QUESTION 1

The Clipper (an architecture developed by Fairchild) had a status bit in the System Status Word (SSW) called FRD for "Floating Registers Dirty. This bit could be cleared by the OS and was set by the hardware whenever a floating-point register was stored to.  Discuss how one can take advantage of that bit to reduce unnecessary register copying during a context switch.

Ans: The system status word is used to record the interrupt number and level, enable interrupts, set the mode (user/supervisor), and set the protection key. It can only be written in the supervisory state. Overall, these status registers are critical to the operation of the processor and are used by both the operating system and user processes to manage and control the execution of instructions and the handling of interrupts.

The performance of context switches can be increased by using the FRD (Floating Registers Dirty) bit in the System Status Word (SSW) of the Clipper microprocessor. During a context switch, the operating system must save the processor's current state, including the contents of all registers. The need to save a register's value once more does not exist if it has not changed since the previous context switch, which can be a substantial optimization. To identify which floating-point registers have changed since the last context transition and need to be saved, the operating system can examine the FRD bit.

The operating system can determine the FRD bit's value before switching contexts. The operating system can forego preserving the values of any floating-point registers that have not changed since the last context transition if the FRD flag is not set. The operating system can similarly use the value of the FRD bit to identify which floating-point registers need to be restored when restoring the saved state after a context switch.

The operating system can "clean" the floating-point registers by clearing the FRD bit in the SSW. Each time a floating-point register is stored, the hardware sets the FRD bit to indicate that the register has been altered. The operating system can decrease the overhead of context switches and enhance system performance by avoiding needless register copying. The frequency of floating-point register alterations between context transitions, however, determines how effective this optimization is. The cost of verifying the FRD bit may outweigh the advantages of bypassing register copying if most registers are often updated.

One way that one can take advantage of the Floating Registers Dirty (FRD) bit is by looking at the current state of the System Status Word (SSW), specifically the FRD bit, to determine whether to copy the floating-point registers during a context switch. If the FRD bit is not set, this indicates that the floating-point registers have not been altered since the last time they were saved to memory, and hence any copying of the floating-point registers can be omitted. However, if the FRD bit is set,

then the floating-point registers must be copied as they have been modified since the last context switch. This saves time as unnecessary copying of unchanged or unmodified data is omitted. Additionally, this process further declutters the system and eliminates redundant operations that normally occur during context switching.

QUESTION 2

Suppose we have a system with a very large number of cores, say 256.  Further, suppose that we are designing our OS so that only one of the cores runs that OS code, and all other cores just run application code.  Discuss the circumstances under which it would make sense to do away with scheduling altogether.

Ans: It would make sense to do away with scheduling in a system with a very large number of cores if all applications are running in a consistent manner that does not require any scheduling. For example, if the applications running on the other cores are compute-bound and all have the same workload that repeats itself, then there is no need for scheduling as the processor cores are already doing the same thing in a consistent manner. However, if the system were to include applications that require scheduling, such as interactive applications or tasks that need to be timed, then scheduling could provide an advantage in resource utilization, and it may be in the best interest of the system to include scheduling.

Even in a system with many homogeneous cores, there may be variations in the workload or resource demands of different applications that require dynamic resource allocation or load balancing. Scheduling can help ensure that all applications receive an appropriate share of the available resources, preventing some applications from starving while others hoard resources. Scheduling can also help with managing shared resources. Therefore, while scheduling may not be necessary in all cases, it can still provide benefits in terms of performance and fairness in resource allocation.

In some cases, even if there is an OS core, it may make sense to forego scheduling in a system with a very large number of cores. The types of applications being used by the other cores would have an impact on the decision to stop scheduling, though. Scheduling is not necessary if all the apps using the other cores are compute-bound and have a repetitive workload because the processor cores are already performing the same task consistently. Since the same computation is being carried out concurrently on all cores without the requirement for coordination or time-sharing, scheduling might be removed in this situation to simplify the system and lower overhead.

