**Memoria Interna**

La memoria interna de un ordenador está formada por los componentes de almacenamiento más rápidos y cercanos al procesador, como los registros de la CPU y la memoria caché, estos niveles intermedios, que incluyen cachés L1, L2 y L3, permiten acceder a los datos en fracciones de nanosegundo, garantizando que el procesador no se detenga esperando información.[1]

En esta categoría también se encuentra la memoria principal, normalmente DRAM, que aunque es más lenta que la caché, sigue siendo considerablemente más veloz que la memoria externa, su función es almacenar temporalmente los programas y datos que están siendo utilizados, manteniendo un equilibrio entre velocidad y capacidad.[1]

La jerarquía de la memoria interna busca reducir el tiempo de acceso mediante el uso de diferentes niveles con velocidades y tamaños distintos.[1]

A medida que se avanza hacia la CPU, el almacenamiento es más rápido, pero más costoso y de menor capacidad, mientras que hacia abajo se sacrifica velocidad a cambio de espacio.[1]

La memoria interna es la parte principal donde se realiza toda la computación. En general, los sistemas operativos no permiten controlar directamente el movimiento de bloques de datos en la memoria principal, lo que dificulta que los programadores implementen eficientemente algoritmos que minimicen las operaciones de E/S.[3]

TPIE ofrece una gestión automática de la memoria interna, controlando la cantidad de memoria utilizada y distribuyéndola entre los diferentes componentes de un programa de forma eficiente. Esto permite a los desarrolladores no preocuparse por el manejo manual de memoria, a diferencia de STXXL, donde el programador debe asignar memoria explícitamente a cada parte.[3]

Además, TPIE soporta la ejecución paralela en CPUs multinúcleo, lo que mejora el rendimiento en tareas complejas como la ordenación multi-fusión. Esta capacidad facilita que distintas partes del algoritmo se ejecuten simultáneamente y aprovechen mejor los recursos disponibles.[3]

**Memoria Interna en Linux**

La memoria interna en Linux corresponde principalmente a la memoria RAM disponible para los procesos y al espacio usado para optimizar el acceso a datos mediante cachés y buffers. Estos elementos permiten que el sistema reduzca la frecuencia de accesos al disco, mejorando así el rendimiento general y la velocidad de respuesta.[2]

Dentro de esta memoria, la porción denominada *MemFree* indica la cantidad de RAM libre que el sistema tiene disponible en un momento dado. Aunque este valor puede parecer bajo, Linux aprovecha la RAM no utilizada para almacenamiento temporal de datos en caché, liberándola cuando un proceso la necesita.[2]

La *memoria en búfer* (*buffers*) se emplea para mantener temporalmente información relacionada con operaciones de escritura y lectura en dispositivos de almacenamiento. Por otro lado, la *memoria caché* (*cached*) guarda datos y programas recientemente utilizados, permitiendo que su acceso sea mucho más rápido que volver a leerlos desde el disco.[2]

En algunas arquitecturas, Linux distingue entre *memoria alta* (*High Memory*) y *memoria baja* (*Low Memory*). La memoria alta es aquella que el kernel accede de forma indirecta y que no tiene un mapeo permanente, mientras que la memoria baja sí está mapeada de manera directa, lo que facilita el acceso a los procesos.[2]

Además, el sistema gestiona otras áreas internas como *Slab Memory*, utilizada por la caché del kernel para almacenar estructuras de datos internas; *KernelStack*, dedicada a las pilas de ejecución del núcleo; y *PageTables*, que contienen las tablas de páginas para la traducción de direcciones de memoria virtual a física. Todas estas partes trabajan en conjunto para asegurar un uso eficiente y ordenado de la memoria interna.[2]

La memoria interna es la parte principal donde se realiza toda la computación. En general, los sistemas operativos no permiten controlar directamente el movimiento de bloques de datos en la memoria principal, lo que dificulta que los programadores implementen eficientemente algoritmos que minimicen las operaciones de E/S.[3]

TPIE ofrece una gestión automática de la memoria interna, controlando la cantidad de memoria utilizada y distribuyéndola entre los diferentes componentes de un programa de forma eficiente. Esto permite a los desarrolladores no preocuparse por el manejo manual de memoria, a diferencia de STXXL, donde el programador debe asignar memoria explícitamente a cada parte.[3]

