EJERCICIO 1: Multiplicación matriz vector

En el libro la multiplicación matriz x vector, la matriz se reparte en bloques contiguos de filas al hilo “my_rank” le toca desde my_first_row hasta my_last_row, de forma que cada hilo inicializa y calcula sus y[i]. Las variables se declaran globales y compartidas, es decir que todos los hilos leen “x” completo y solo escriben sus propias posiciones de “y” y por eso no se usan locks.

```
1  void *Pth_mat_vect(void* rank) {
2     long my_rank = (long) rank;
3     int i, j;
4     int local_m = m/thread_count;
5     int my_first_row = my_rank*local_m;
6     int my_last_row = (my_rank+1)*local_m - 1;
7
8     for (i = my_first_row; i <= my_last_row; i++) {
9        y[i] = 0.0;
10       for (j = 0; j < n; j++)
11           y[i] += A[i][j]*x[j];
12    }
13
14    return NULL;
15 }  /* Pth_mat_vect */
```

Resultados del libro:

Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

En mi enfoque empaqueto las variables en un struct de manera que cada hilo solo usa lo que está en su ctx

```
8     typedef struct {
9        const double* A; const double* x; double* y;
10       int m, n, T, tid;
11    } Ctx;
```

Y la gran diferencia es la manera en la que se reparten los datos, a diferencia de dar bloques contiguos de filas a cada hilo, se hace una asignación cíclica para las filas y los hilos, es decir, que se reparte la primera fila al primer hilo, la segunda al segundo, la tercera al tercero y la cuarta al cuarto, y se repite, la quinta al primero...

```
35      static void* worker(void* arg){
36          Ctx* c = (Ctx*)arg;
37          for(int i=c->tid;i<c->m;i+=c->T){
38              const double* Ai = c->A + (size_t)i*c->n;
39              double s=0.0; for(int j=0;j<c->n;j++) s += Ai[j]*c->x[j];
40              c->y[i]=s;
41          }
42          return NULL;
43      }
```

Cada y[i] es de un solo hilo

Se calcula la eficiencia con la siguiente formula del libro:

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

Resultados:

```
jean@DESKTOP-I276I98:~$ ./matvec 8 8000000 1
# m=8 n=8000000 T=1 seed=1
T_serial=0.086741 s   T_parallel=0.086005 s   Efficiency=1.008559
jean@DESKTOP-I276I98:~$ ./matvec 8 8000000 2
# m=8 n=8000000 T=2 seed=1
T_serial=0.086766 s   T_parallel=0.045110 s   Efficiency=0.961723
jean@DESKTOP-I276I98:~$ ./matvec 8 8000000 4
# m=8 n=8000000 T=4 seed=1
T_serial=0.085119 s   T_parallel=0.026037 s   Efficiency=0.817283
```

```
jean@DESKTOP-I276I98:~$ ./matvec 8000 8000 1
# m=8000 n=8000 T=1 seed=1
T_serial=0.077873 s   T_parallel=0.079041 s   Efficiency=0.985223
jean@DESKTOP-I276I98:~$ ./matvec 8000 8000 2
# m=8000 n=8000 T=2 seed=1
T_serial=0.083989 s   T_parallel=0.043699 s   Efficiency=0.961007
jean@DESKTOP-I276I98:~$ ./matvec 8000 8000 4
# m=8000 n=8000 T=4 seed=1
T_serial=0.083944 s   T_parallel=0.025031 s   Efficiency=0.838413
```

```
jean@DESKTOP-I276I98:~$ ./matvec 8000000 8 1
# m=8000000 n=8 T=1 seed=1
T_serial=0.159712 s   T_parallel=0.163274 s   Efficiency=0.978183
jean@DESKTOP-I276I98:~$ ./matvec 8000000 8 2
# m=8000000 n=8 T=2 seed=1
T_serial=0.156577 s   T_parallel=0.119676 s   Efficiency=0.654172
jean@DESKTOP-I276I98:~$ ./matvec 8000000 8 4
# m=8000000 n=8 T=4 seed=1
T_serial=0.161159 s   T_parallel=0.132192 s   Efficiency=0.304783
```

