

Tarea 2

Vicente Hernández

Alex Blanchard

Ignacio Diaz

Vicente Wiederhold

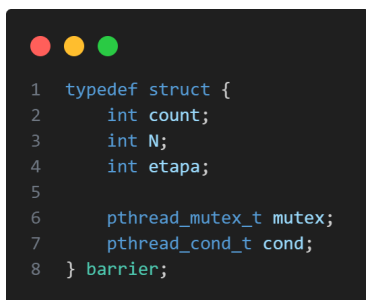
1 de diciembre de 2025

1. Introducción

El siguiente documento detallara la implementación de ambos códigos correspondientes a los enunciados de la tarea 2 del ramos Sistemas Operativos, se analizarán los resultados obtenidos en el segundo enunciado y por ultimo se concluirá y reflexionará sobre el trabajo realizado. Este informe tiene como objetivo reforzar conceptos vistos de manera teórica en las horas de clase y de esta manera poder interiorizarlos mejor. Repositorio Git: <https://github.com/BisenteWiederhold/Tarea2-SO>

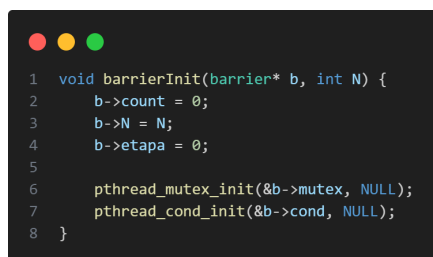
2. Parte I: Sincronización con Barrera Reutilizable

2.1. Actividad 1: Barrera de hebras (Monitor)



```
1 typedef struct {
2     int count;
3     int N;
4     int etapa;
5
6     pthread_mutex_t mutex;
7     pthread_cond_t cond;
8 } barrier;
```

Figura 1: Struct para barrera

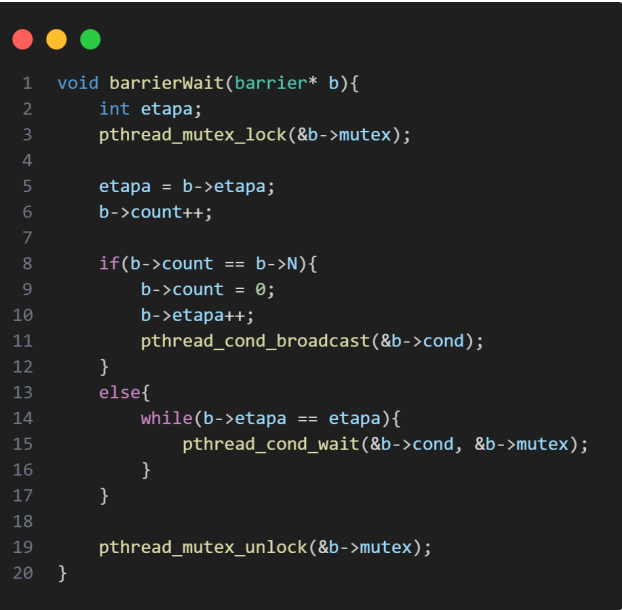


```
1 void barrierInit(barrier* b, int N) {
2     b->count = 0;
3     b->N = N;
4     b->etapa = 0;
5
6     pthread_mutex_init(&b->mutex, NULL);
7     pthread_cond_init(&b->cond, NULL);
8 }
```

Figura 2: Función que inicia barreras

En este código podemos ver la creación de una "estructura" que hará la función de barrera, se consideran 3 datos "globales" para la barrera siguiendo las recomendaciones del enunciado: count, que representa la cantidad de hebras que han llegado al nivel actual, N que representa el numero total de hebras que deben esperar y etapa, que es el identificador de la etapa donde operan las N hebras.

La barrera cuenta también con un mutex y su correspondiente condición de bloqueo. A la derecha esta la correspondiente función que usaremos para iniciar estas barreras.



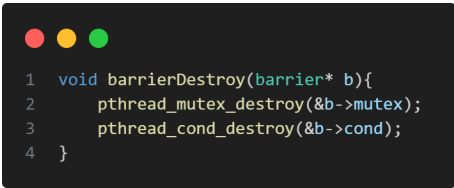
```

1 void barrierWait(barrier* b){
2     int etapa;
3     pthread_mutex_lock(&b->mutex);
4
5     etapa = b->etapa;
6     b->count++;
7
8     if(b->count == b->N){
9         b->count = 0;
10        b->etapa++;
11        pthread_cond_broadcast(&b->cond);
12    }
13    else{
14        while(b->etapa == etapa){
15            pthread_cond_wait(&b->cond, &b->mutex);
16        }
17    }
18
19    pthread_mutex_unlock(&b->mutex);
20 }