Además, TPIE soporta la ejecución paralela en CPUs multinúcleo, lo que mejora el rendimiento en tareas complejas como la ordenación multi-fusión. Esta capacidad facilita que distintas partes del algoritmo se ejecuten simultáneamente y aprovechen mejor los recursos disponibles.[3]

**Memoria Externa**

La memoria externa hace referencia a dispositivos de almacenamiento de gran capacidad pero menor velocidad en comparación con la memoria interna.[1]

Ejemplos comunes incluyen los discos duros magnéticos, las unidades ópticas y las cintas, empleadas principalmente para archivo y respaldo de información.[1]

En estos medios, el acceso a los datos puede tardar varios milisegundos debido al movimiento físico de los cabezales y a la latencia de rotación, lo que los hace millones de veces más lentos que la caché o los registros del procesador. Por esta razón, se transfieren bloques grandes de datos de forma secuencial para optimizar el tiempo de espera.[1]

Los sistemas de almacenamiento modernos, como RAID, permiten acceder a varios discos en paralelo para aumentar el ancho de banda y reducir la brecha de velocidad frente a la memoria interna. Aun así, la diferencia de rendimiento persiste, por lo que los algoritmos deben aprovechar la localidad de referencia para mejorar la eficiencia de las operaciones de entrada y salida.[1]

La memoria externa, como los discos, se utiliza para almacenar grandes volúmenes de datos que no caben en la memoria principal, el traslado de información entre la memoria

interna y el disco suele ser el factor que más limita el rendimiento cuando se trabaja con cantidades masivas de datos.[3]

El acceso a disco es mucho más lento que el acceso a la memoria interna, por lo que los datos se transfieren en bloques contiguos grandes para optimizar las operaciones de entrada y salida (E/S), los algoritmos en este contexto miden su eficiencia principalmente por la cantidad de operaciones de E/S que requieren para procesar N elementos.[3]

Para hacer frente a estas limitaciones, se han desarrollado algoritmos especialmente diseñados para minimizar la cantidad de accesos al disco. El modelo I/O, creado por Aggarwal y Vitter, ayuda a analizar estos algoritmos considerando una memoria interna limitada y una memoria externa de gran tamaño donde se deben manejar los datos.[3]

Además, existen bibliotecas en C++ como TPIE y STXXL que facilitan la implementación de algoritmos eficientes en E/S. Mientras STXXL expone al programador detalles del hardware para maximizar el rendimiento, TPIE busca simplificar la programación ocultando estos detalles y automatizando la gestión de memoria y E/S.[3]

**Memoria externa en (DNC)**

La memoria externa en un Differentiable Neural Computer (DNC) es una matriz que permite a la red almacenar y procesar secuencias complejas de datos. El DNC usa cabezas de lectura y escritura para interactuar con esta memoria de forma diferenciable, lo que facilita su entrenamiento mediante descenso de gradiente.[4]

Para mejorar el uso de la memoria, el DNC implementa mecanismos como la vinculación temporal, que mantiene el orden de escritura, y la asignación dinámica, que reutiliza posiciones libres. Durante la escritura y lectura, combina atención basada en contenido con estas técnicas para acceder eficazmente a la información.[4]

Se propuso una mejora que divide la memoria en pares clave-valor, reduciendo el tiempo de cómputo y aumentando la cantidad de información capturada. Otra mejora usa una red neuronal para la lectura, que mejora la precisión pero aumenta el tiempo de ejecución, por lo que su combinación con la memoria clave-valor no siempre es conveniente.[4]

## BIBLIOGRAFÍA:


[1] J. S. Vitter, “Algorithms and data structures for external memory,” *Foundations and Trends in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2006, doi: 10.1561/0400000014.

[2] D. Kayande and U. Shrawankar, “Performance Analysis for Improved RAM Utilization for Android Applications.”

[3] L. Arge, M. Rav, S. C. Svendsen, and J. Truelsen, “External memory pipelining made easy with TPIE,” in *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, Dec. 2017, pp. 319–324. doi: 10.1109/BigData.2017.8257940.

[4] A. Yadav and K. Pasupa, “Augmenting Differentiable Neural Computer with Read Network and Key-Value Memory,” in *2021 25th International Computer Science and Engineering Conference (ICSEC)*, IEEE, Nov. 2021, pp. 262–266. doi: 10.1109/ICSEC53205.2021.9684629.

