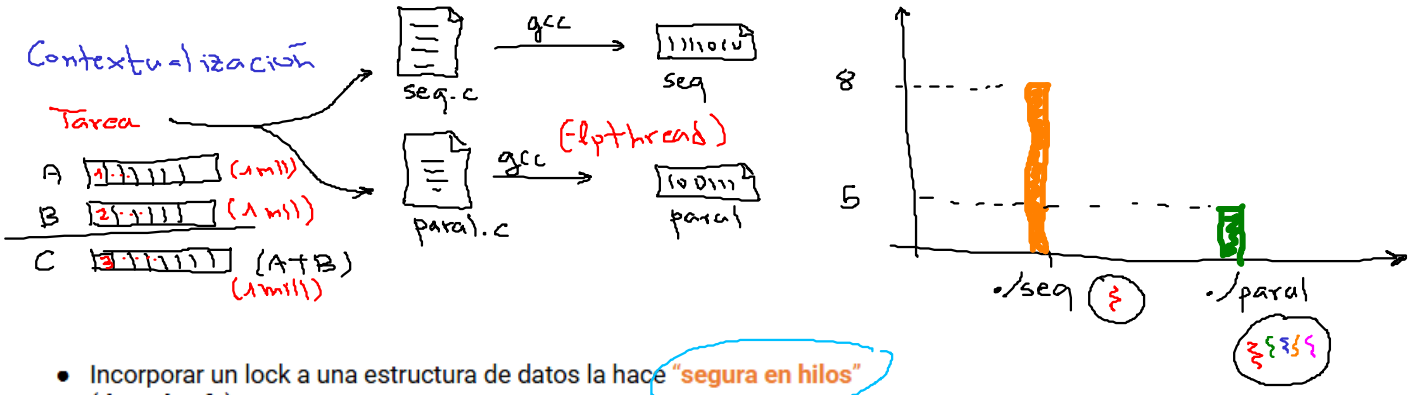
06 / 11 / 2025 – Sistemas Operativos (Vde@)

Estructuras de datos concurrentes

1. Contextualización



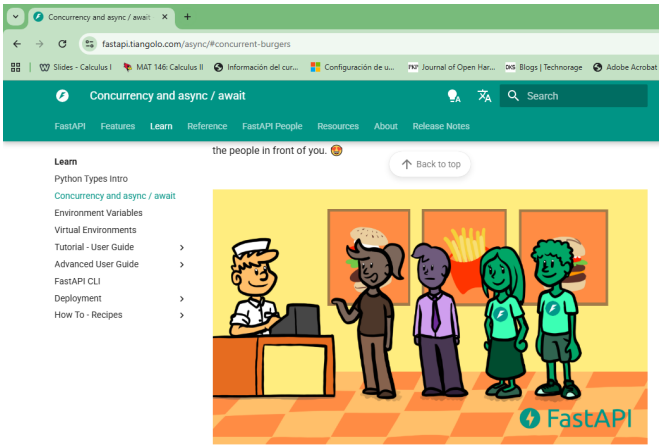
- Incorporar un lock a una estructura de datos la hace "segura en hilos" (**thread safe**).

Idea clave: ¿Cómo agregar locks a estructuras de datos?

- **Funcionamiento correcto (correctness)**: Cuando una estructura de datos particular es dada, ¿Como podríamos agregar locks para hacer que esta trabaje adecuadamente?
- **Rendimiento (performance)**: Como agregar locks de tal manera que la estructura de datos permita que muchos procesos accedan a la estructura sin que el rendimiento caiga?

- Agregar concurrencia en estructuras de datos es una disciplina ampliamente estudiada. → Computacion de alto desempeño.

https://fastapi.tiangolo.com/async/



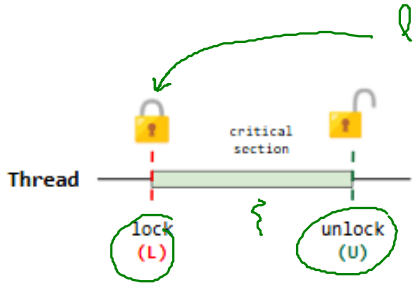
Objetivo: Evitar Race condition → lock

2. Contadores concurrentes

¿Cómo hacer un contador thread-safe?