Conversely, scheduling could offer an advantage in resource utilization if the system includes applications that need it, such as interactive apps or tasks that must be timed. In this case, including scheduling may be in the system's best interests. In this scenario, the single OS core system might coordinate scheduling and make sure that programs are run equitably and effectively, taking into consideration the workload and priority of each activity. Some programs might compete for resources and perform less effectively without scheduling.

Overall, whether to use scheduling in a system with many cores depends on the specific requirements of the applications running on the system. While scheduling may not be necessary for all scenarios, it can provide benefits in terms of performance, fairness in resource allocation, and managing shared resources. Therefore, it is important to carefully consider the needs of the system and its applications when deciding whether to include scheduling.

QUESTION 3

How would process creation work on the system described in the previous question (many cores with one dedicated to the OS)?

Ans: If there are only a few large, long-lived applications on the system that can make use of all the cores and the system has a very large number of cores, such as 256, and an OS that runs on only one core while all other cores are reserved for application code, scheduling may be completely unnecessary. The operating system could therefore assign each application to a set of cores that would be solely dedicated to that application and would not be shared with any other application.

Under such a system, a request for a new process would come from an application and be processed by the dedicated OS core. The operating system would then assign a group of cores to that process, set aside the required amount of memory, and transfer the process to those cores. The OS would have to wait until a set of cores became available if there were insufficient cores available to allocate to the new process. The operating system may employ a method known as "affinity scheduling" when allocating a set of cores to a new process. Affinity scheduling reduces cache misses and other sorts of overhead by attempting to keep a process on the same set of cores for as long as possible. This can assist increase speed. To guarantee that the workload is divided fairly among the available cores, the OS may also employ load balancing mechanisms. This can help prevent overloading some cores while leaving others idle.

It seems improbable, though, that a system with this many cores would only have a few significant, long-lasting applications in most real-world circumstances. Instead, the system is more likely to run many lightweight programs simultaneously, each requiring only a few cores. In this scenario, the OS would have to schedule the programs to execute effectively and efficiently on the available cores. Several scheduling techniques, such as round-robin, priority-based, or shortest-job-first scheduling, could be used by the OS to allocate cores to processes to achieve this. According to the relevance of each application, a priority would be assigned to it, and the OS would then assign cores to the processes in accordance with their priorities and the resources that were available.

When the system runs real-time applications, scheduling may be unnecessary in other circumstances. In this situation, the system may allocate cores to activities because time constraints are more important than making efficient use of the available resources. It's also important to remember that, even when the system has a lot of cores, there can be other things to consider before selecting whether to employ scheduling. It might be more effective to employ scheduling to capitalize on the various advantages of each core, for instance, if the cores are heterogeneous (i.e., have varied capabilities). The choice of whether to use scheduling ultimately comes down to the needs of the system and the applications that use it.

QUESTION 4

Design a mechanism for issuing system calls in the system described in the previous two questions (many cores with one dedicated to the OS).

Ans: A mechanism for issuing system calls in the system starts with a very important thing, which is a message protocol that might be used to let the application threads running on the various cores and the dedicated OS core communicate with each other to issue system calls in the system described. The messaging protocol would make it possible for the application threads to ask the OS core for system calls and receive responses back with the output data generated by the system call.

As previously mentioned, the application thread would send a request message to the OS core that would include the system call number and any additional arguments that were specified by the system call. After confirming the request's legitimacy and authorizing it, the OS core would carry out the system call on the application thread's behalf and create a response message that included any output data that the system call produced. The application thread would then get the reply message and use the output information to carry on with its work.

Overall, a system with many cores can utilize its resources effectively while protecting the security and stability of the system by using a messaging protocol and a dedicated OS core. To guarantee the best system performance, a carefully thought-out scheduling method is necessary.

