Parcial 1 - Sistemas Operativos

Manejo de Grandes Volúmenes de Datos en Linux con C/C++

2025B

Objetivo

Desarrollar un programa en C/C++ (Linux) que procese grandes volúmenes de datos generados por el Generador de Datos, midiendo tiempo/memoria y comparando:

- Valores vs. apuntadores
- Struct (C) vs. Class (C++)

Datos Clave

Cada registro incluye (como mínimo):

- Nombre completo
- Fecha de nacimiento (para cálculo de edad)
- Lugar de residencia (ciudad en Colombia)
- Patrimonio - Deudas
- Documento de identidad
- Calendario de declaración de renta (A/B/C):
    - Asignado según los dos últimos dígitos del documento
    - Siguiendo el Calendario Tributario DIAN 2025
        - `Grupo A`: Dígitos 00-39
        - `Grupo B`: Dígitos 40-79
        - `Grupo C`: Dígitos 80-99

Preguntas Obligatorias

1. 👵 Persona más longeva: laura I clases y por referencia
    - En todo el país
    - Por cada ciudad
2. 💰 Persona con mayor patrimonio: David
    - En el país
    - Por ciudad (Lista)
    - Por grupo de declaración (A/B/C) (Lista)
3. 📅 Declarantes de renta: Laura A
    - Listar y contar personas por calendario (puede usar el calendario Grupo A, Grupo B y Grupo C en lugar del calendarios tributario de la DIAN)
    - Validar asignación según terminación del documento(Lista)

Tres Preguntas Adicionales

1. País, ciudad,grupo con más deudas

2. Grupo con más personas de cada ciudad

3. Ciudades con patrimonio promedio más alto

Requerimientos Técnicos

- Comparaciones críticas:
    - Rendimiento: Valores vs. Apuntadores
    - Eficiencia: Struct (C) vs. Class (C++)
- Métricas obligatorias:
    - Tiempo de ejecución
    - Uso de memoria (RAM)
- Optimización:
    - Algoritmos eficientes para >1 millón de registros

Entrega

- 📦 Qué entregar:

Componente	Formato
Código	C/C++ (Linux)
Análisis	PDF con resultados y métricas
Explicación	Video (5 min máximo)

- ⏰ Plazo: 1 semana (buzón de Interactiva)
- 👥 Equipos:
  - Todos deben subir la misma entrega
  - Nota 0.0 si algún miembro no publica

Rúbrica Simplificada

Aspecto	Puntos	Criterios Clave
Funcionalidad	1.0	Respuestas correctas + Preguntas adicionales relevantes
Eficiencia	1.5	Análisis valores/apuntadores + Struct/Class + Métricas
Calidad de Código	1.0	Legibilidad + Documentación
Claridad	0.5	Informe y video comprensibles
Pensamiento Crítico	1.0	Profundidad en 4 preguntas técnicas

4 Preguntas de Pensamiento Crítico

1. Memoria: ¿Por qué usar *apuntadores* reduce 75% de memoria con 10M registros?
2. Datos: Si el calendario depende de dígitos del documento, ¿cómo optimizar búsquedas por grupo?
3. Localidad: ¿Cómo afecta el acceso a memoria al usar *array de structs* vs. *vector de clases*?
4. Escalabilidad (Consulta): Si los datos exceden la RAM, ¿cómo usar `mmap` o memoria virtual?

Solución:

1. Usar apuntadores reduce bastante el uso de la memoria debido a que, cuando trabajamos con ellos, no copiamos todo el contenido de los registros, sino que únicamente almacenamos la dirección en memoria donde está cada uno. En cambio, cuando trabajamos con valor, cada vez que pasamos o usamos un registro se crea una copia completa, lo que con millones de datos significa ocupar varias veces más memoria de la necesaria. Por ejemplo, si cada registro ocupa 100 bytes, con 10 millones de registros serían alrededor de 1 GB; al copiarlos por valor ese consumo prácticamente se duplica, mientras que con apuntadores solo se agregan unos pocos bytes por cada dirección (4 u 8 bytes), lo que representa un ahorro cercano al 75%. En este orden de ideas, trabajar con direcciones de memoria es mucho más eficiente cuando se manejan grandes volúmenes de datos, evitando que la memoria se sature rápidamente.

2. Nosotros encontramos 2 principales formas de hacerlo: Una forma es, desde el inicio, asignar a cada persona su grupo como un atributo. Esto permite comparar por

grupo, pero igual obliga a recorrer a todas las personas cada vez que se hace una búsqueda.

Otra estrategia más eficiente es crear un vector de punteros por cada grupo. Así, cuando se busca en un grupo específico, no se revisa el 100% de los registros sino sólo el subconjunto correspondiente, reduciendo de forma significativa el rango de búsqueda. Además, como estos vectores se mantienen ordenados por ID, las búsquedas pueden hacerse incluso en tiempo logarítmico.

