Arquitectura del Computador
y
Sistemas Operativos

Vigésima Clase

Ambientes multiproceso y/o multihilo - Variables atómicas (1/5)

Asumamos que hay un recurso, como un disco, que no puede ser usado por dos a la vez. La forma de proceder es serializar los accesos, colocando las solicitudes una detrás de otra y sirviéndolas secuencialmente.

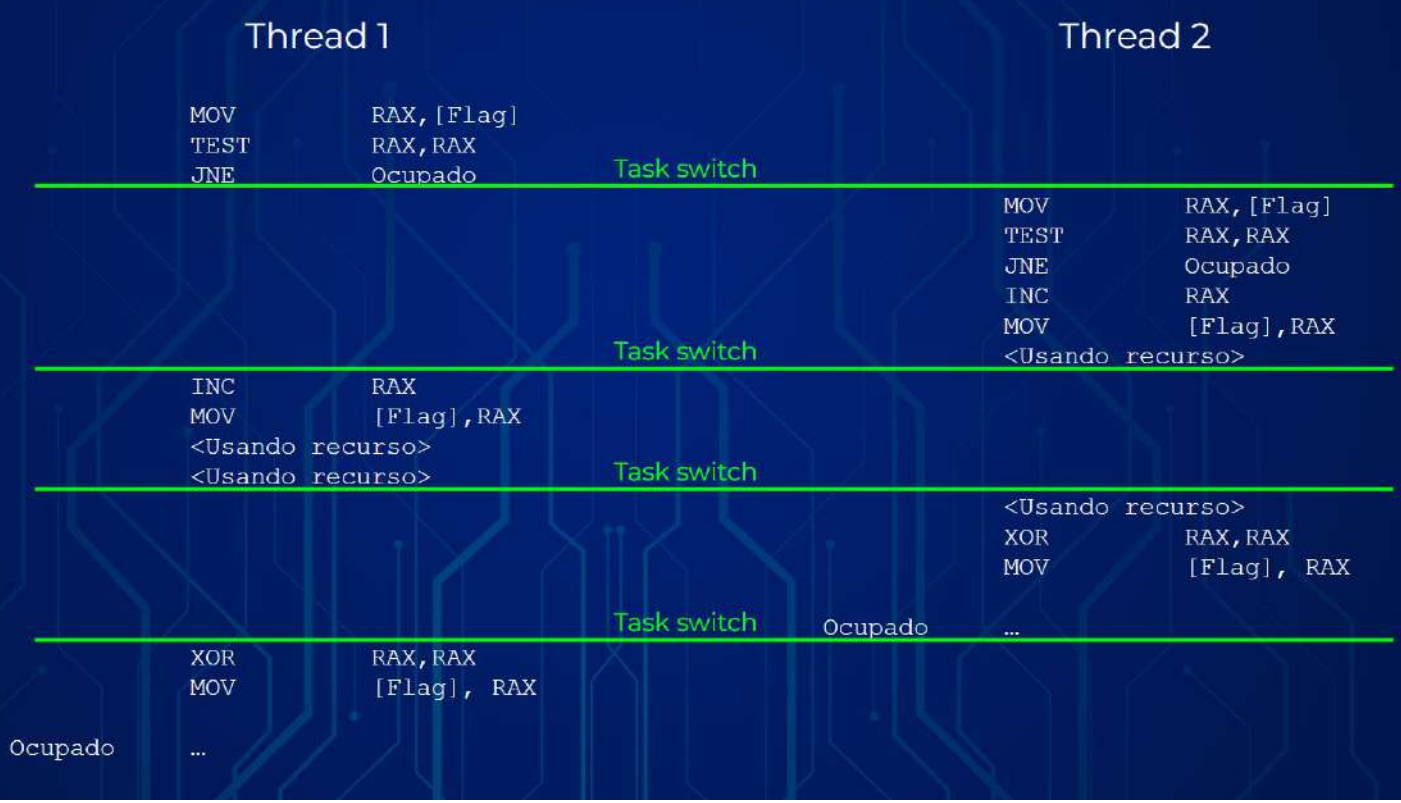
El siguiente código podría usarse para implementar un acceso secuencial:

```
            MOV         RAX,[Flag]      ; Levantar la variable Flag
            TEST        RAX,RAX         ; Si es 1, esperar que el recurso se libere
            JNE         Ocupado
            INC         RAX             ; Marcar el recurso como ocupado
            MOV         [Flag],RAX

            <Usando recurso>
            <Usando recurso>

            XOR         RAX,RAX         ; Indicar que está libre
            MOV         [Flag], RAX

Ocupado     …
```

Y este código funciona correctamente cuando una sola tarea lo usa.

