform to multi-socket multi-core directory and broadcast.

Hackenberg conducted studies on cache-coherence protocols in multi-sockets, measuring the latency of only data loading. **Molka** studied the effect of memory hierarchy on various workloads which was not very relevant to high contention systems. **Moses** demonstrated a decrease in performance of spin locks under non-uniformity, but this was confined to one type of lock and hardware model. This paper builds upon the previous works and generalizes the effect on synchronization for a variety of scenarios.

Mellor-Crumney, **Anderson** and **Luchangco** observed the limited scalability of specific locks, which did not cover different hardware platforms. The paper also presents some optimizations targeted at improving the scalability of the plain spin locks.

Several works study the OS scalability on multi-core systems and restructure the OS. **Tornado**, **Corey**, **Barrelfish** and **fos** are few of the works in this domain that look at tweaking the kernel design. This paper however shows how these issues stem from the hardware and offer software techniques and optimizations can eliminate the need for a kernel overhaul.

What are the key contributions of the paper?
The paper offers an exhaustive analysis of synchronization by building a cross-platform synchronization suite called SSYNC. This implementation includes various state-of-the-art lock algorithms, implemented with certain optimizations. It also abstracts message passing for various platforms. Further, it provides microbenchmarks that can compute the latencies of the cache-coherence protocols, the locks and the message passing. SSYNC also provides libraries that can be used to build software implementation of a portable transactional memory, a concurrent hash table, and a key-value store.

This paper leverages SSYNC to perform measurements on various platforms. Two multi-socket multi-core systems were selected - 4-socket directory-based AMD Opteron and 8-socket broadcast-based Intel Xeon. Two chip multi-processors (CMPs) were selected - 8-socket uniform Sun Niagara 2 and 36-core non-uniform Tilera TILE-Gx36. This wide range of platforms allows for a comprehensive study.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
Using SSYNC, the authors provide an insight into the scalability of various synchronization schemes across the above mentioned hardware architectures. It was observed that inter-socket latencies for any operation on cache lines does not scale, even under no contention. Thus, sharing across sockets should be avoided. Even in cases where threads are explicitly placed on the same socket, an incomplete cache directory and a non-inclusive LLC may give rise to inter-socket communication, which is expensive, as observed on Opteron. This can however be overcome by explicitly maintaining a modified state on the cache line. Further, it was observed that load and store operations are not particularly cheaper than the atomic operations,

Under high contention, the uniformity in the distance of the cores from the LLC inside a socket improves scalability of synchronization. Also, in such situations, message passing performs better. But locks outperform message passing in cases of low contention, and so message passing can be confined to highly contended parts of the system. In particular, none of the locking schemes can be labeled as the most optimal across all architectures and workloads, and the choice of lock is based on the particular scenario. However, the authors state that an efficient implementation of ticket lock would be optimal in most situations.

Based on these observations, the authors reinforce their conclusion that scalability of synchronization is in fact based on the underlying hardware and that it can prove to be a bottleneck when scaling systems.

Write down one question you may want to ask the authors.
How would the introduction of a multi-kernel OS impact the scalability of synchronization?

Paper title: FastTrack: Efficient and Precise Dynamic Race Detection
Paper authors: Cormac Flanagan, Stephen Freund
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
This paper addresses the issue of race conditions in multithreaded programs. It considers the speed and precision tradeoff of various race detectors. Precise race detectors are generally slow - they use vector clocks to represent the happens-before relation. These vector clocks however record information about every thread in the system, and are thus expensive. For n threads, it takes O(n) space and O(n) time. In turn, several race detectors have tried to eliminate the usage of vector clocks to improve performance but this raises false alarms and compromises precision.

DJIT$^+$ is a high-performance race detector that uses vector clocks to represent the happens-before relation. **BasicVC** is a traditional VC-based race detector using a vector clock for every read and write operation on each memory location. Both these race detectors use vector clocks and provide precise detections, but are prone to performance overheads.

In contrast, there are several works exploring faster but imprecise race detectors. **Eraser** uses a LockSet algorithm which follows a lock-based synchronization principle. They may, however, report incorrect cases when the synchronization scheme varies. **Goldilocks** is a precise race detector that uses a modified version of the above LockSet algorithm, but is a complex program. **MutiRace** combines LockSet algorithm with happens-before reasoning to improve precision.

Each of these race detectors are lacking either in precision or in performance. Building on the best aspects of precise race detectors, the paper develops a race detector that is precise and enhances performance by eliminating the general use of vector clocks for all cases.