Siga el patrón básico:

1. Agregar el lock y adquirirlo.
2. Manipular la estructura de datos (Región crítica).
3. Liberar el lock

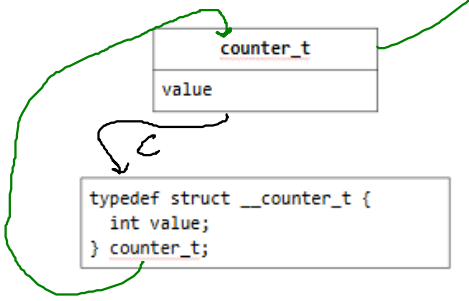


a. Contador sin locks

Contadores sin locks

No se protege el acceso a la región crítica:

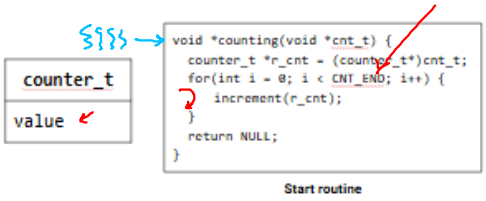
- Se producen condiciones de carrera.
- No es safe thread.




```
typedef struct __counter_t {
  int value;
} counter_t;
void init(counter_t *c) {
  c->value = 0;
}
void increment(counter_t *c) {
  c->value++;
}
void decrement(counter_t *c) {
  c->value--;
}
int get(counter_t *c) {
  return c->value;
}
```

Funciones.

Contadores sin locks



Start routine

Prueba	Número de hilos	Valor esperado	Valor obtenido
1	1	1000000	1000000
2	8	8000000	1162851

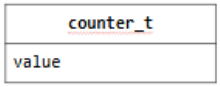
Race condition.

```
int real_value = 0;
counter_t cnt;
int cnt_value;

int main() {
  init(&cnt);
  int i;
  pthread_t tid[NUMTHREADS];

  for(i = 0; i < NUMTHREADS; i++) {
    pthread_create(&tid[i], NULL, counting, &cnt);
  }

  for(i = 0; i < NUMTHREADS; i++) {
    pthread_join(tid[i], NULL);
  }

  printf("-> El contador debe quedar en:
        %d\n",NUMTHREADS*CNT_END);
  printf("-> El valor real del contador es: %d\n",
        get(&cnt));

  return 0;
}
```

Contadores sin locks

No se protege el acceso a la región crítica:

- Se producen condiciones de carrera.
- No es safe thread.




```
typedef struct __counter_t {
  int value;
} counter_t;
```

```
typedef struct __counter_t {
  int value;
} counter_t;
void init(counter_t *c) {
  c->value = 0;
}
void increment(counter_t *c) {
  c->value++;
}
void decrement(counter_t *c) {
  c->value--;
}
int get(counter_t *c) {
  return c->value;
}
```

Race condition.

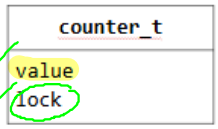
Prueba	Número de hilos	Valor esperado	Valor obtenido	tiempo gastado (seg)
1	1	1000000	1000000	0.004213
2	2	2000000	1030174	0.004213
3	4	4000000	1033970	0.025465
4	8	8000000	1309690	0.044530
5	16	16000000	1177806	0.088573

b. Contador con locks. ← Solucionar el problema de las condiciones de carrera.

Contadores con locks

Se protege el acceso a la región crítica:

- Se incorpora un único **lock** el cual protege el acceso al contador.
- Es **safe thread**.




```
typedef struct __counter_t {
  int value;
  pthread_mutex_t lock;
} counter_t;
```

```
void init(counter_t *c) {
  c->value = 0;
  Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
  c->value++;
  Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
  c->value--;
  Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
  int rc = c->value;
  Pthread_mutex_unlock(&c->lock);
  return rc;
}
```

```
counter_t
value
lock
```

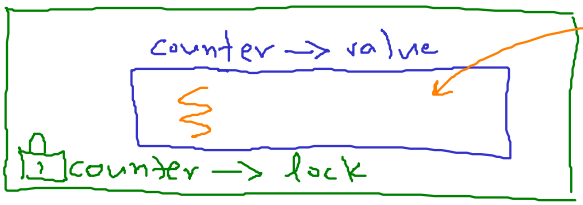
```
void *counting(void *cnt_t) {
  counter_t *r_cnt = (counter_t*)cnt_t;
  for(int i = 0; i < CNT_END; i++) {
    increment(r_cnt);
  }
  return NULL;
}
```

Ya usa lock!

