JUAN SEBASTIÁN GARIZAO PUERTO - PRACTICAL SESSION

1. **Theory**
    a. La arquitectura de Flynn está compuesta por 4 clasificaciones:
        i. SISD (Single Instruction, Single Data): una sola secuencia de instrucciones operando sobre un solo flujo de datos.
        ii. SIMD (Single Instruction, Multiple Data): una instrucción se transmite a muchas unidades ALUs para operar en datos distintos en paralelo.
        iii. MIMD (Multiple Instruction, Multiple Data): es el arreglo más popular, múltiples CPUs, cada una con su propio flujo de instrucciones y datos.
        iv. MISD (Multiple Instruction, Single Data): varios flujos de instrucciones aplicados al mismo flujo de datos.
        v. La arquitectura MISD ha sido considerada insensata tradicionalmente debido a que no aporta ventajas significativas a la hora de usarlo. En cambio, genera redundancia y chequeos fallidos.
    b. Las limitaciones de Instruction Level Parallelism son causadas porque hay dependencias de datos, pues una instrucción necesita del resultado de otra; existen múltiples cuellos de botellas en el CPU, Datapath o en memoria. Eso sí, han intentado salir adelante, aprovechándose de los múltiples transistores y unidades por ciclo, sin embargo, no todo resulta posible de paralelizar.
    c. La necesidad de aumentar el rendimiento ha hecho que los programadores tengan que diseñar software que explote el paralelismo explícitamente, pues ya no les basta solo con aumentar la frecuencia del reloj, sino que tienen que implementar más núcleos y unidades paralelas. Las tendencias más relevantes de Micro y Macro Arquitectura han sido: la brecha entre rendimiento pico y sostenible, los cuellos de botellas del sistema de memoria y las diferencias entre las arquitecturas UMA y NUMA. Los retos que se enfrentan al exponer estas dos corrientes son: la sincronización, la correcta partición de tareas y datos y la gestión de localidad de los datos.
    d. Hyperthreading es una técnica que permite a un núcleo físico ejecutar múltiples hilos de modo que se aprovechan mejor las unidades funcionales cuando un hilo se bloquea esperando datos. Para poder sacarle ventaja al uso de esta técnica los programadores deben diseñar aplicaciones multihilo de forma que las tareas queden bien divididas y evitar la contención en memoria compartida.

2. **Scripting languages**
    1.
        1.1. Pregunta: Explain the difference between the output observed on the terminal and that contained in the target piped file.
        1.2. La diferencia en las ejecuciones es que mientras que cuando ejecutamos por terminal se imprime primero “Line 1..” y luego “Line 2”, cuando ejecutamos por pipe, se imprime al revés. En otras palabras, la diferencia entre ejecuciones es el orden cómo se imprimen las cosas. Esto ocurre porque printf usa buffered output, lo cual hace que se guarde en un buffer temporal antes de ser escrito en el terminal. En cambio, la función write no tiene buffered output, lo que hace que escriba directa e inmediatamente.

2.

2.1. Pregunta: Explain what has happened with the addition of the fflush system call.

2.2. La adición de la llamada fflush hace que el contenido almacenado en el buffer de printf se vacíe inmediatamente hacia la salida antes de que se ejecute la función write. Sin este comando, al redirigir la salida a un archivo, el texto de printf podía quedarse en el buffer y no mostrarse en el archivo, generando un cambio en el orden de impresión. Con fflush, se asegura que “Line 1 ..” se escriba en el destino de salida justo antes de “Line 2”, manteniendo así el orden esperado tanto en la terminal como en un archivo.

3.

3.1. Pregunta: Improve and run the following 'C' program several times interactively. Note the different execution order on different runs, Why?

3.2. Al ejecutar el programa varias veces, el orden de las impresiones cambia porque cada llamada a fork crea un proceso nuevo que continúa ejecutando el mismo bucle sin ninguna sincronización entre padres e hijos. Como no hay wait en el padre, el planificador del SO decide cuándo corre cada proceso. Además, como los hijos también siguen iterando el for y vuelven a llamar a fork(), se genera un árbol de procesos que aumenta el grado de concurrencia y por tanto, la variabilidad del orden.

4.

4.1. Pregunta: Explain how and why the order of the output from this program is different from that of the after program.

4.2. En el programa con wait(), el padre se bloquea en cada iteración hasta que su hijo termina. Eso sincroniza parcialmente la salida: para cada i, se ve primero child i y luego parent i. En el programa anterior (sin wait()), no hay sincronización, así que el planificador del SO intercalaba padre e hijo de forma no determinista, variando el orden en cada ejecución. Aquí, en cambio, wait() impone un orden por iteración (hijo → padre), reduciendo la aleatoriedad observada.

5.