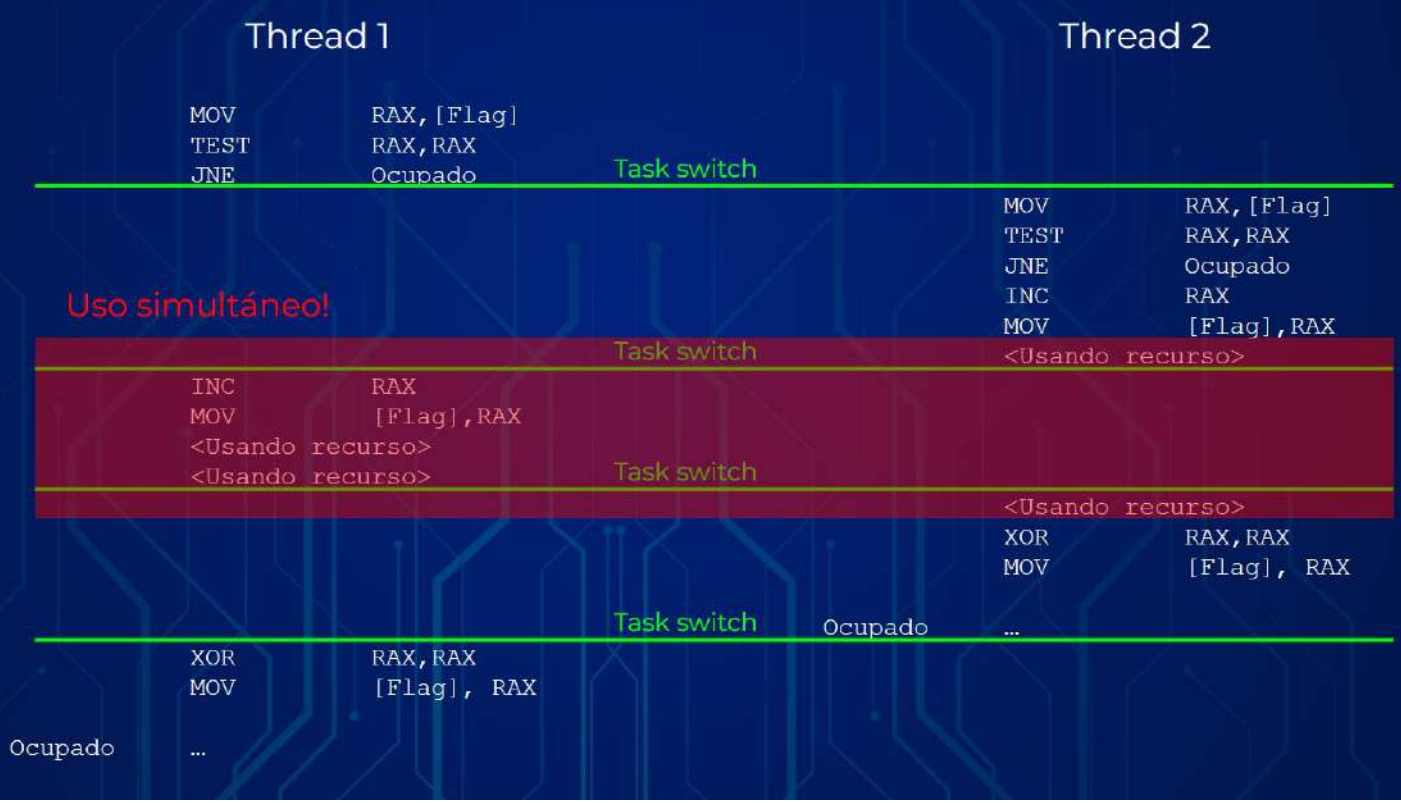
Ambientes multiproceso y/o multihilo - Variables atómicas (2/5)

Ahora veamos qué pasa en un ambiente multihilo:



Ambientes multiproceso y/o multihilo - Variables atómicas (2/5)

Ahora veamos qué pasa en un ambiente multihilo:



Ambientes multiproceso y/o multihilo - Variables atómicas (3/5)

Podemos pensar en la siguiente alternativa:

```
            MOV       RAX, 1          ; Levantar la variable Flag y ponerla en 1
            XCHG      RAX,[Flag]      ; si está en 0.
            TEST      RAX,RAX         ; Si era 1, esperar que el recurso se libere
            JNE       Ocupado
            <Usando recurso>
            <Usando recurso>
            XOR       RAX,RAX         ; Indicar que está libre
            XCHG      RAX, [Flag]
Ocupado     …
```

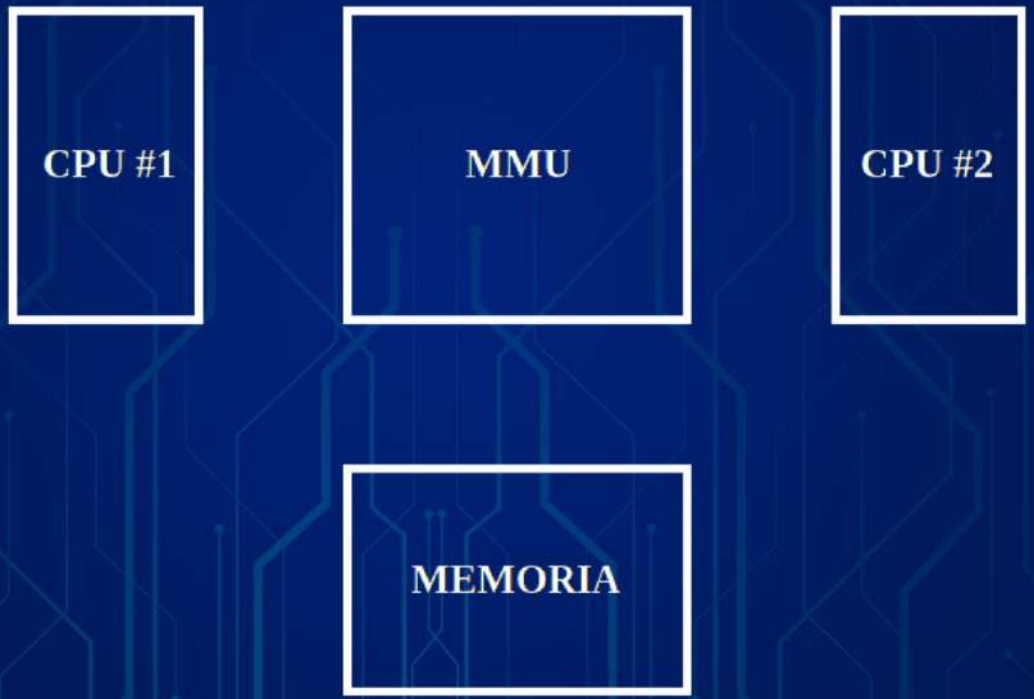
Este código funciona perfectamente en un ambiente multihilo, siempre y cuando haya un solo procesador.