Start routine

```
int real_value = 0;
counter_t cnt;
int cnt_value;

int main() {
  init(&cnt);
  int i;
  pthread_t tid[NUMTHREADS];

  for(i = 0; i < NUMTHREADS; i++) {
    pthread_create(&tid[i], NULL, counting, &cnt);
  }

  for(i = 0; i < NUMTHREADS; i++) {
    pthread_join(tid[i], NULL);
  }

  printf("-> El contador debe quedar en:
         %d\n",NUMTHREADS*CNT_END);
  printf("-> El valor real del contador es: %d\n",
         get(&cnt));

  return 0;
}
```

Main function



8 hilos:

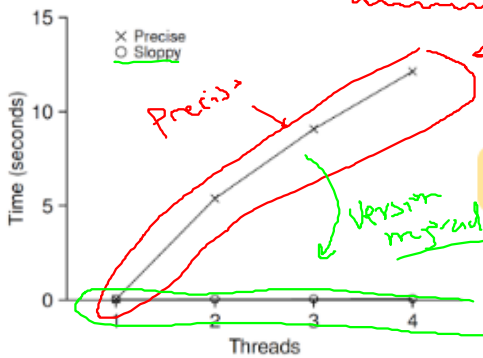
Prueba	Número de hilos	Valor esperado	Valor obtenido
1	1	1000000	1000000
2	2	2000000	2000000
3	4	4000000	4000000
4	8	8000000	8000000
5	16	16000000	16000000

Prueba	Número de hilos	Valor esperado	Valor obtenido	tiempo gastado (seg)
1	1	1000000	1000000	0.019288
2	2	2000000	2000000	0.080396
3	4	4000000	4000000	0.173779
4	8	8000000	8000000	0.331235
5	16	16000000	16000000	0.678422

Análisis de desempeño – Contador con locks

- Cada hilo actualiza un contador compartido.
  - Se actualiza un millón de veces.
  - Máquina de test (Remzi): iMac con 4 Intel 2.7GHz i5 CPUs.

```
counter_t
value
lock
```



Performance of Traditional vs. Sloppy Counters
(Threshold of Sloppy, S, is set to 1024)

Ya no presenta Race cond.
- No escala x

Precisa

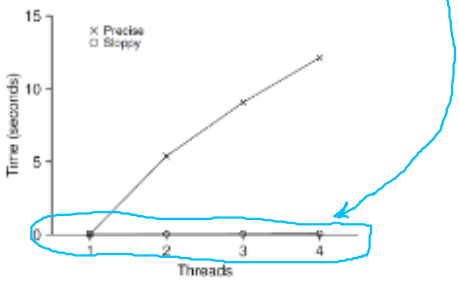
Versión mejorada

El contador sincronizado escala pobremente.

Contador aproximado.
Como se logra que escale?

Análisis de desempeño – Contador con locks

- **Conclusión**: El contador sincronizado escala pobremente.
- **Objetivo**: ¿Cómo lograr un scaling perfecto?
  - A pesar de que se hace más trabajo, se hace en paralelo.
  - El tiempo que toma realizar la tarea completa no debería incrementar.