3. La diferencia importante en localidad de memoria no está entre un array de structs y un vector de clases, ya que ambos almacenan los elementos de manera contigua en memoria, lo que permite que el procesador lea bloques completos de datos de forma muy eficiente al cargarlos en la caché. Esa contigüidad asegura que, cuando recorres millones de registros secuencialmente, gran parte de los accesos ya esté en caché y no tengas que ir a memoria principal, lo que ahorra bastante tiempo. El problema real aparece cuando en lugar de guardar los objetos directamente se guardan punteros a ellos (por ejemplo, vector<Persona*>), porque en ese caso los punteros sí están contiguos, pero cada objeto puede terminar disperso en cualquier parte del heap. Esa falta de continuidad obliga al procesador a saltar por direcciones de memoria alejadas, generando más fallos de caché y, en consecuencia, retrasos en la transferencia de datos desde la RAM hacia la CPU. En cargas de trabajo con millones de registros, esa diferencia se traduce en un costo de memoria mucho más alto y en tiempos de acceso considerablemente más lentos. Después de este análisis, creemos que, como se dice en el punto anterior, puede que la mejor manera no haya sido la planteada (guardar en 3 vectores diferentes los apuntadores a cada persona dependiendo del grupo), ya que, a pesar que se reducen las personas a visitar, las direcciones de memoria están más dispersas, por tanto tienen más probabilidad de tener errores en el caché, y más carga

para la CPU y la memoria al final de cuentas. Por lo que la respuesta más acertada para el punto anterior sería la primera planteada, agregarle ese atributo a cada persona.

4. En Linux, mmap se utiliza para crear una nueva asignación en el espacio de direcciones virtuales del proceso que realiza la llamada. Asigna archivos o dispositivos a la memoria, lo que permite acceder a los datos como si estuvieran en ella, puede ser más eficiente que las operaciones de E/S tradicionales (como read() y write())para archivos grandes o patrones de acceso secuencial.

mmap() no necesariamente carga todo el archivo en la RAM de inmediato, sino que asigna el archivo a una parte del espacio de direcciones virtuales del proceso. Este espacio es una abstracción proporcionada por el sistema operativo, que mapea la memoria física real a direcciones virtuales accesibles para el proceso. Esto significa que el proceso puede trabajar con datos como si estuvieran en memoria, aunque físicamente puedan residir en disco.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Parámetros

- addr Sugiere la dirección inicial para el mapeo. Si es <nombre del archivo> NULL, el núcleo elige la dirección.
- length:La longitud de la región de memoria a mapear.
- prot: Especifica la protección de memoria deseada (por ejemplo, PROT_READ, PROT_WRITE).

- flags: Determina el tipo de mapeo (por ejemplo, MAP_SHARED, MAP_PRIVATE).
- fd:Descriptor de archivo del archivo a mapear.
- offset:Desplazamiento dentro del archivo donde se inicia el mapeo.

La función establece un mapeo entre el espacio de direcciones de un proceso y un archivo. La región de vuelta comienza en la dirección asignada por el sistema, cubre la longitud indicada y corresponde a un segmento del archivo a partir de un desplazamiento (offset), que es la distancia en bytes desde el inicio del archivo hasta donde comienza el mapeo.

El archivo asociado permanece referenciado mientras exista al menos un mapeo activo, incluso si se cierra el descriptor con close(). El vínculo solo desaparece cuando se libera el mapeo. Si se usa la bandera MAP_FIXED, cualquier asignación previa en ese rango será reemplazada automáticamente.

Restricciones importantes:

- El desplazamiento debe ser cero o un múltiplo del tamaño de página del sistema
- Si el archivo aumenta después del mapeo, las páginas más allá del tamaño original no estarán disponibles.
- Si el archivo se reduce, los accesos más allá del nuevo final producen un comportamiento indefinido.
- Escribir más allá del final del archivo no garantiza que los datos se conserven.
- En algunos sistemas, mmap no puede aplicarse sobre archivos con journaling activo o puede estar limitado por configuraciones de seguridad que impiden ciertos modos de escritura compartida.

Además, es importante comprender que mmap() no solo sirve para mapear archivos, sino también para reservar memoria anónima (sin respaldo en disco) cuando se utiliza la bandera MAP_ANONYMOUS junto con MAP_PRIVATE o MAP_SHARED. En este modo, las páginas asignadas no están ligadas a un archivo y se inicializan en cero, lo que resulta útil para implementar asignadores de memoria personalizados o para crear segmentos de memoria compartida entre procesos.

- **https://www.ibm.com/docs/en/i/7.6.0?topic=ssw_ibm_i_76/apis/mmap.html**
- **https://www.tencentcloud.com/techpedia/106433**
- **https://unix.stackexchange.com/questions/712651/does-mmap-allow-creating-a-mapping-that-is-much-larger-than-the-amount-of-physic**