Éso se debe a que cuando el Sistema Operativo cambia de un hilo a otro, siempre lo hace al final de una instrucción. En otras palabras una instrucción se ejecuta completa o no se ejecuta en lo absoluto. Por ende el XCHG se ejecuta o no, y éso sólo toma el recurso.

A este tipo de instrucciones se las conoce como *Read-Modify-Write*

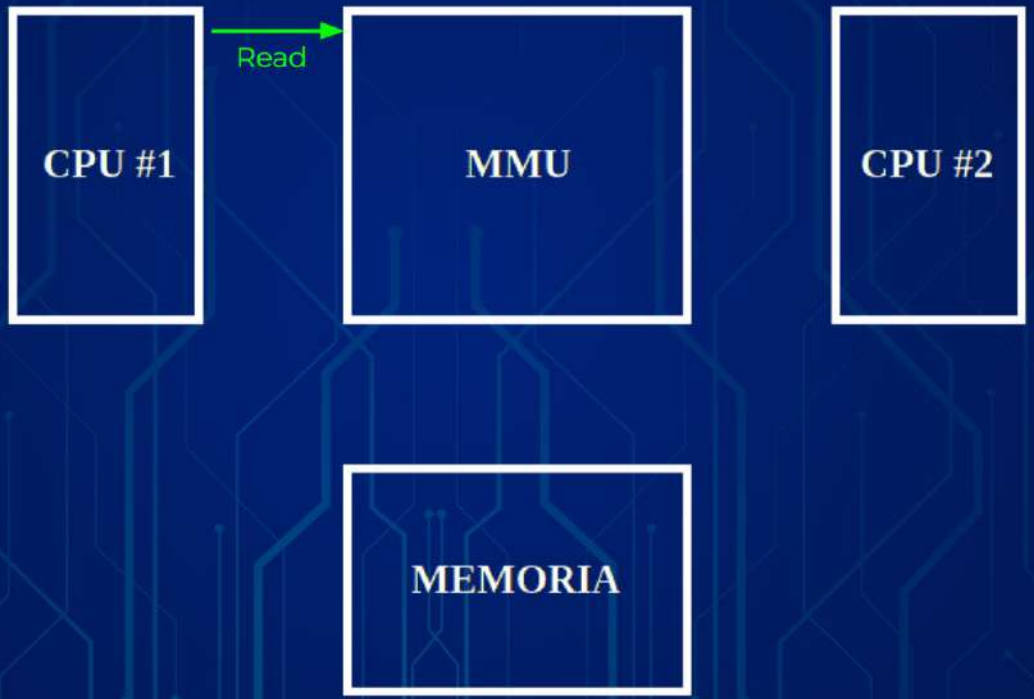
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



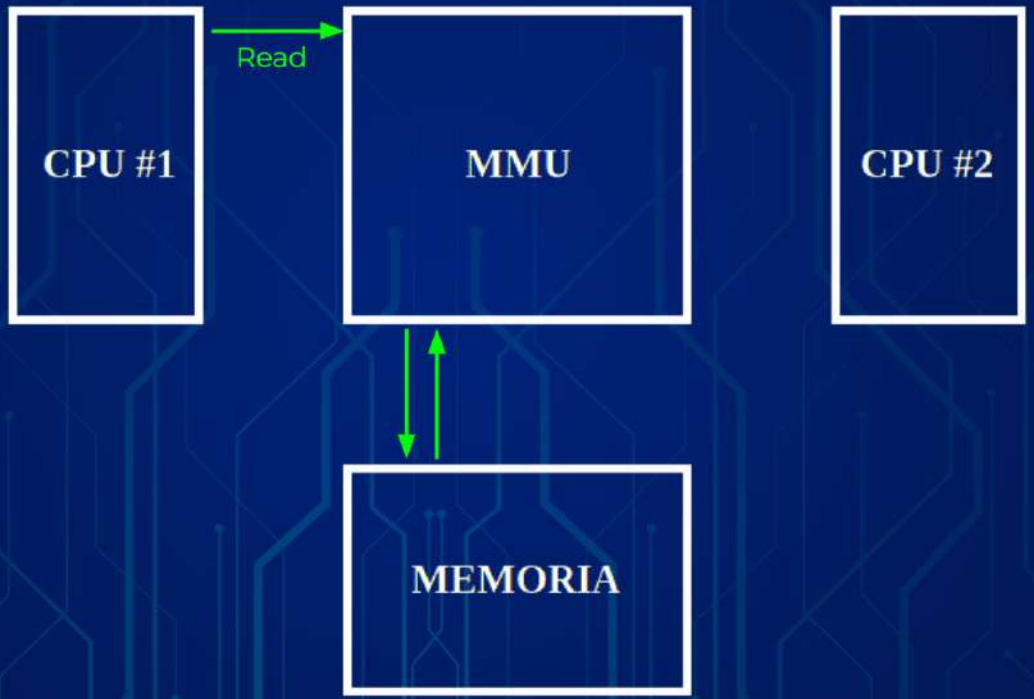
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