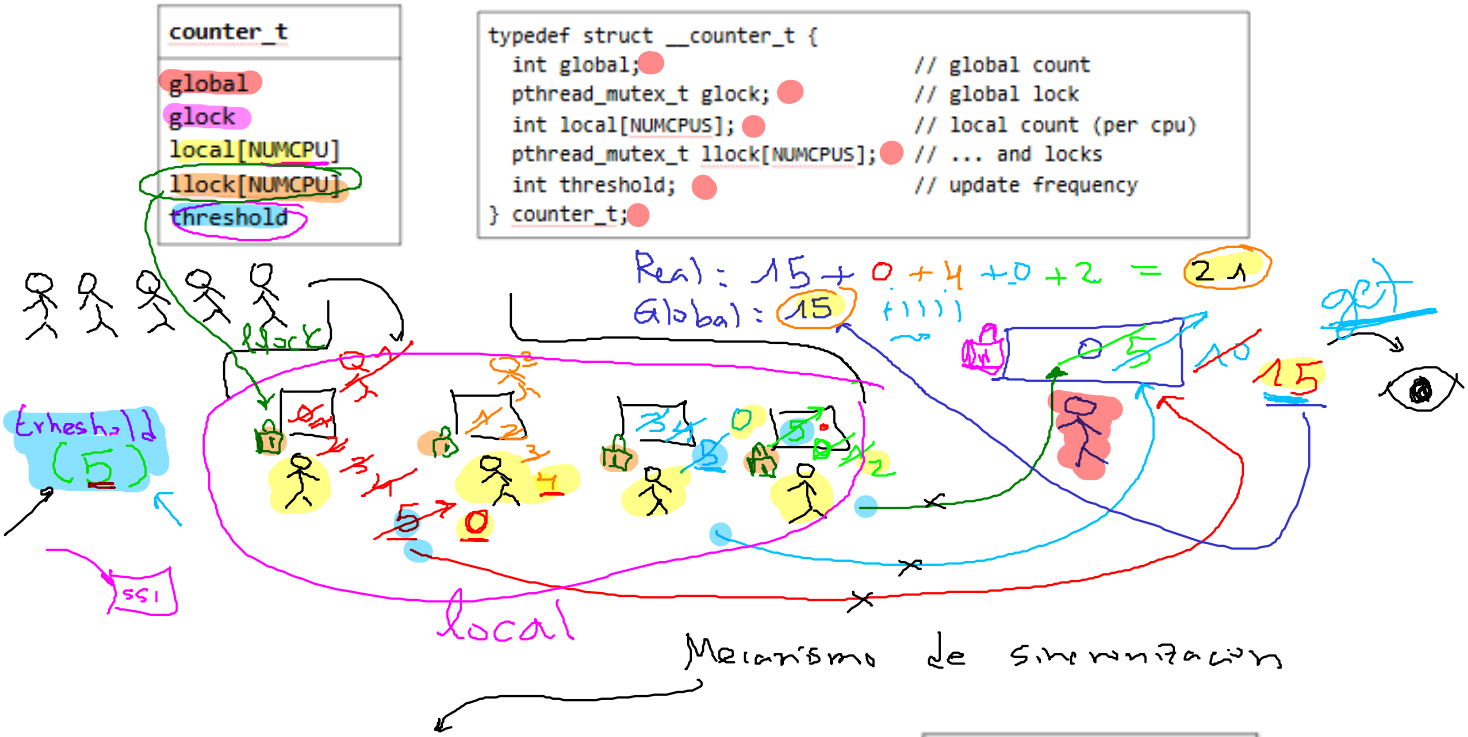

```c
int main() {
  // Declaration of struct timeval variables
  struct timeval start, end;
  long int time_elapsed;

  init(&cnt);
  int i;
  pthread_t tid[NUMTHREADS];

  // Get the start time
  gettimeofday(&start, NULL);

  for(i = 0; i < NUMTHREADS; i++) {
    pthread_create(&tid[i], NULL, counting, &cnt);
  }

  for(i = 0; i < NUMTHREADS; i++) {
    pthread_join(tid[i], NULL);
  }
  // Get the end time
  gettimeofday(&end, NULL);
  time_elapsed = ((end.tv_sec*1000000 + end.tv_usec)
                  - (start.tv_sec*1000000 + start.tv_usec));
  ...
  return 0;
}
```

c. Contador aproximado.

Contador aproximado (Sloppy counter)

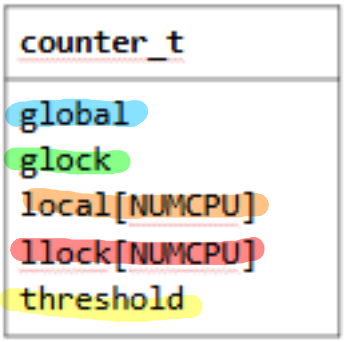
Implementación:

- Un contador lógico se implementa a través de múltiples contadores físicos, uno por core.
- Un contador global.
- Locks: Uno por contador local y uno para el contador global



Contador aproximado (Sloppy counter)