To reduce overhead and ensure effective communication between the application threads and the OS core, the messaging protocol used for making system calls should be designed. Making use of a lightweight messaging protocol that is designed for low-latency communication, such as the User Datagram Protocol (UDP), is one way to do this. Making use of a lightweight messaging protocol that is designed for low-latency communication, such as the User Datagram Protocol (UDP), is one way to do this. To guarantee that the system is secure and stable, the communications protocol should also include procedures for error handling and recovery. For instance, the OS core could have a watchdog timer that checks the messaging protocol for unusual activity, such as a high number of unsuccessful requests or odd traffic patterns. The watchdog timer could launch a system-wide recovery process, such as resetting every core and returning the system to a known safe state, if an irregularity is discovered.

Additionally, the message protocol needs to provide system call prioritizing so that urgent system calls can be attended to right away. This can be accomplished by giving each system call a priority level and managing the requests sent to the OS core using a priority queue. The system calls would then be handled by the OS core in order of priority, with calls with higher priorities being addressed first.

Overall, several aspects, including latency, error handling, recovery, and prioritizing, must be carefully considered when designing the messaging protocol for making system calls in a system with one dedicated OS core and numerous application cores. The system can effectively use its resources while ensuring stability and security by implementing an effective and reliable messaging protocol that enables efficient communication between the OS core and the application threads.

QUESTION 5

Donald Knuth updated the machine he uses for his Art of Computer Programming series of books to a new 64-bit machine called the MMIX.  It has an instruction called CSWAP that operates as follows:

CSWAP $X, $Y, $Z

where $X, $Y, and $Z are registers. Let A = $Y + $Z. If M[A] = rP, where rP is the special prediction register, set M[A] to $X and set $X to 1.  Otherwise, set rP to M[A] and set $X to 0.  This is an atomic (indivisible) operation, useful when independent computers share a common memory.

Here the notation M[A] means the contents of memory location A.

Describe how to use this instruction to implement a spin lock.

Ans: The CSWAP (Conditional Swap) instruction is a special instruction on the MMIX machine, a 64-bit processor designed by Donald Knuth. The instruction has the following format:

CSWAP $X, $Y, $Z

where $X, $Y, and $Z are registers. The instruction performs an atomic swap operation on memory location M[A], where A is equal to the sum of the values in registers $Y and $Z.

A "spin lock" is a synchronization mechanism that allows only one thread to access a shared resource at a time. The basic idea of a spin lock is that a thread repeatedly checks a shared memory location (the lock variable) to see if the lock is available. If the lock is not available, the thread repeatedly performs a simple operation (such as incrementing a counter) until the lock becomes available. Once the lock is available, the thread can access the shared resource.

To implement a spin lock using the CSWAP instruction, we can use the lock variable as the memory location M [A]. Here is the algorithm for acquiring and releasing the lock:

Acquiring the lock:

Load the value 0 into the $Y register (this is the value that will be written to the lock variable).

Load the address of the lock variable into the $Z register.

Execute the CSWAP instruction with $X = 0.

Releasing the lock:

Load the value 0 into the $Y register (this is the value that will be written to the lock variable).

Load the address of the lock variable into the $Z register.

Execute the CSWAP instruction with $X = $r0.

In this algorithm, step 3 in the acquiring process is the atomic operation that checks the lock variable and either sets it to 1 (if it was 0) or leaves it unchanged (if it was 1). If the lock is not available, the CSWAP instruction sets $X to 0, indicating that the operation failed. If the lock is available, the CSWAP instruction sets $X to 1, indicating that the lock was acquired.

Similarly, step 3 in the releasing process performs the same atomic operation and sets the lock variable to 0, releasing the lock. The value of $X is set to $r0, which is the constant 0, indicating that the operation always succeeds.

By using the CSWAP instruction in this way, we can ensure that only one thread at a time can acquire the lock and access the shared resource, avoiding race conditions and other synchronization issues.

A "spin lock" is a synchronization technique used on the MMIX machine to guarantee that only one thread or process can access a shared resource at once. When a process or thread tries to acquire a spin lock that is currently being held by another process or thread, that process or thread will begin spinning in a loop and keep trying to acquire the lock until it becomes free.