What are the key contributions of the paper?
The paper demonstrates a new algorithm called FastTrack that has improved precision along with a good performance. The authors analyzed the different scenarios causing race conditions and realized the possibility of eliminating low-performing vector clocks in the more frequently occurring scenarios. Instead, an adaptive representation is suggested for the happens-before relation, which has lesser overheads. This efficient algorithm is also simple and easy to implement. FastTrack also enhances the performance of the several dynamic analysis tools by recognizing the numerous race-free accesses.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
The authors developed a prototype implementation of FastTrack that runs on RoadRunner, a framework for dynamic analyses on multithreaded software. For comparison, several other race detectors were also tuned to run on RoadRunner - these include Eraser, DJIT$^+$, MultiRace, Goldilocks and BasicVC.

FastTrack uses the following principles for faster and more precise detections. Instead of recording the clock of the last write operation to each variable by each thread, the FastTrack algorithm observes that every write on a variable occurs according to the happens-before relation and so it only records the last write on any variable and tracks the clock as well the thread. This pair is known as an epoch, and it requires only constant space as compared to the O(n) space requirement of vector clocks for n threads. In case of read operations, if they occur on thread-local and lock-protected data, the operations can be considered as ordered. So FastTrack records only the epoch of the last read on such data. The algorithm is also able to switch to vector clocks adaptively in certain cases to guarantee precision. This may happen when data becomes read-shared. It can also switch back to epochs when the condition changes.

The above principles have allowed FastTrack to perform better than the other 5 implementations that were compared. FastTrack is slightly faster than **Eraser**, while maintaining more precision than its counterpart. When compared to **DJIT$^+$** and **BasicVC**, FastTrack has equal precision but is 10x faster than BasicVC and 2.3x faster than DJIT$^+$ - a difference caused mainly due to lesser allocation of vector clocks by FastTrack. **MultiRace** performed similarly to DJIT$^+$ but had a memory footprint larger than DJIT$^+$ and had a small percentage of operations that required an Eraser operation which introduced additional overhead. Lastly, **Goldilocks** proved to be a complicated algorithm with issues in JVM integration and did not offer any significant improvement in comparison to FastTrack.

Write down one question you may want to ask the authors.
What kind of updates would be required for FastTrack to be able to efficiently detect race conditions across distributed systems?

Paper title: Shared Memory Consistency Models: A Tutorial
Paper authors: Sarita V. Adve, Kourosh Gharachorloo
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
The paper talks about the various hardware-based shared memory models, used across uniprocessor and multiprocessor architectures. It addresses the differences in the behavior expected from the various memory consistency models, in particular, the various hardware and compiler optimizations that are enabled by each model. This model decides how a program runs on a particular system, and so it becomes significant to understand the effect of the model. The model affects *programmability*, *performance* and the *portability* of a program and thus has a huge impact on how parallel programs are developed.

For utilizing certain hardware and compiler optimizations without violating sequential consistency, several techniques have been discussed. **Kourosh et al.** provide two techniques for systems with hardware support for cache coherence - prefetching ownership of write operations delayed due to program order in cache-based systems using invalidation-based protocol & rolling back and reissuing read operations in dynamically scheduled processors. **Shasha and Snir** created an algorithm to detect the *safe* memory operations which do not violate sequential consistency when they are reordered, which can be used to implement the hardware and compiler optimizations. These works have been simplified and a clear understanding is imparted through the paper.

What are the key contributions of the paper?
To provide a thorough understanding of the various memory consistency models, the paper uses simple and uniform terminology, along with program examples to explain the caveats of each model. The paper also clears several misconceptions related to these models, by providing coherent examples. This can thus be used by computer science professionals to predict the behavior of these models and effectively write programs and optimizations based on them. Since most of the models have been explained in terms of the system optimizations enabled by them, this paper attempts to provide an alternative programmer-centric view that describes the models in terms of program behavior rather than hardware and compiler optimizations.

Uniprocessors require simple semantics for memory operations - the memory operations must occur sequentially as specified by the program order. This is supported by maintaining the uniprocessor and control dependencies.

In shared memory multiprocessors, sequential consistency is primarily used, which has two main requirements. Program order among operations from the same processor must be maintained (*program order*) and a single sequential order among operations from all the processors must be maintained (*atomicity*). In architectures that do not have a cache, the following hardware optimizations may violate sequential consistency - write buffers having bypassing capability, overlapping write operations and non-blocking read operations. Overlapping write operations can avoid violation of sequential consistency by enforcing a write operation from a processor to wait until the previous write operation from the same processor has reached its memory module by using an acknowledgement response. In architectures that do have a cache, there are three additional problems - ensuring cache coherence, detecting the completion of write operations and the maintenance of atomicity for the write operations. Furthermore, the compilers for shared memory parallel programs cannot directly apply several optimizations while also preserving sequential consistency.