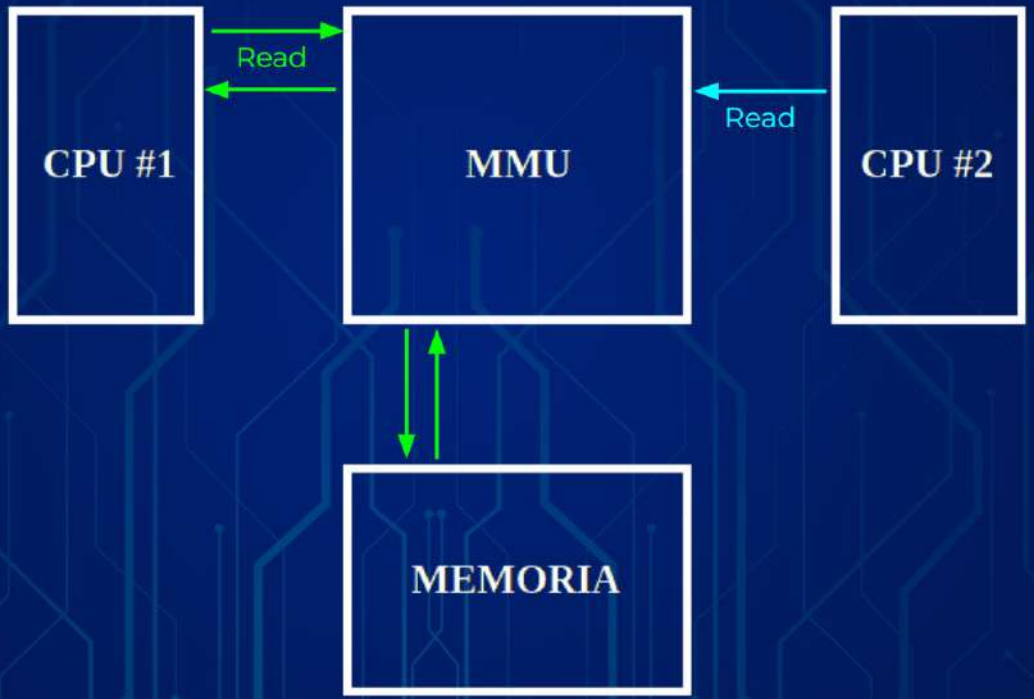
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



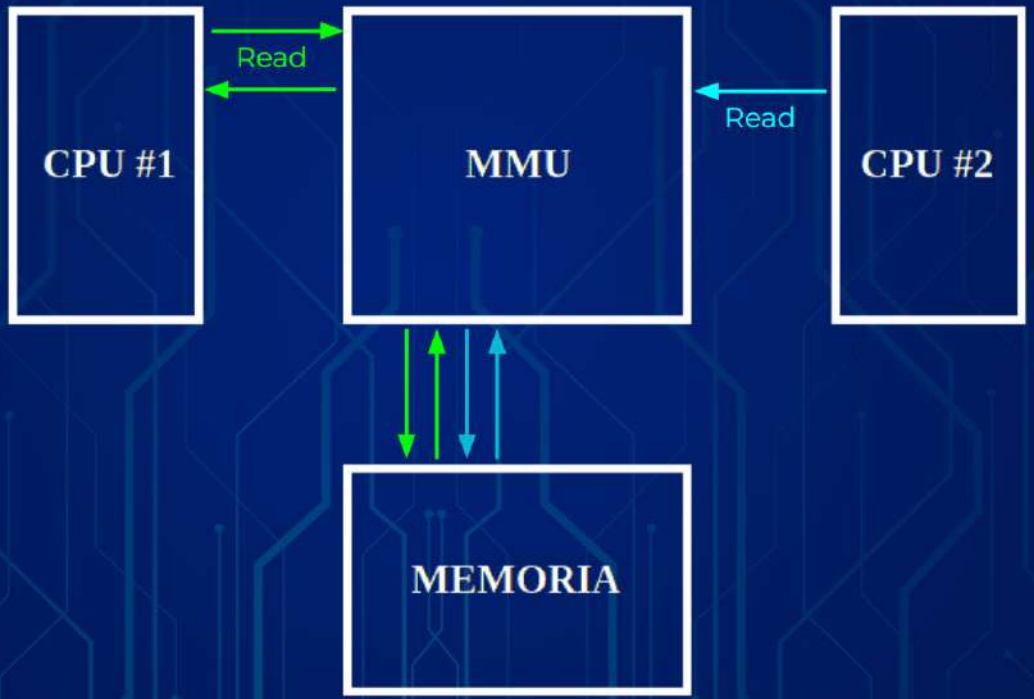
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



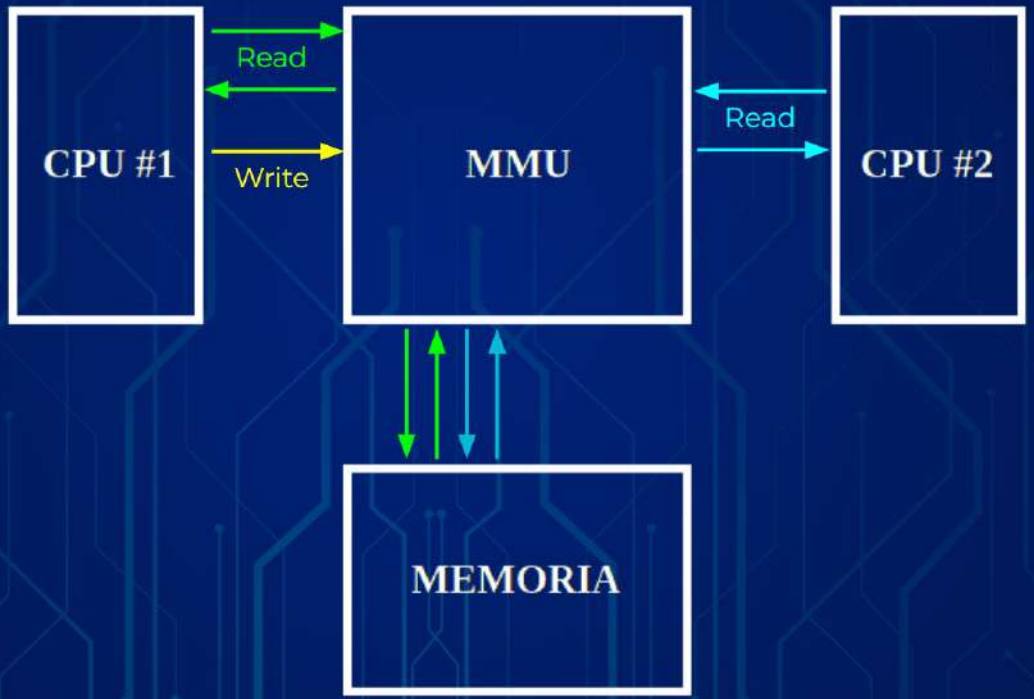
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