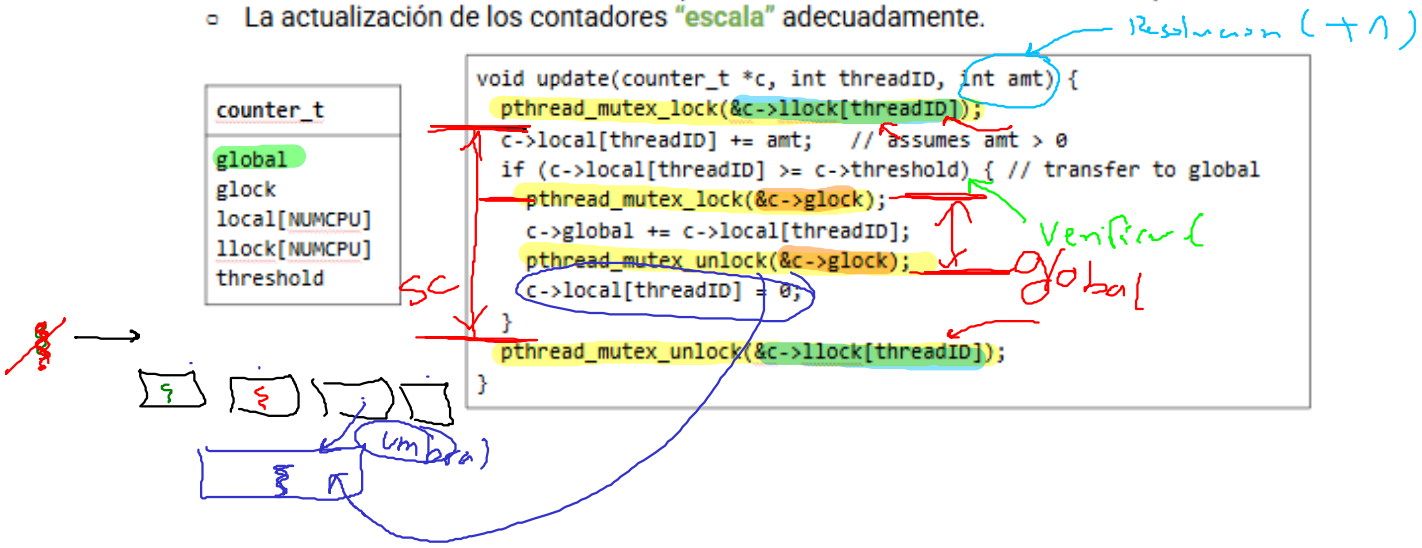
- Cuando un hilo desea **incrementar el contador**:
  - Incrementa solo el contador local (`local[cpu_i]`).
  - Cada CPU tiene su contador local (`local[cpu_i]`).
    - Hilos corriendo en **diferentes CPUs** pueden actualizar su contador sin competencia.
    - La actualización de los contadores "escala" adecuadamente.
- Los valores del contador local (`local[cpu_i]`). se envían al contador global (`global`) periódicamente.
  - Adquiere el lock global (`glock`).
  - Incrementa el valor de acuerdo al contador local (`local[cpu_i]`).
  - El contador local se "**resetea**" (se lleva a 0).



Contador aproximado (Sloppy counter)

Contador local (asociado a cada hilo):

- Incrementa solo el **contador local** (`local[cpu_i]`).
- Cada CPU tiene su **contador local** (`local[cpu_i]`).
  - Hilos corriendo en **diferentes CPUs** pueden actualizar su contador sin competencia.
  - La actualización de los contadores "escala" adecuadamente.




```
void update(counter_t *c, int threadID, int amt) {
  pthread_mutex_lock(&c->llock[threadID]);
  c->local[threadID] += amt;   // assumes amt > 0
  if (c->local[threadID] >= c->threshold) { // transfer to global
    pthread_mutex_lock(&c->glock);
    c->global += c->local[threadID];
    pthread_mutex_unlock(&c->glock);
    c->local[threadID] = 0;
  }
  pthread_mutex_unlock(&c->llock[threadID]);
}
```

Contador aproximado (Sloppy counter)

¿Con qué frecuencia se actualiza el contador global (`global`)?

```
counter_t
global
glock
local[NUMCPU]
llock[NUMCPU]
threshold
```

```
void update(counter_t *c, int threadID, int amt) {
  pthread_mutex_lock(&c->llock[threadID]);
  c->local[threadID] += amt;   // assumes amt > 0
  if (c->local[threadID] >= c->threshold) { // transfer to global
    pthread_mutex_lock(&c->glock);
    c->global += c->local[threadID];
    pthread_mutex_unlock(&c->glock);
    c->local[threadID] = 0;
  }
  pthread_mutex_unlock(&c->llock[threadID]);
}
```

Parámetro S (threshold):

- **S pequeño**: el contador se comporta con un contador non-scalable.
- **S grande**:
  - Contador más escalable.
  - El valor del contador local se aleja del valor real.