5.1. Pregunta: Have you reached parallel code? Or parallel behavior?

5.2. Con fork generamos múltiples procesos que el SO puede ejecutar al mismo tiempo si hay varios núcleos. Por eso, hemos alcanzado comportamiento paralelo y, potencialmente, ejecución paralela real en hardware multinúcleo. En una máquina de un solo núcleo, el SO hace time-slicing y obtenemos concurrencia más que paralelismo, pero el código ya expresa concurrencia y permite paralelismo cuando el hardware lo posibilita.

6.

6.1. Pregunta: Does the shell-script code perform parallelism or pipelining?

6.2. Los scripts encrypt_it.sh/decrypt_it.sh tal como los usamos no hacen paralelismo por sí mismos, ejecutan pasos secuenciales.

7.

7.1. Pregunta: But if script-based parallel programming is so easy, why bother with anything else?

7.2. Porque los scripts tienen límites claros, mayor sobrecarga de procesos frente a hilos, poca granularidad de control, sincronización limitada, gestión de datos sin memoria compartida ni afinidad, menor portabilidad de rendimiento y difícil depuración en escenarios intensivos.

8.

8.1. Pregunta: Child processes run in parallel mode? Or concurrently? Why?

8.2. Los procesos hijo creados con fork pueden correr en paralelo real si el hardware dispone de varios núcleos y el SO los agenda simultáneamente. En una máquina de un solo núcleo se corre con concurrencia. En ambos casos el código se ejecuta concurrentemente, si hay recursos suficientes, el planificador la ejecuta como paralelismo.

9.

9.1. Pregunta: Is the running time equal for sum and adder programs? Which is faster? Why?

9.2. No, el tiempo de ejecución no es igual. sum usa un solo proceso y recorre todo el rango/lista; adder reparte el trabajo entre $k \approx n/3$ procesos. Sin embargo, adder incurre en sobrecoste de creación de procesos y comunicación por pipes. Esto implica que para listas pequeñas ese tiempo extra puede hacer a sum más rápido, mientras que para listas grandes y con varios núcleos, adder suele ganar por al ejecutar de modo paralelo.

10.

10.1. Do parent and child processes use shared memory? Is it important?

10.2. Tras fork, padre e hijo no comparten la memoria del proceso. Comparten algunos descriptores de archivo heredados , pero no una RAM común salvo que se use explícitamente memoria compartida . Esto es importante porque obliga a usar IPC para comunicar resultados.

11.

11.1. Pregunta: Which implementation is faster? Why?

11.2. La versión secuencial resultó tan rápida como la de hilos e incluso en algunas corridas fue más veloz. Esto se debe a que la parte más costosa es la multiplicación de matrices, que sigue ejecutándose en el hilo principal en ambos programas.

12. .

12.1. Pregunta: How do threads work? Always threads implementations are faster?

12.2. Los hilos funcionan dentro del mismo proceso compartiendo memoria, y permiten que distintas partes del programa se ejecuten de forma concurrente o en paralelo, dependiendo del número de núcleos de la máquina. Sin embargo, no siempre son más rápidos. Si lo que se va a paralelizar es pequeño, si hay demasiado tiempo añadido al crear y

sincronizar hilos, o si el problema depende más de entrada/salida que de CPU, la versión con hilos puede ser más lenta que la secuencial.

13.
    13.1. Pregunta: Threads and processes share the same region of memory? What memory segments share threads?
    13.2. Los hilos sí comparten el mismo espacio de direcciones del proceso. Eso significa que acceden al mismo código, a los datos globales y al heap. Lo único que no comparten es la pila, porque cada hilo necesita su propio stack para manejar sus variables locales y su ejecución.
14.
    14.1. Pregunta: Can any programming challenge be implemented using threads?
    14.2. No todos los problemas se pueden o se deben resolver con hilos. Hay cálculos que son completamente secuenciales, y otros donde el riesgo de condiciones de carrera o el costo de sincronización supera los beneficios.
15.
    15.1. Pregunta: Your strategy works when the quantity of producer instances increases?
    15.2. Sí. Cuando aumento la cantidad de productores, el buffer se llena más rápido y los consumidores pasan a ser el cuello de botella. En mi implementación, cada productor toma el primer 0, escribe un valor y luego suelta el mutex; si el buffer queda lleno, el productor se retira y notifico a los consumidores. Funciona correctamente porque el acceso al arreglo está protegido con mutex y la disponibilidad de ítems se comunica con variable de condición.
16.
    16.1. Pregunta: Your strategy works when the quantity of consumers instances increases?
    16.2. También funciona al aumentar consumidores, el arreglo tiende a vaciarse más rápido. Cuando no hay items y todavía quedan productores vivos, los consumidores esperan en not_empty. Cuando todos los productores terminan y ya no quedan valores ≠ 0, los consumidores salen de forma ordenada.
17.
    17.1. Pregunta: What occurs when you use locked vars? Semaphores?
    17.2. Con locked vars garantizo exclusión mutua al recorrer y modificar el arreglo, con condition variables resuelvo la espera bloqueante.