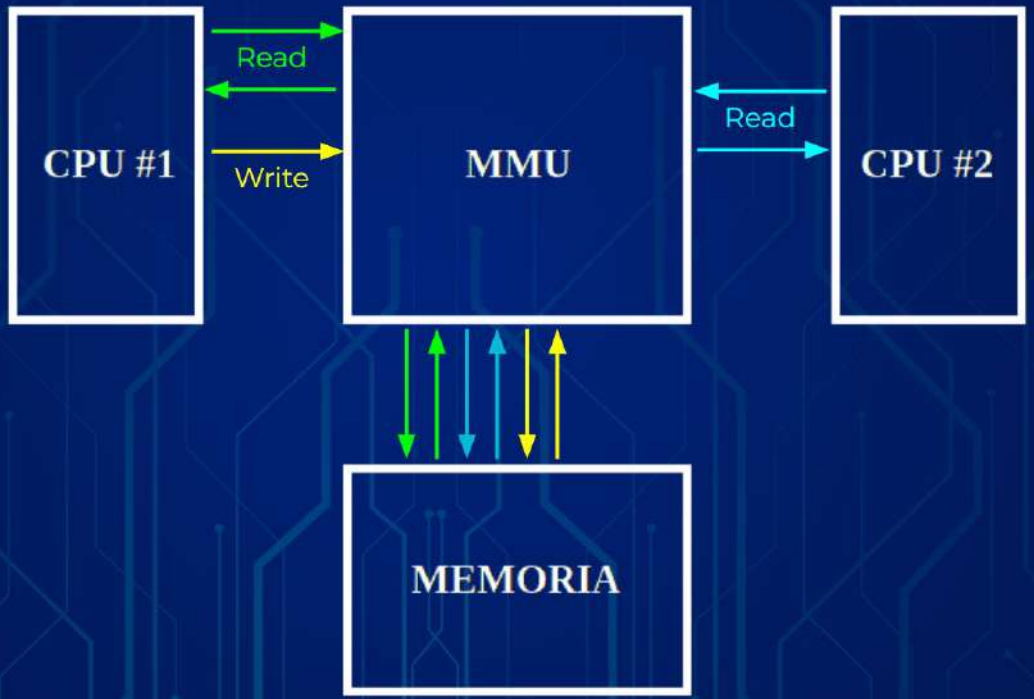
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



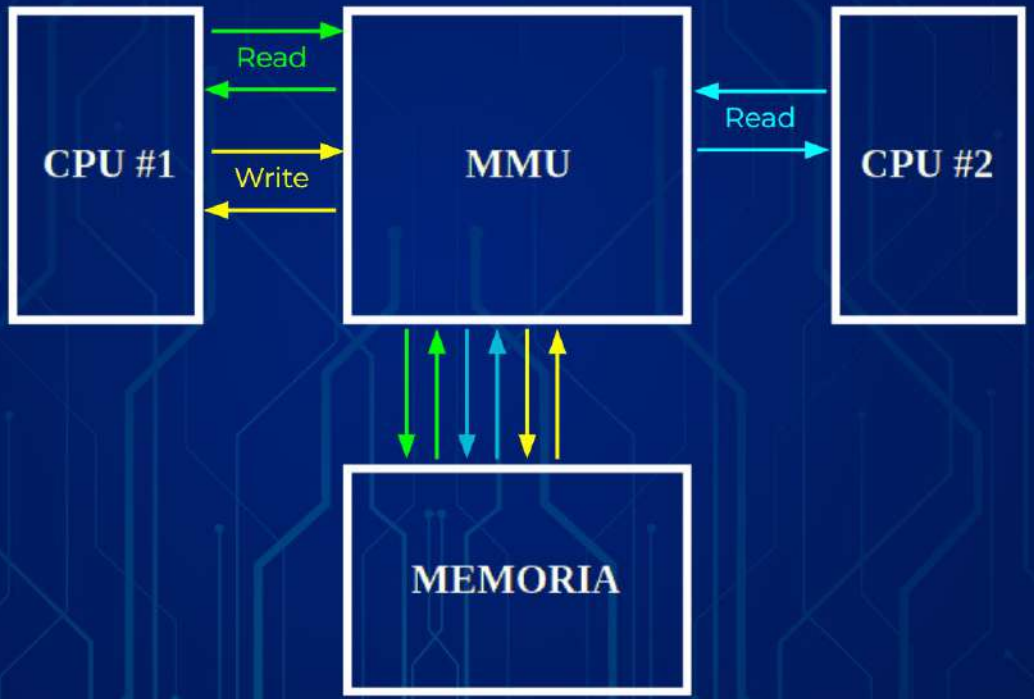
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