Several *relaxed* memory consistency models have been suggested as an alternative to sequential consistency. One way to distinguish models that relax program order consistency is by the type of order they relax, which can be categorized as relaxation from a write to a subsequent read, between two writes, or from a read to a subsequent read or write. The paper also discusses a set of models that relax the program order between all the operations - weak ordering (WO) model, two flavors of the release consistency models namely RCsc and RCpc, Digital Alpha, SPARC V9 Relaxed Memory Order (RMO) and IBM PowerPC models. Models that relax the write atomicity requirement can be differentiated by their ability to permit a read operation to retrieve the value of a write operation from another processor prior to all cached copies of the accessed location receiving the invalidation or update message generated by the write.

The paper also suggests a programmer-centric specification that requires the programmer to provide information about the operations that may be involved in a race in a program which can be utilized by the system to decide the application of a particular optimization. This information can either be passed through the programming language or through the hardware.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
There is no explicit experimentation in this paper and thus, there is no elaborate answer for this question.

Write down one question you may want to ask the authors.
Can we eliminate the problem of portability of programs across hardwares with different memory consistency models by conveying the information about synchronization operations at the programming level?

Paper title: Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel
Paper authors: Andrea Arcangeli, Mingming Cao, Paul E. McKenney and Dipankar Sarma
Your name: Parth Kansara (115135130)

What problem does the paper address? How does it relate to and improve upon previous work in its domain?
The paper looks at read-copy update (RCU) as a concurrent update mechanism which allows for a read-only access to data structures without the requirement of locking. It has been established to provide better performance as compared to other mechanisms, yet there is not one implementation of RCU that outperforms others. There are trade-offs in latencies and system overheads and none of the existing implementations have surpassed in both.

McK02a showed that there is no best overall algorithm. *rcu-poll* has the shortest latency, while *rcu-ltimer* has the lowest system overhead.

Herlihy93 proposes other methods for getting rid of the lock mechanisms which can be leveraged in combination with some refinements from **Michael02a** and **Michael02b**. However, these still count on expensive atomic operations on the shared storage, which causes pipeline stalls, cache thrashing and memory contention even on read-only accesses.

Gamsa99 and **McK02a** offer an RCU implementation that does not need to suppress preemptions but it results in longer grace periods, which is highly undesirable.

What are the key contributions of the paper?
The paper proposes an RCU-based implementation of the Linux System V primitives, which combines the best features of several RCU implementations. *rcu-poll* uses interrupts to force CPU into a quiescent state, which results in low grace periods but high scheduler overheads. *rcu-ltimer* on the other hand calls the scheduler_tick() function on each CPU with a certain frequency, which has extremely low overheads but a longer grace period. *rcu_sched* uses token-passing, which results in extremely low overheads, while having extremely long grace periods. The paper addresses the high overheads in the *rcu-poll* algorithm and works to eliminate the cache thrashing, but this modified *rcu-poll* is outperformed by *rcu_ltimer.*

Further, the paper explains how the analogy between the reader-writer-lock and RCU was used to replace the global locks in the System V IPC primitives. The global locks were replaced by RCU which guarded the mapping arrays. For guarding the IPC operations, the *per-kern-ipc-perm* locks were used. This modification results in a significant speed up on the database benchmarks.

Briefly describe how the paper's experimental methodology supports the paper's conclusions.
The paper describes in depth the various changes made to the System V's semaphores. Both the *ipc_id* array and the *sem_array* were prefixed with a structure called the *ipc_rcu_kmalloc* containing the *rcu_head* structure. This structure is used by the *call_rcu()* function to track the structures during a grace period. Also, the global *ary* lock is replaced by individual locks for each *sem_array* to guard operations on the corresponding set of semaphores. Lastly, to avoid stale data during the translation of a semaphore ID to *kern_ipc_perm*, a *deleted* flag is set in the *kern_ipc_perm*, whenever a particular *sem_array* is removed.

Semaphore removal uses *ipc_rmid* which sets the *deleted* flag and uses *ipc_rcu_free()* to free up the memory of the semaphore. For acquiring a lock on the semaphore state, the *semop()* system call invokes the *ipc_lock().* Since *ipc_lock()* does not block, the semaphore structure is kept alive for the RCU grace period so that it is not lost before the *ipc_lock()* can check the *deleted* flag. During the grace period, the *call_rcu()* function uses the *rcu_head* structure to queue up the semaphore structure. The *kfree()* function is invoked after the end of the grace period. RCU pushes the expansion of the *ipc_id* array to occur in parallel with the searching of the *ipc_lock()* using the *grow_ary()* function. No *deleted* flag is needed as the old version of the array is kept valid throughout the grace period.

The above modifications were made to the RCU implementation on the System V's semaphores which showed a significant improvement in performance. The system showed an reduction in runtime by an order of magnitude on *semopbench,* a System V semaphore microbenchmark. Also these changes only caused an increase of 5% in the overall lines of code, which is a negligible increase in complexity for the scale of improvement offered in terms of performance.

Write down one question you may want to ask the authors.
Can we eliminate the problem of portability of programs across hardwares with different memory consistency models by conveying the information about synchronization operations at the programming level?