```

Figura 3: Función que recibe las hebras

Esta función es la encargada de recibir las hebras, primero podemos ver la creación de una variable local de etapa que tendrá el valor global designado antes. Dentro del mutex creado anteriormente en el struct, la función aumenta el valor de `b->count` cada vez que es llamada, esto simboliza la llegada de una hebra a la barrera. Finalmente se revisa si el numero de hebras que han llegado es el numero de hebras totales, de ser así se reinicia el contador, se aumenta la etapa en 1 y se manda una señal a la barrera de que la condición se cumple, de lo contrario seguimos esperando hebras.



```

1 void barrierDestroy(barrier* b){
2     pthread_mutex_destroy(&b->mutex);
3     pthread_cond_destroy(&b->cond);
4 }

```

Figura 4: Función BarrierDestroy

La función "*BarrierDestroy*" elimina una barrera `b` junto a su condición y su mutex.

```

1 void *thread_work(void *arg) {
2     long tid = (long)arg;
3
4     for (int e = 1; e <= nEtapas; e++) {
5         usleep(rand() % 100000);
6
7         printf("[TID %ld] esperando en etapa %d.\n", tid, e);
8
9         barrierWait(&b);
10
11        printf("[TID %ld] paso barrera en etapa %d.\n", tid, e);
12    }
13
14    return NULL;
15 }

```

Figura 5: función threadWork

"*threadWork*" simula que las hebras están haciendo algún trabajo con "*usleep*", imprime un mensaje informativo en consola y las "*atrapa*" en la barrera hasta que *barrierWait* permita el paso, para terminar informando por consola que cada una de las hebras pasó.

```

1 int main(int argc, char* argv[]){
2
3     if(argc > 2){
4         nThreads = atoi(argv[1]);
5         nEtapas = atoi(argv[2]);
6     }
7
8     srand(time(NULL));
9
10    barrierInit(&b, nThreads);
11
12    pthread_t threads[nThreads];
13
14    for(long i = 0; i < nThreads; i++) pthread_create(&threads[i], NULL, thread_work, (void *)i);
15
16    for(int i = 0; i < nThreads; i++) pthread_join(threads[i], NULL);
17
18    barrierDestroy(&b);
19
20    return 0;
21 }

```

Figura 6: Función Main

Finalmente llegamos al Main del programa, donde asignamos los valores correspondientes a *nThreads* y *nEtapas*, iniciamos el randomizador para el "*trabajo*" de las hebras, iniciamos la barrera, luego creamos las hebras y esperamos a que terminen cada una su "*trabajo*", por ultimo eliminamos la barrera. La impresion por consola en forma escalonada y desordenada de cada una de las hebras implica que cada una esta llegando en un tiempo distinto y que todas son liberadas cuando se llena el cupo de la barrera.

3. Parte II: Simulador Simple de Memoria Virtual

3.1. Actividad 1: Simulador de Traducción de Direcciones

```
1 std::vector<std::pair<int, bool>> frames(nFrames, std::make_pair(-1, false));
2 int pointer = 0;
```

Figura 7: Inicialización de estado

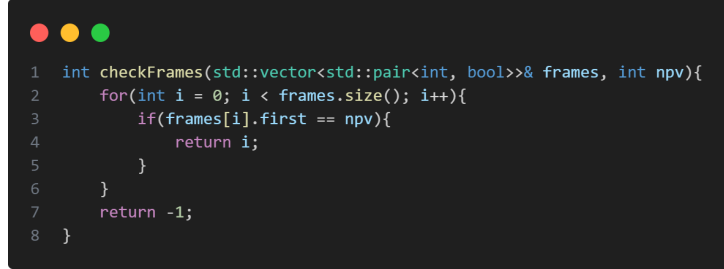
Para modelar la memoria física y el estado necesario para el algoritmo de reemplazo, se utilizaron las siguientes estructuras. Para los marcos de pagina se utilizo un vector de pares para representar la tabla de marcos físicos, first almacena el numero de pagina virtual (nvp) residente del marco inicializado en -1 para que el marco esta libre, second representa el bit de referencia, siendo fundamental para el algoritmo del reloj, indicando si una pagina debe ser reemplazada inmediatamente. Para el puntero del reloj una variable entera que indica la posición actual del puntero de remplazo, recorriendo los marcos de manera circular.

```
1 std::vector<std::pair<int, bool>> frames(nFrames, std::make_pair(-1, false));
2 int pointer = 0;
3 int b = 0;
4 while(nFramesSize){
5     nFramesSize = nFramesSize >> 1;
6     b++;
7 }
8 b--;
9 int mask = (1 << b) - 1;
10
11 std::ifstream file(filename);
12 std::ofstream out("output.txt");
13 std::string vaStr;
14
15 int total = 0;
16 int fallos = 0;
17
18 while(file >> vaStr){
19     total++;
20     int va = strtol(vaStr.c_str(), nullptr, 0);
21     int offset = va & mask;
22     int npv = va >> b;
```