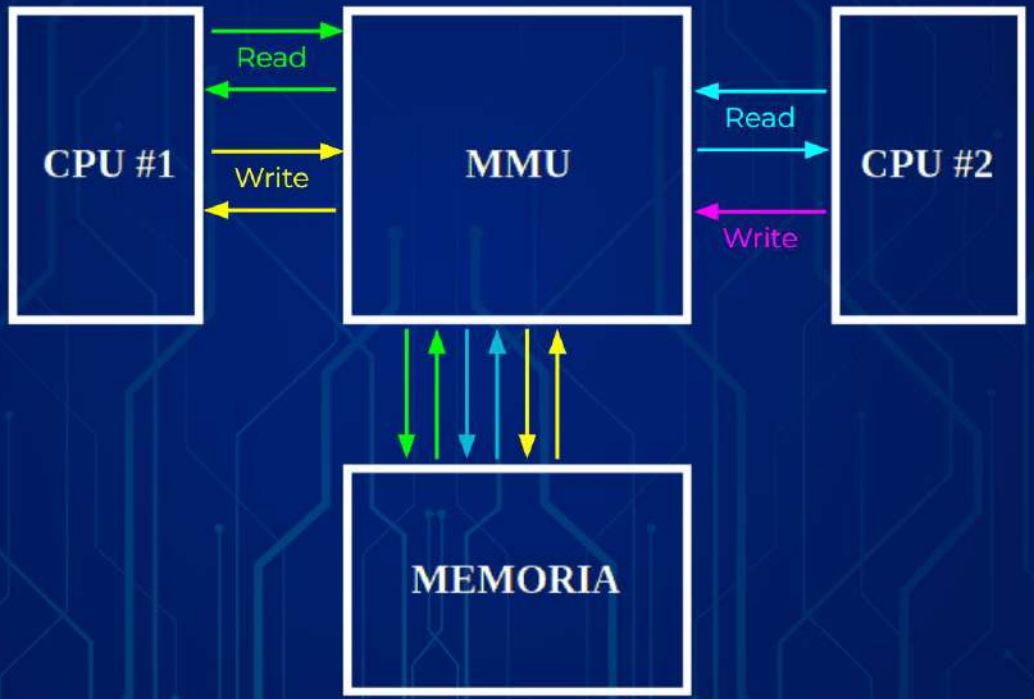
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



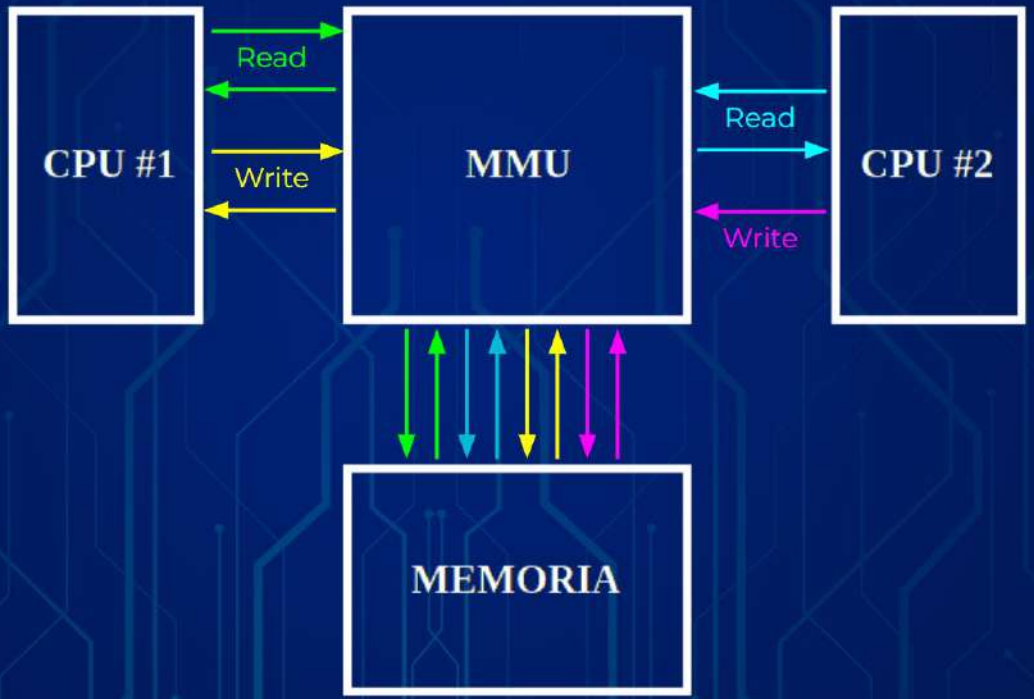
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



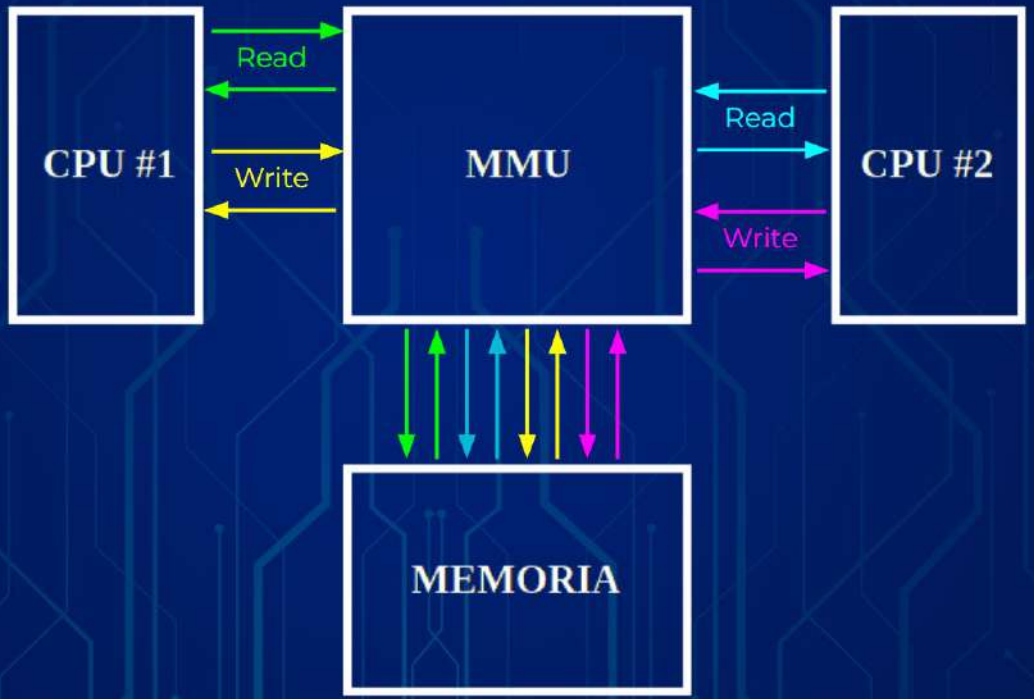
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