Contador aproximado (Sloppy counter)

Ejemplo: máquina con 4 cores

- 4 contadores locales: **`local[NUMCPU] = Li`**; donde **`NUMCPU`** está asociado al hilo por medio de su **`threadID`**
- 1 contador global: **`global = G`**

Seguimiento del sloppy counters:

- El threshold S es fijado a 5: **`threshold = 5`**
- Hay 4 hilos uno por cada CPU.
- Cada hilo actualiza su propio contador local (**`L1`**, **`L2`**, **`L3`** y **`L4`**).

Ejemplo: máquina con 4 cores

- 4 contadores locales: L1, L2, L3 y L4
- 1 contador global: G
- El threshold S es fijado a 5: S = 5

```
void update(counter_t *c, int threadID, int amt) {
  pthread_mutex_lock(&c->llock[threadID]);
  c->local[threadID] += amt;
  if (c->local[threadID] >= c->threshold) {
     pthread_mutex_lock(&c->glock);
     c->global += c->local[threadID];
     pthread_mutex_unlock(&c->glock);
     c->local[threadID] = 0;
  }
  pthread_mutex_unlock(&c->llock[threadID]);
}
```

Time	L1	L2	L3	L4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L1)
7	0	2	4	5 → 0	10 (from L4)

Contador aproximado (Sloppy counter)

Implementación - Estructura de datos

counter_t
global
glock
local[NUMCPU]
llock[NUMCPU]
threshold

```
typedef struct __counter_t {
  int global;                      // global count
  pthread_mutex_t glock;           // global lock
  int local[NUMCPUS];              // local count (per cpu)
  pthread_mutex_t llock[NUMCPUS]; // ... and locks
  int threshold;                   // update frequency
} counter_t;
```

Implementación - Función de inicialización

counter_t
global
glock
local[NUMCPU]
llock[NUMCPU]
threshold

```
// init: record threshold, init locks, init values
//       of all local counts and global count
void init(counter_t *c, int threshold) {
  c->threshold = threshold;
  c->global = 0;
  pthread_mutex_init(&c->glock, NULL);
  int i;
  for (i = 0; i < NUMCPUS; i++) {
    c->local[i] = 0;
    pthread_mutex_init(&c->llock[i], NULL);
  }
}
```

Contador aproximado (Sloppy counter)

Implementación - Función actualización del contador

```
counter_t
global
glock
local[NUMCPU]
llock[NUMCPU]
threshold
```

```
// update: usually, just grab local lock and update local amount
//         once local count has risen by 'threshold', grab global
//         lock and transfer local values to it

void update(counter_t *c, int threadID, int amt) {
  pthread_mutex_lock(&c->llock[threadID]);
  c->local[threadID] += amt;     // assumes amt > 0
  if (c->local[threadID] >= c->threshold) { // transfer to global
    pthread_mutex_lock(&c->glock);
    c->global += c->local[threadID];
    pthread_mutex_unlock(&c->glock);
    c->local[threadID] = 0;
  }
  pthread_mutex_unlock(&c->llock[threadID]);
}
```

Contador aproximado (Sloppy counter)

Implementación - Función para obtener el valor del contador

```
counter_t
global
glock
local[NUMCPU]
llock[NUMCPU]
threshold
```

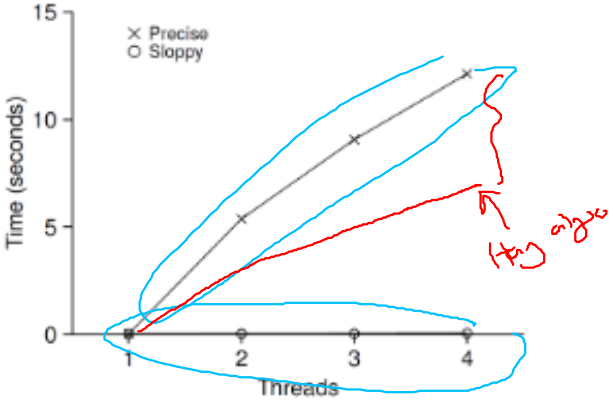
```
// get: just return global amount (which may not be perfect)
int get(counter_t *c) {
  pthread_mutex_lock(&c->glock);
  int val = c->global;
  pthread_mutex_unlock(&c->glock);
  return val; // only approximate!
}
```

Análisis de desempeño –Contador aproximado

- Cada hilo actualiza un contador compartido.
  - Se actualiza un millón de veces.
  - Máquina de test (Remzi): iMac con 4 Intel 2.7GHz i5 CPUs.

```
counter_t
value
lock
```