Figura 8: Traducción

Para la traducción de direcciones, el simulador esta diseñado para ser flexible con respecto al tamaño de página, calcula los parámetros dinamicamente al inicio de la ejecución.

Como se aprecia en la Figura 8, el simulador primero configura la geometría de la pagina calculando los bits de desplazamiento (b) y la máscara, luego entra en el bucle principal donde aplica las operaciones bit a bit a cada dirección virtual entrante.



```

1  int checkFrames(std::vector<std::pair<int, bool>>& frames, int npv){
2      for(int i = 0; i < frames.size(); i++){
3          if(frames[i].first == npv){
4              return i;
5          }
6      }
7      return -1;
8  }

```

Figura 9: Reloj checkframes

Cuando ocurre un fallo de página y no hay marcos libres, se ejecuta el algoritmo de reemplazo. La lógica implementada recorre circularmente los marcos usando el puntero. Si el bit de referencia del marco actual es true, se le da una segunda oportunidad se marca como false y el puntero avanza, si el bit es false, se selecciona ese marco como víctima, se actualiza el marco con el nuevo npv, se enciende su bit de referencia true y se actualiza la posición del puntero.

La figura detalla la lógica de reemplazo. Al detectar un fallo, el puntero recorre circularmente los marcos apagando los bits de uso hasta encontrar una víctima, momento en el cual realiza la sustitución

3.2. Verificación y Pruebas

Para validar la correctitud del simulador, se utilizó el archivo de prueba trace.txt con una configuración de 8 marcos y un tamaño de página de 4096 bytes.

3.2.1. Análisis de la Traza de Prueba

La ejecución generó la siguiente salida detallada, la cual verifica el comportamiento esperado de aciertos (HIT) y fallos (FALLO):

Dir. Virtual	NPV	Offset	Evento	Marco	Dir. Física
16384	4	0	FALLO	0	0
4096	1	0	FALLO	1	4096
28672	7	0	FALLO	2	8192
4096	1	0	HIT	1	4096
16388	4	4	HIT	0	4

Cuadro 1: Traza de ejecución de prueba con trace.txt

Como se observa en la Tabla 1, las primeras referencias a páginas nuevas generan fallos y ocupan los marcos secuencialmente. Las referencias posteriores a las páginas 1 y 4 resultan en HITS y la dirección física se calcula correctamente concatenando el índice del marco con el offset.

3.3. 3. Evaluación Experimental

Se realizó un análisis de sensibilidad para observar el comportamiento del algoritmo Reloj ante la variación de memoria física disponible. Se utilizaron dos trazas de referencia con características distintas:

- **Trace 1:** Evaluada con un tamaño de página pequeño (8 bytes).
- **Trace 2:** Evaluada con un tamaño de página estándar (4096 bytes).

Los resultados obtenidos se presentan en la siguiente tabla:

Cantidad de Marcos (N)	Tasa de Fallos (Trace 1)	Tasa de Fallos (Trace 2)
8	0.9855	0.9329
16	0.9694	0.8716
32	0.9415	0.7510

Cuadro 2: Impacto del número de marcos en la tasa de fallos de página.

3.3.1. Análisis de Resultados

Como se evidencia en la Tabla 2, existe una correlación inversa entre el número de marcos asignados y la tasa de fallos.

Se nota una marcada tendencia general, al aumentar la memoria disponible de 8 a 32 marcos, la tasa de fallos disminuye en ambos escenarios. Esto valida el principio de localidad, ya que al aumentar el número de marcos, el conjunto de trabajo del proceso tiene más espacio para residir en memoria, reduciendo la necesidad de reemplazos.

Con respecto al impacto del tamaño de página se observa una mejora mucho más significativa en la Trace 2 (donde la tasa baja de 0.93 a 0.75) en comparación con la Trace 1. Esto sugiere que con páginas más grandes (4096 bytes) y una mayor cantidad de marcos, el algoritmo del Reloj logra capturar mejor la localidad espacial y temporal del programa, reduciendo drásticamente el thrashing(hiperpaginación).