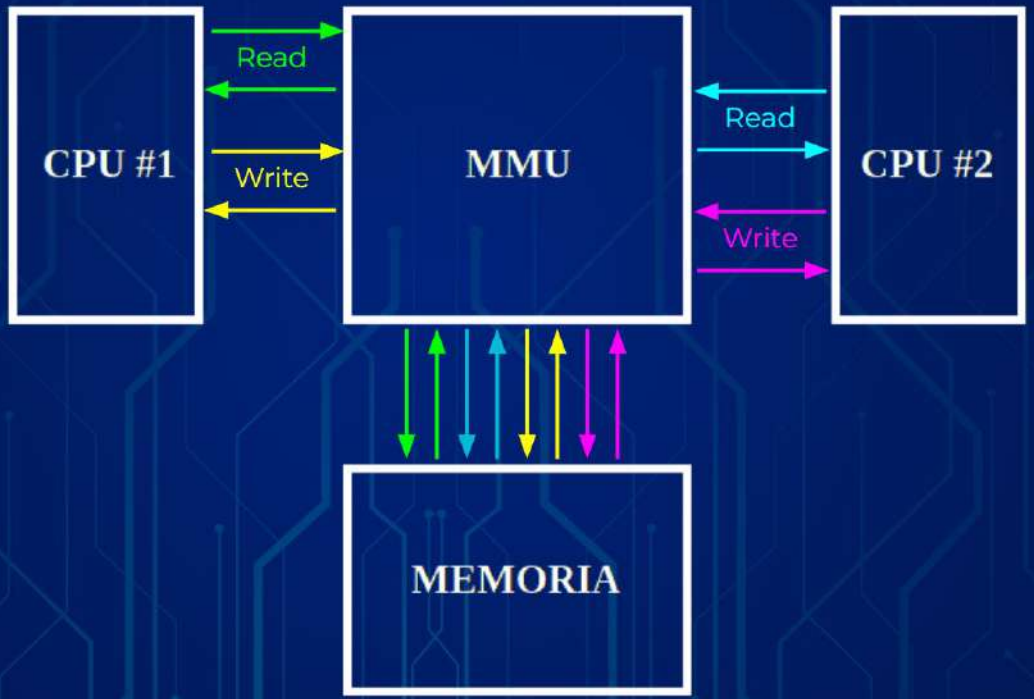
Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



Ambientes multiproceso y/o multihilo - Variables atómicas (4/5)

Cuando hay varios procesadores, la MMU controla los accesos a memoria.
Veamos qué pasa si los dos XCHG se ejecutaran simultáneamente en dos CPUs:



¡ Los dos threads, uno en cada CPU, se quedan con el recurso a la vez !

Ambientes multiproceso y/o multihilo - Variables atómicas (5/5)

Podemos pensar en la siguiente alternativa:

```
            MOV       RAX, 1          ; Levantar la variable Flag y ponerla en 1
            LOCK XCHG RAX,[Flag]      ; si está en 0.
            TEST      RAX,RAX         ; Si era 1, esperar que el recurso se libere
            JNE       Ocupado
            <Usando recurso>
            <Usando recurso>
            XOR       RAX,RAX         ; Indicar que está libre
            XCHG      RAX, [Flag]
Ocupado     …
```

Ahora, el prefijo LOCK bloquea el bus de datos durante toda la ejecución de la instrucción.

OJO! Para el XCHG es innecesario, el procesador lo pone por nosotros.

Se puede usar para otras instrucciones *Read-Modify-Write* como INC (Increment), BTS (Bit Test and Set), BTR (Bit Test and Reset), BTC (Bit Test and Complement), etc.

Soporte p/multiprocesamiento y/o multihilo - SpinLocks

El SO ofrece varios mecanismos para sincronizar el trabajo de tareas e hilos. Uno de ellos es el *SpinLock*.
Un spin lock tiene dos estados (0: rojo y 1: verde). El SO ofrece funciones para cambiar el estado y para esperar que cambie a verde.

Operación	Windows	Linux pthreads
Definir	LONG SL;	pthread_spinlock_t SL;
Inicializar	SL = 0;	pthread_spin_init(&SP, 0):
Destruir	No requiere	pthread_spin_destroy(&SL);
Intentar setear sin bloqueo	si = InterlockedExchange(&SL, 1)== 0;	si = pthread_spin_trylock(&SL)==0;
Intentar setear con bloqueo	No hay!	pthread_spin_lock(&SL);
Liberar	InterlockedExchange(&SL, 0);	pthread_spin_unlock(&SL);

Poco usado, es un caso particular de otros mecanismos

Soporte p/multiprocesamiento y/o multihilo - Semáforos (1/2)

Los semáforos son similares a los SpinLocks, pero en vez de valer 0 o 1 pueden tener cualquier valor positivo. Decimos que el semáforo está en rojo (*non-signaled*) cuando vale 0 y en verde (*signaled*) cuando es ≥ 1.