Traditional Counter



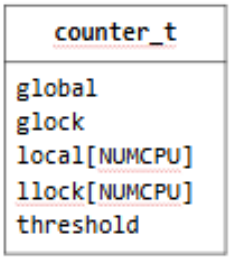
Performance of Traditional vs. Sloppy Counters
(Threshold of Sloppy, S, is set to 1024)

```
counter_t
global
glock
local[NUMCPU]
llock[NUMCPU]
threshold
```

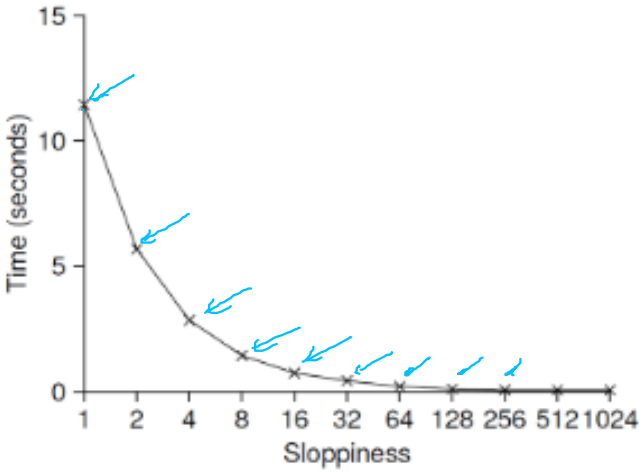
Sloppy Counters

Análisis de desempeño –Contador aproximado

- Cuatro hilos, cada uno aumenta contador un millón de veces:
  - **S pequeño**: pobre desempeño, valor global siempre exacto.
  - **S grande**: excelente desempeño, valor global se retrasa.



Sloppy Counters

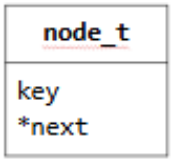


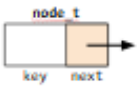
Scaling Sloppy Counters

3 - Listas enlazadas concurrentes

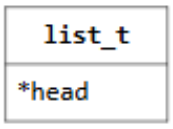
Lista enlazada sin concurrencia

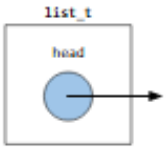
Definición de las estructuras de datos empleadas (Nodo y lista)






```
// basic node structure
typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;
```






```
// basic list structure (one used per list)
typedef struct __list_t {
  node_t *head;
} list_t;
```

Lista enlazada sin concurrencia

Operaciones básicas sobre la lista enlazada

node_t
key *next

list_t
*head

Inicialización: List_Init

```
void List_Init(list_t *L) {
  L->head = NULL;
}
```

Inserción: List_Insert

```
int List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return -1; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  return 0; // success
}
```

Busqueda: List_Lookup

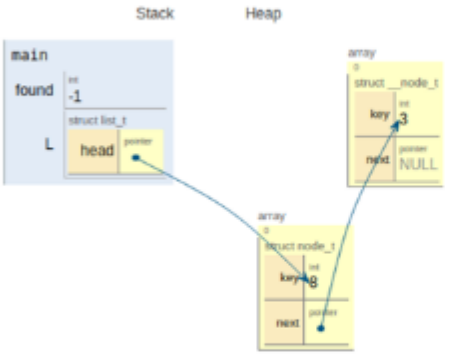
```
int List_Lookup(list_t *L, int key) {
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      return 0; // success
    }
    curr = curr->next;
  }
  return -1; // failure
}
```

Lista enlazada sin concurrencia

Ejemplo

```
int main() {
  int found;
  list_t L;
  List_Init(&L);
  List_Insert(&L,3);
  List_Insert(&L,8);
  found = List_Lookup(&L, 3);
  found = List_Lookup(&L, 8);
  printf("3 in L? %d\n",found);
  found = List_Lookup(&L, 5);
  printf("3 in L? %d\n",found);
  return 0;
}
```

Link simulación



list_t
*head

node_t
key *next

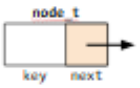
Print output

```
3 in L? 0
3 in L? -1
```

Lista enlazada concurrente

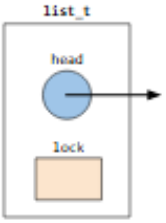
Definición de las estructuras de datos empleadas (Nodo y lista)

node_t
key
*next




```
// basic node structure
typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;
```

list_t
*head
lock




```
// basic list structure (one used per list)
typedef struct __list_t {
  node_t *head;
  pthread_mutex_t lock;
} list_t;
```