A spin lock is implemented using the MMIX machine's CSWAP (compare and swap) instruction. If the value in the memory location matches the value in the prediction register, the instruction atomically swaps the value in the memory location with the value in the specified register (rP). On the MMIX machine, you would first generate a flag in memory to represent the lock state to implement a spin lock using CSWAP.

When a process or thread wants to obtain the lock, it would use the CSWAP instruction to atomically swap the flag value with a register and set the register to 1 to denote that it has obtained the lock successfully. The CSWAP instruction will fail if the flag is already set, indicating that the lock is being held by another process or thread. In this case, the process or thread will spin endlessly to acquire the lock once it becomes available. The lock will be released by setting the flag value back to its initial value once the process or thread has finished accessing the shared resource.

In concurrent programming, spin locks are a sort of synchronization technique used to prevent concurrent access to shared resources by multiple threads or processes. A thread or process must first attempt to obtain the lock by "spinning" in a loop until it becomes accessible before it may access a shared resource that is secured by a spin lock. The spinning thread or process will wait until the lock is released if it is already being held by another thread or process before attempting to retake it.

CSWAP is an example of an atomic instruction, which means that it is executed atomically and cannot be stopped or interspersed with other instructions. Because it guarantees that operations on shared resources are completed without disruption from other threads or processes, this is crucial in concurrent programming. The CSWAP instruction offers a mechanism to conditionally execute the swap based on the value of a prediction register and a way to atomically swap the value of a shared memory location with the value in a register, making it handy for building spin locks. This makes it possible to synchronize shared resources in a concurrent context in an effective and trustworthy manner.

The CSWAP instruction is particularly useful in the implementation of synchronization primitives such as spin locks, where multiple threads or processes may attempt to access a shared resource concurrently. Because of the atomicity of the CSWAP procedure, only one process can receive the lock at a time, and conditional behavior lets processes "spin" until the lock becomes available.

Spin locks are a common synchronization technique in concurrent programming, and they can be effective in situations where the lock is expected to be held for only a short period of time and contention for the lock is low. However, if contention for the lock is high or if the lock is held for a long time, spin locks can lead to wasteful CPU usage as threads spin in a loop waiting for the lock to become available. In such cases, other synchronization techniques such as semaphores or mutexes may be more appropriate.

QUESTION 6

Core memory involved a destructive read operation where, to determine whether a core stored a 0 or a 1, it had to be set to 0. It was then required that a read be immediately followed by a write to restore the value that was stored there. This led to what was known as the "read-modify-write" memory cycle. It was used for various purposes, such as being able to increment a variable in memory while it could be read. Describe how this read-modify-write cycle made a test-and-set instruction natural to implement.

Ans: A read-modify-write cycle is a process that involves reading data from a memory location, modifying the data in some way, and then writing the modified data back to the same memory location. This cycle is commonly used in computer memory systems to perform operations such as incrementing a variable or setting a flag.

A test-and-set instruction is a type of atomic operation used in concurrent programming to provide mutual exclusion for shared resources. It is used to set a flag in memory to indicate that a particular resource is currently being used, while also checking the previous state of the flag to ensure that only one thread is accessing the resource at a time.

The implementation of a test-and-set instruction typically involves using a read-modify-write cycle to atomically test the value at a given memory location and set it to a new value if the previous value meets certain criteria. The basic idea is to read the value, modify it if necessary, and then write it back to the same memory location in a single, indivisible operation to prevent race conditions and ensure atomicity.

The read-modify-write cycle in computer memory made a test-and-set instruction natural to implement because it provided a way to perform an atomic test and modification of a memory location. A test-and-set instruction is used in concurrent programming to ensure mutual exclusion when accessing a shared resource. It involves atomically testing the value of a lock or flag and setting it to a new value in a single operation, guaranteeing that no other thread can access the shared resource between the test and the set.

With the read-modify-write cycle, a processor could perform a read operation to check the value of the lock or flag, modify it in memory, and write it back to memory in a single atomic operation. This ensured that the value of the lock or flag was not changed by any other processor between the read and the write and that the modification of the lock or flag was performed atomically.