	Matrix Dimension					
	8000000 x 8		8000 x 8000		8 x 8000000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.1633	0.978	0.0790	0.985	0.0860	1.00
2	0.1197	0.654	0.0437	0.961	0.0451	0.962
4	0.1322	0.305	0.0250	0.838	0.0260	0.817

EJERCICIO 2: LISTA ENLAZADA

En el libro se presentan 3 variantes de lista enlazada:

1. Mutex para toda la lista: Se pone un mutex para toda la lista de manera que cualquier operación toma ese lock, hace su trabajo y lo suelta, de manera que si un hilo está trabajando con una función las otras no pueden utilizar ni esa ni otra función. Se serializa el acceso a la lista. Si la mayoría de las llamadas son Member, se falla en el intento de paralelismo.
2. Mutex por Nodo: En lugar de darle un solo candado a toda la lista, cada nodo tiene su propio mutex, se avanza en la lista bloqueando el nodo anterior y el nodo actual, y se libera el anterior para avanzar, esto implica hacer locks y unlocks en cada paso, pero permite el paralelismo siempre y cuando las operaciones no caigan en la misma zona.
3. Read-Write locks: Se dividen en 2 tipos de operaciones, read y write, las operaciones del tipo read como Member se pueden correr en paralelo por más de un hilo, pero las operaciones del estilo write que modifican el código no, parecido a un Mutex para toda la lista, se bloquea el acceso hasta que se acabe la operación.

Implementación propia: Estoy usando un busy waiting, al mismo estilo del mutex para toda la lista, en lugar de bloquear la lista entera se utiliza un spinlock global que funciona con un atomic_flag, se realiza una operación a la vez pero en lugar de "dormir" el hilo este se queda esperando girando hasta que sea su turno.

```
18    static inline void spin_lock(atomic_flag* f){
19        while (atomic_flag_test_and_set_explicit(f, memory_order_acquire)) {
```

Si el lock ya está tomado devuelve true y sigue girando hasta que quede libre.

Las operaciones están protegidas por el spin_lock global y se liberan al finalizar su uso:

```c
/*  Operaciones */
static int ll_member(int key){
    spin_lock(&spin);
    Node* cur = head;
    while (cur && cur->key < key) cur = cur->next;
    int found = (cur && cur->key == key);
    spin_unlock(&spin);
    return found;
}
static int ll_insert(int key){
    spin_lock(&spin);
    Node **pp = &head, *cur = head;
    while (cur && cur->key < key) { pp=&cur->next; cur=cur->next; }
    if (cur && cur->key == key) { spin_unlock(&spin); return 0; }
    Node* n = (Node*)malloc(sizeof(Node)); n->key=key; n->next=cur; *pp=n;
    spin_unlock(&spin);
    return 1;
}
static int ll_delete(int key){
    spin_lock(&spin);
    Node **pp = &head, *cur = head;
    while (cur && cur->key < key) { pp=&cur->next; cur=cur->next; }
    if (!cur || cur->key != key) { spin_unlock(&spin); return 0; }
    *pp = cur->next; free(cur);
    spin_unlock(&spin);
    return 1;
}
```

```c
typedef struct { int ops; RNG rng; } ThArgs;

static void* worker(void* a_){
    ThArgs* a = (ThArgs*)a_;
    for (int i=0;i<a->ops;i++){
        int key  = (int)(xr(&a->rng) & 0x7fffffff);
        int pick = (int)(xr(&a->rng) % 100);
        if (pick < MEMBER_PCT)                ll_member(key);
        else if (pick < MEMBER_PCT+INSERT_PCT) ll_insert(key);
        else                                   ll_delete(key);
    }
    return NULL;
}
```