Operación	Windows	Linux pthreads
Definir	HANDLE S;	sem_t S;
Inicializar	S = CreateSemaphore(NULL, 0, 10, “name”); S = OpenSemaphore(SEMAPHORE_ALL_ACCESS, false, “name”)	sem_init(&S, false, 0); SP = sem_open(“name”, O_CREAT);
Destruir	CloseHandle(S);	sem_destroy(&S);
Ver estado sin bloquear	NtQuerySemaphore(&S, SemaphoreBasicInformation, &V, sizeof(V), NULL);	sem_getvalue(&S, &V);
Esperar ≠ 0 con bloqueo	WaitForSingleObject(S, INFINITE);	sem_wait(&S);
Incrementar	ReleaseSemaphore(S, 1, &Dummy);	sem_post(&S);

Se usa para sincronizar hilos, pero se puede usar para procesos

Soporte p/multiprocesamiento y/o multihilo - Semáforos (2/2)

Los semáforos son útiles para implementar *thread pools*.

Los *thread pools* son conjuntos de threads que llevan a cabo una determinada tarea, por ejemplo devolver una página web. Cada vez que hay algo para hacer (llega un pedido al servidor HTTP), se encola la misma e incrementa en uno el semáforo. Cada uno de los threads está esperando el semáforo, por lo tanto cada vez que se encola una tarea un (y solo un) thread se destraba para servirla.

Soporte p/multiprocesamiento y/o multihilo - Mutex (1/2)

Los Mutex son estructuras que un thread puede poseer y luego liberar. Sólo un thread puede poseer el Mutex en determinado momento, si dos lo requieren, sólo uno lo obtiene y el otro queda en espera.

Operación	Windows	Linux pthreads
Definir	HANDLE M:	pthread_mutex_t M;
Inicializar	M = CreateMutex(NULL, FALSE, “name”);	pthread_mutex_init(&M, NULL);
Destruir	CloseHandle(M);	pthread_mutex_destroy(&M);
Pedir	WaitForSingleObject(M, INFINITE);	pthread_mutex_lock(&M);
Devolver	ReleaseMutex(M);	pthread_mutex_unlock(&M);

Se usa para sincronizar hilos, pero se puede usar para procesos

Soporte p/multiprocesamiento y/o multihilo - Mutex (2/2)

Imaginemos que un thread encola y otro desencola.

Si ambas alteran los punteros involucrados los resultados son indeterminados.

Una forma de evitar el conflicto es crear un Mutex. Cada vez que alguno de los threads involucrados desea alterar la estructura de punteros de la cola (sea para agregar o sacar elementos) debe pedir y obtener primero el Mutex.

De esta forma se garantiza que sólo un thread está modificando los punteros en cada instante. Ésto es así aún si dos de esos threads corren simultáneamente en dos núcleos de procesador multi-núcleo.

Soporte p/multiprocesamiento y/o multihilo - Pipes (1/3)

Los *pipes* son un mecanismo de comunicación entre procesos. Se llaman así por la palabra inglesa *pipe* que significa *tubería*. Al crear un pipe siempre tiene dos extremos. Los mensajes que se introducen por un extremo salen por el otro en el mismo orden en que entraron. Pueden ser bidireccionales.

Operación	Windows	Linux
Definir	HANDLE P;	Int P;
Crear	P = CreateNamedPipe("nam", …);	P = mkfifo("name", …);
Abrir	P = CreateFile("name", FILE_SHARE_READ | FILE_SHARE_WRITE, ...);	P = open("name", O_WRONLY|O_RDONLY, …)
Enviar Mensaje	WriteFile(P, Buffer, BufferLen, …);	write(P, Buffer, BufferLen);
Recibir Mensaje	ReadFile(P, Buffer, sizeof(Buffer), …);	pthread_mutex_lock(&M);

Se usa principalmente para comunicar procesos

Soporte p/multiprocesamiento y/o multihilo - Pipes (2/3)

Si bien los pipes están pensados para conectar solo dos procesos (uno de cada lado), es posible conectar más de dos.

Sin embargo es raramente usado, dado que lo escrito por un proceso no llega a todos, sino que el primero que lee recibe los datos y los demás se los pierden.

Tampoco hay forma, si hay más de un proceso que escribe datos, de saber cuál de los procesos envió la información que se lee.

Soporte p/multiprocesamiento y/o multihilo - Pipes (3/3)

Como se vio en la práctica, al crear un proceso de consola éste recibe tres file handles abiertos:

stdin	Recibe la entrada del teclado
stdout	Lo que se escribe aparece en la pantalla como salida standard.
stderr	Lo que se escribe también aparece en la pantalla, pero diferenciado como error.

Sin embargo, al ejecutar un comando como:

```
ls | grep "a.txt"
```

ni la salida standard (stdout) del programa *ls* es la consola ni la entrada standard (stdin) de programa *grep* es el teclado. Lo que hace el shell es crear un *pipe* unidireccional y colocar un extremo como stdout del programa *ls* y el otro como stdin del programa *grep*.
De esta forma, toda la salida generada por el primer programa (*ls*) es procesada como entrada del segundo programa (*grep*).

Soporte p/multiprocesamiento y/o multihilo - Signals

Los programas bajo Linux (y hasta cierto punto bajo Windows) reciben notificaciones de ciertos eventos mediante una función callback. Esta función es declarada por el programa al SO y este último lo usa para pasarle notificaciones.

En el caso de Windows estas notificaciones son limitadas, permiten detectar el CTRL-C, CTRL-Break, LogOff y Shutdown.

En el caso de Linux son más de 30, pero lo más importante es que no solamente las envía el SO. Cualquier tarea con el debido privilegio puede enviar señales a otro proceso para notificarlo. La señal SIGUSR1 es un caso típico que es interpretada distinto por distintos programas dado que es de propósitos generales.

Bajo Linux las señales pueden ser enviadas, inclusive, desde la consola:

```
kill -USR1 <PID>
```

Fin

¿Preguntas?