The read-modify-write cycle provided a natural way to implement the test-and-set instruction in hardware, without the need for complex software or synchronization mechanisms. It was particularly useful in early computer architectures that used core memory, where the destructive read operation made it necessary to perform a write operation to restore the value of a memory location after a read.

The read-modify-write cycle allowed for a simple way to implement a test-and-set instruction using core memory, which was useful for ensuring mutual exclusion in concurrent programming. By setting a memory location to a known value and then checking and updating it using the read-modify-write cycle, the latch could be used to ensure that only one process could access a shared resource at a time.

Thus, the read-modify-write cycle allowed for the implementation of the test-and-set instruction with guaranteed atomicity, making it an important building block for concurrent programming and operating system design. The read-modify-write cycle in core memory made it natural to implement a test-and-set instruction because it allowed for an atomic test and modification of a memory location. This was particularly useful in early computer architectures that used core memory, where the destructive read operation made it necessary to perform a write operation to restore the value of a memory location after a read. The read-modify-write cycle provided a natural way to perform an atomic test and set operation without the need for complex software or synchronization mechanisms.

Suppose we have a shared resource that multiple threads need to access. We want to ensure that only one thread can access the resource at a time to avoid data inconsistency. To achieve this, we can use a test-and-set instruction to set a lock flag in memory that indicates whether the resource is currently being used.

Here's how we could use the read-modify-write cycle to implement the test-and-set instruction in pseudocode:

// Initialize the lock flag to 0 to indicate that the resource is not in use.

lock\_flag = 0

// Function to acquire the lock

acquire\_lock():

    // Loop until we successfully set the lock flag to 1

    while test\_and\_set(&lock\_flag) == 1:

        // The lock is already held, so wait before trying again.

        sleep(1)

// Function to release the lock

release\_lock():

    // Set the lock flag back to 0 to indicate that the resource is no longer in use.

lock\_flag = 0

// Function to perform the test-and-set operation using the read-modify-write cycle

test\_and\_set(address):

    // Read the current value at the given memory address

value = read(address)

    // Set the value to 1 to indicate that the lock is being held.

    write(address, 1)

    // Return the original value that was read

    return value

The test\_and\_set() function uses the read-modify-write cycle to atomically test the value at the given memory address and set it to 1 if it is currently 0 (indicating that the lock is not currently held). This ensures that only one thread can acquire the lock at a time. The acquire\_lock() function uses the test\_and\_set() function to set the lock flag to 1, and it loops until it successfully acquires the lock. The release\_lock() function simply sets the lock flag back to 0 to indicate that the resource is no longer in use.

The exact implementation of the test\_and\_set() function will depend on the architecture and hardware being used. Some modern CPUs have instructions specifically designed for atomic operations, such as the compare-and-swap instruction. However, the basic idea of using a read-modify-write cycle to implement an atomic test-and-set operation is still relevant and widely used in concurrent programming.

QUESTION 7

The CDC 6000 series computers (originally designed by Seymour Cray) had a set of peripheral processing units (PPUs) that were independent of the main CPU. (Some models had a single main CPU, and some had two.) The PPU had an instruction called the exchange jump (EXN). There was a PPU register called A that specified the address of a block of memory. When the PPU executed the EXN instruction, the main CPU was interrupted, all its registers were exchanged with the block of memory pointed to by the PPU A register, and the main CPU resumed operation. If the operating system is running on a PPU, describe how to perform a context switch using the EXN instruction. (Note: This question is not asking you to implement the EXN instruction.  It is asking you to use the EXN instruction to implement a context switch.)

Ans: The EXN (Exchange Jump) instruction on the CDC 6000 series computers was designed to exchange the contents of the main CPU's registers with a block of memory pointed to by a peripheral processing unit (PPU) register A. This instruction can be used to implement a context switch on a system where the operating system is running on a PPU. Here is the implementation of a context switch using the EXN instruction:

Save the current context of the CPU:

The PPU loads the CPU's registers into a memory block pointed to by register A.