Acá se define el trabajo de cada hilo, repite “ops” veces, genera una clave y decide la operación en base a los porcentajes:

```c
enum { OPS_TOTALES = 100000, INIT_KEYS = 1000, MEMBER_PCT = 80, INSERT_PCT = 10, DELETE_PCT = 10 };
```

```
94          pthread_t* th = (pthread_t*)malloc((size_t)T*sizeof *th);
95          ThArgs*    ta = (ThArgs*)   malloc((size_t)T*sizeof *ta);
96
97          int base = OPS_TOTALES / T, extra = OPS_TOTALES % T;
98          double t0 = now_sec();
99          for (int k=0;k<T;k++){
100             ta[k].ops = base + (k < extra);
101             ta[k].rng.x = 777u + (unsigned)k;
102             pthread_create(&th[k], NULL, worker, &ta[k]);
103         }
104         for (int k=0;k<T;k++) pthread_join(th[k], NULL);
105         double t1 = now_sec();
106
107         printf("threads=%d  time=%.3f s  ops=%d  mix=%d/%d/%d\n",
108                T, t1-t0, OPS_TOTALES, MEMBER_PCT, INSERT_PCT, DELETE_PCT);
109
110         free(ta); free(th); ll_free_all();
111         return 0;
112     }
113
```

Se crean los hilos y se reparte los OPS_TOTALES entre la cantidad de hilos, con el create se pone a trabajar los hilos (worker), y el join espera que terminen su función para poder medir el tiempo final y que no haya hilos que no hayan terminado su trabajo.

Resultados del libro:

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

100,000 ops/thread
80% Member
10% Insert
10% Delete

Resultados propios (busy waiting)

```
jean@DESKTOP-I276I98:~$ ./llist 1
threads=1  time=1.522 s  ops=100000  mix=80/10/10
jean@DESKTOP-I276I98:~$ ./llist 2
threads=2  time=1.665 s  ops=100000  mix=80/10/10
jean@DESKTOP-I276I98:~$ ./llist 4
threads=4  time=1.712 s  ops=100000  mix=80/10/10
jean@DESKTOP-I276I98:~$ ./llist 8
threads=8  time=1.930 s  ops=100000  mix=80/10/10
```

	Número de Threads			
Implementación	1	2	4	8
Busy-waiting	1.522	1.665	1.712	1.930

EJERCICIO 3: SEGURIDAD

En el libro usa semáforos para turnarse y luego tokeniza (strtok no es thread-safe y falla), recomienda usar strtok_r y round-robin, no lo implementa y esta la versión que uso.

```
 9    void* worker(void* a_) {
10        Args* a = (Args*)a_;
11        for (int idx = a->tid; idx < a->n; idx += a->T) {
12            char* line = a->lines[idx];
13            if (!line) continue;
14            char* save = NULL;
15            char* tok = strtok_r(line, " \t\n", &save);   // versión segura
16            while (tok) {
17                printf("[T%02d L%03d] %s\n", a->tid, idx, tok);
18                tok = strtok_r(NULL, " \t\n", &save);
19            }
20        }
21        return NULL;
22    }
```

```
32            lines[n++] = strdup(buf); // cada hilo tokeniza su copia
```

Se reparte el trabajo con al estilo round-robin, en este caso las filas se reparten entre los hilos. A diferencia de strtok que usa un estado interno global y no es seguro con hilos concurrentes, strtok_r es reentrante, se le pasa un “save” local (puntero a puntero que guarda el estado de tokenización de esa cadena), de manera que cada hilo tenga su save, esto evita las carreras y los errores.

Cada hilo modifica y tokeniza sus filas y copias de las filas con strdup (corta las palabras como tokens usando los separadores espacio, tab, nueva línea).

Texto:

```
  GNU nano 7.2
Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.
```

```
jean@DESKTOP-I276I98:~$ ./safe 2 < texto.txt
[T00 L000] Pease
[T00 L000] porridge
[T00 L000] hot.
[T00 L002] Pease
[T00 L002] porridge
[T00 L002] in
[T00 L002] the
[T00 L002] pot
[T01 L001] Pease
[T01 L001] porridge
[T01 L001] cold.
[T01 L003] Nine
[T01 L003] days
[T01 L003] old.
```