Lista enlazada concurrente

Operaciones básicas sobre la lista enlazada concurrente

Función inicializar recurrente:

- El lock es inicializado.

node_t
key
*next

list_t
*head
lock

Inicialización: `List_Init`

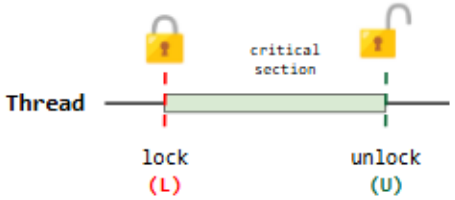
```
void List_Init(list_t *L) {
  L->head = NULL;
  pthread_mutex_init(&L->lock, NULL);
}
```

Lista enlazada concurrente

Operaciones básicas sobre la lista enlazada concurrente

Función insertar recurrente:

- El código **adquiere** el lock, en la entrada de la rutina insertar.
- El código **libera** el lock al salida.
- Si el malloc falla, **libera** el lock.



Inserción: `List_Insert`

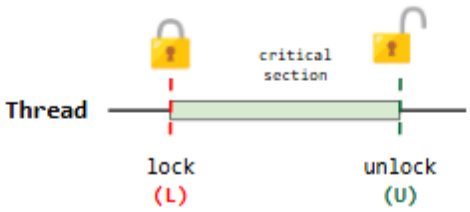
```
int List_Insert(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->lock);
    return -1; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return 0; // success
}
```

Lista enlazada concurrente

Operaciones básicas sobre la lista enlazada concurrente

Función busqueda recurrente:

- El código **adquiere** el lock, en la entrada de la rutina insertar.
- El código **libera** el lock al salida.
- Si el malloc falla, **libera** el lock.



Busqueda: `List_Lookup`

```
int List_Lookup(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      pthread_mutex_unlock(&L->lock);
      return 0; // success
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return -1; // failure
}
```

Mejora:

- En las versiones anteriores (List_Insert y List_Lookup) se podía dar una falla en el malloc. Si esto sucedía lo que se hacía era liberar el lock.
- Manejo de flujo de control excepcional (propenso a errores).
- **Solución**: adquisición y liberación del lock solo alrededor de la región crítica

Insert: `List_Insert`

```
int List_Insert(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->lock);
    return -1; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return 0; // success
}
```

Insert mejorado: `List_Insert`

```
void List_Insert(list_t *L, int key) {
  // synchronization not needed
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return;
  }
  new->key = key;

  // just lock critical section
  pthread_mutex_lock(&L->lock);
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
}
```


Mejora:

- En las versiones anteriores (List_Insert y List_Lookup) se podía dar una falla en el malloc. Si esto sucedía lo que se hacía era liberar el lock.
- Manejo de flujo de control excepcional (propenso a errores).
- **Solución**: adquisición y liberación del lock solo alrededor de la región crítica

Busqueda: `List_Lookup`

```
int List_Lookup(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      pthread_mutex_unlock(&L->lock);
      return 0; // success
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return -1; // failure
}
```

Busqueda mejorada: `List_Lookup`

```
int List_Lookup(list_t *L, int key) {
  int rv = -1;
  pthread_mutex_lock(&L->lock);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      rv = 0;
      break;
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return rv; // now both success and failure
}
```

Resumen implementación

Insert: `List_Insert`

```
void List_Insert(list_t *L, int key) {
  // synchronization not needed
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return;
  }
  new->key = key;

  // just lock critical section
  pthread_mutex_lock(&L->lock);
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
}
```

Inicialización: `List_Init`

```
void List_Init(list_t *L) {
  L->head = NULL;
  pthread_mutex_init(&L->lock, NULL);
}
```

Busqueda: `List_Lookup`

```
int List_Lookup(list_t *L, int key) {
  int rv = -1;
  pthread_mutex_lock(&L->lock);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      rv = 0;
      break;
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return rv; // now both success and failure
}
```

Escalamiento

Hand-over-hand locking:

- Incluye un `lock` por nodo en la lista, en lugar de tener solo un `lock` por lista.
- **Al recorrer la lista**:
  - Primero se adquiere el `lock` del siguiente nodo.
  - Luego libera el bloqueo del nodo actual.
- **Alto grado de concurrencia en las operaciones**:
  - En la práctica, el **sobrecosto (overhead)** general de la adquisición y liberación de locks para cada nodo de un recorrido es **prohibitivo**.