The PPU sets the program counter to the address of the next instruction after the EXN instruction.

The PPU executes the EXN instruction, which saves the CPU's context in the memory block pointed to by register A.

Load the new context:

The PPU loads the CPU's registers with the values of the new context from a memory block pointed to by register A.

The PPU sets the program counter to the value of the new context's program counter.

Point to the next context:

The PPU updates register A to point to the memory block of the next context.

The PPU returns control to the CPU.

When a context switch is needed, the PPU can execute the above steps to save the current context, load a new context, and update register A to point to the next context. The CPU will resume execution with the new context loaded into its registers. The PPU can continue to execute instructions and perform other tasks while the CPU is executing.

It should be noted that the operating system will need to maintain track of the locations of the memory blocks containing each process' context, as well as the context of the currently running process. The address of the current process's context can be stored in the PPU A register during execution so that it can be conveniently accessible when it's time to switch back.

It's crucial to remember that before a process can run, the operating system must make sure that its context is correctly initialized. This entails changing the program counter, stack pointer, and other pertinent registers to the proper values. An improper context initialization might lead to unpredictable behavior and potential crashes.

The operating system will also need to make sure that each process has its own unique memory block for its context to avoid data corruption or other problems that can occur if multiple processes access the same memory block concurrently. Memory management strategies like virtual memory or memory segmentation can be used to achieve this.

Process scheduling is a crucial component of operating system functionality that involves determining which processes should be executed and in what order. The operating system must maintain information about each process, including its priority, time slice, and other relevant data, and use this information to decide which process to run next. The process of selecting which process to run is typically performed by a scheduling algorithm, which can vary depending on the specific needs of the system.

Memory allocation is another critical component of operating system functionality, particularly on systems with limited memory resources. The operating system must manage the allocation and deallocation of memory for each process, ensuring that each process has access to the resources it needs without interfering with other processes or system components. This may involve allocating and deallocating memory blocks, updating data structures that track memory usage, and managing memory access for each process.

In addition to process scheduling and memory allocation, the operating system may also need to manage other aspects of its operation, such as input and output operations, file system access, and hardware interrupts. These tasks may involve updating data structures that track the status of input and output operations, handling interrupts from hardware devices, and performing other tasks as necessary to ensure the smooth operation of the system.

Overall, the process of performing a context switch on a CDC 6000 series computer involves a complex interplay of hardware and software components and requires careful management of a wide range of system resources to ensure the efficient and reliable operation of the system.

Example: Suppose we have two processes, Process A and Process B, running on a CDC 6000 series computer. Each process has its own set of CPU registers and its own block of memory in which it stores its context. We want to switch between these two processes using the EXN instruction.

Save the context of Process A: The PPU loads the registers of Process A into memory block A, which is pointed to by the A register. The EXN instruction is then executed, which causes the CPU to exchange its registers with the contents of memory block A. As a result, the context of Process A is saved in memory block A.

Load the context of Process B: The PPU now sets the A register to point to memory block B, which contains the context of Process B. The EXN instruction is then executed again, which exchanges the contents of memory block B with the CPU's registers. As a result, the context of Process B is now loaded into the CPU, and the CPU begins executing instructions from Process B.

Save the context of Process B: The PPU loads the registers of Process B into memory block B, which is pointed to by the A register. The EXN instruction is then executed, which causes the CPU to exchange its registers with the contents of memory block B. As a result, the context of Process B is saved in memory block B.

Load the context of Process A: The PPU now sets the A register to point to memory block A, which contains the context of Process A. The EXN instruction is then executed again, which exchanges the contents of memory block A with the CPU's registers. As a result, the context of Process A is now loaded into the CPU, and the CPU begins executing instructions from Process A.

This process can be repeated to switch back and forth between the two processes as many times as necessary. Each time the EXN instruction is executed, the CPU's registers are exchanged with the contents of the memory block pointed to by the A register, allowing the context of a new process to be loaded into the CPU.