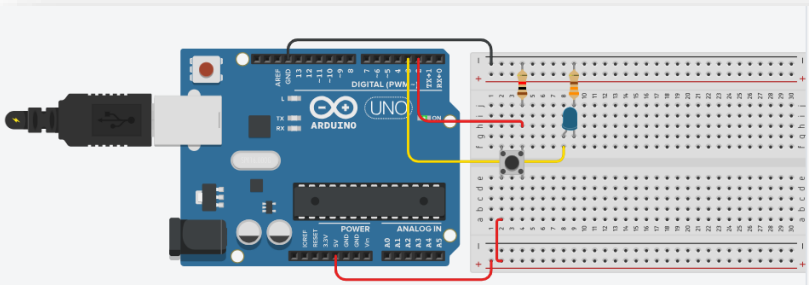

### Procedimiento

Operación de Entrada/Salida:

• Implementar ejemplos de E/S programada y E/S mediante interrupciones usando simuladores.

• Analizar el flujo de datos mediante DMA con un ejemplo práctico

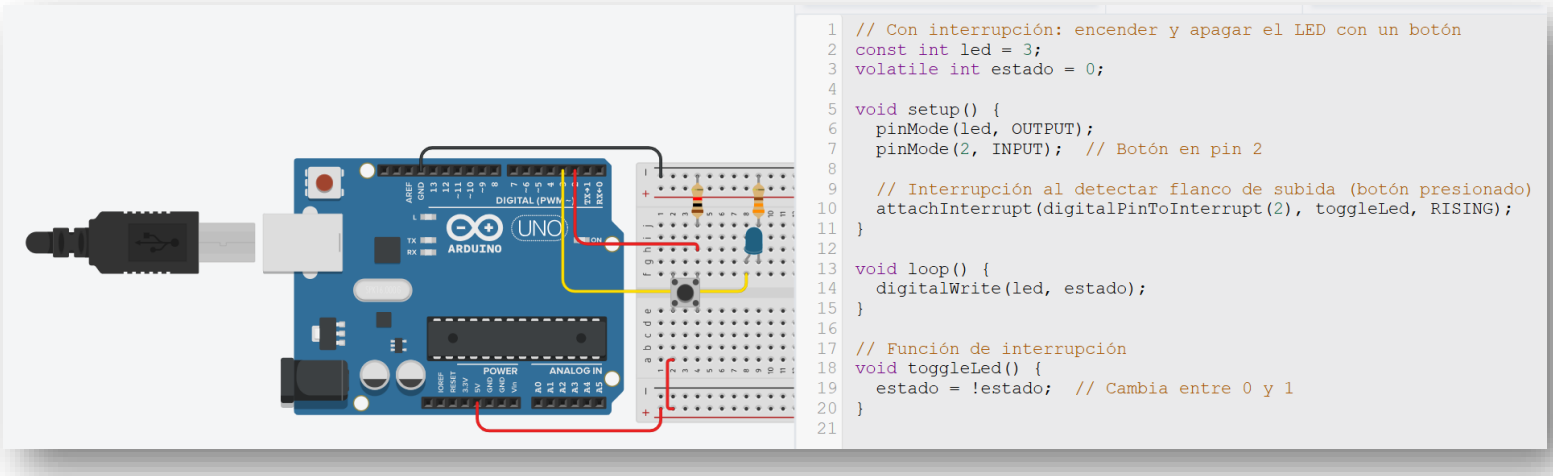
**E/S Programada**




```
//Con un solo pulso encender el LED con otro pulso apagar el LED

const int pul = 2; //Configuracion Pull Down
const int led = 3;
int i = 0;       //variable i inicializada con un valor inicial de0

void setup()
{
  pinMode(pul, INPUT);//definimos el pin 2 como entrada
  pinMode(led, OUTPUT);//definimos el pin 3 como salida
}

void loop()
{
  if(digitalRead(pul)==HIGH)
  {
    delay(400);
    i=1-i;
    if(i == 1)
    {
      digitalWrite(led,HIGH);
    }
    else
    {
      digitalWrite(led,LOW);
    }
  }
}
```

*Figura 1: Implementación de E/S programada para encender y apagar un LED con cada pulsación de un botón usando digitalRead en el ciclo loop().*

## E/S Con Interrupciones



```arduino
// Con interrupción: encender y apagar el LED con un botón
const int led = 3;
volatile int estado = 0;

void setup() {
  pinMode(led, OUTPUT);
  pinMode(2, INPUT);  // Botón en pin 2

  // Interrupción al detectar flanco de subida (botón presionado)
  attachInterrupt(digitalPinToInterrupt(2), toggleLed, RISING);
}

void loop() {
  digitalWrite(led, estado);
}

// Función de interrupción
void toggleLed() {
  estado = !estado;  // Cambia entre 0 y 1
}
```

*Figura 2 : Uso de interrupciones para controlar un LED con un botón, permitiendo una respuesta inmediata al detectar un flanco de subida.*

## Flujo de datos mediante DMA

```csharp
using System;
using System.Threading;

0 referencias
class Program
{

    static int[] sourceData = new int[100];
    static int[] destinationData = new int[100];
    static bool dmaCompleted = false;

    0 referencias
    static void Main(string[] args)
    {

        for (int i = 0; i < sourceData.Length; i++)
            sourceData[i] = i + 1;


        Thread dmaThread = new Thread(() => DmaTransfer(0, 0, sourceData.Length));
        dmaThread.Start();


        while (!dmaCompleted)
        {
            Console.WriteLine("CPU haciendo otras tareas...");
            Thread.Sleep(200);
        }


        Console.WriteLine("DMA terminó la transferencia. Datos en destino:");
        for (int i = 0; i < destinationData.Length; i++)
            Console.Write(destinationData[i] + " ");

        Console.WriteLine("\nProceso finalizado.");
    }


    1 referencia
    static void DmaTransfer(int sourceIndex, int destIndex, int size)
    {
        Console.WriteLine("DMA iniciando transferencia...");
        for (int i = 0; i < size; i++)
        {
            destinationData[destIndex + i] = sourceData[sourceIndex + i];
            Thread.Sleep(50);
        }
        dmaCompleted = true;
        Console.WriteLine("DMA transferencia completada.");
    }
}
```

*Figura 3: Simulación realizada en C# aplicación de consola.*

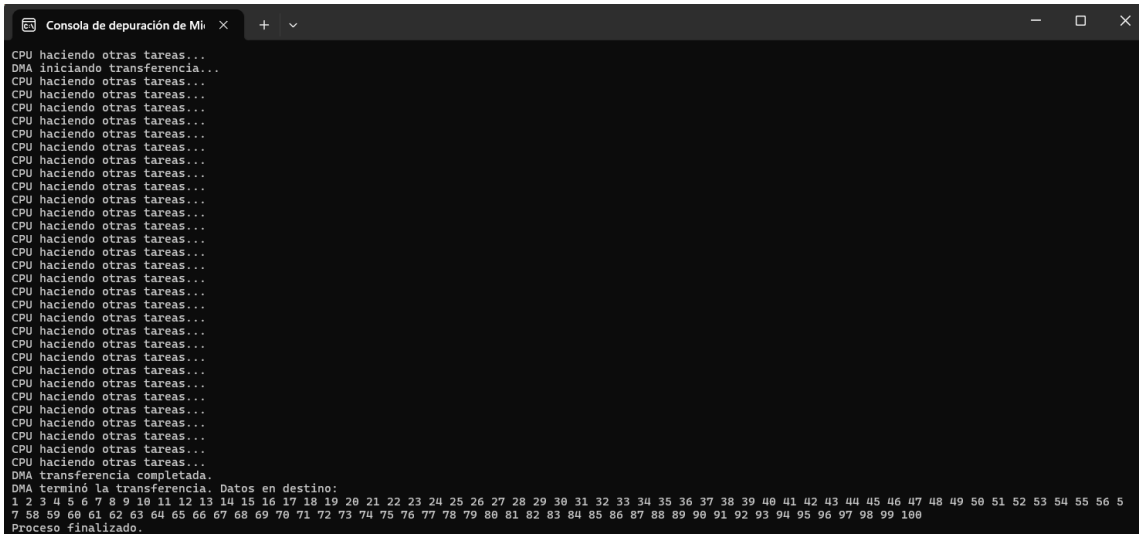


*Figura 4: Ejecución de la simulación del programa.*

En la figura 3 y 4 se presenta la simulación del funcionamiento del DMA (acceso directo a memoria) usando C#. Se crean dos arreglos: uno con los datos fuente y otro vacío como destino. Se utilizó un hilo separado para representar al DMA, que copia los datos de forma paralela mientras la CPU continúa ejecutando otras tareas, esto se refleja en la consola con el mensaje “CPU haciendo otras tareas...”. Cuando el DMA termina la transferencia, se imprime el mensaje de finalización junto con los datos copiados, este ejemplo demuestra cómo el DMA permite liberar a la CPU de tareas de transferencia, mejorando la eficiencia del sistema.