Insert into Sorted Linked List: synchronization with Locks & Atomics

Universitat Autònoma de Barcelona

El programa de la derecha crea una lista enlazada y ordenada de números reales entre 0 y 1000 que se generan de forma aleatoria. Se paraleliza con *threads* el bucle for que viene a continuación, para insertar de forma concurrente los números aleatorios en la lista encadenada. Observar que la generación en paralelo de números pseudo-aleatorios requiere que cada *thread* utilice su propia semilla (*seed*), porque la generación de números aleatorios es una tarea básicamente secuencial.

El código de la función Insert se muestra a continuación, abajo, y no es adecuado para ser ejecutado en paralelo, es decir, no es thread-safe. Hay una condición de carrera (**race condition**) cuando dos *threads* intentan insertar un nodo en la misma posición que puede provocar que se deje de insertar algún nodo, o que la lista no quede bien ordenada.

```
// Define a node struct to hold a value
struct node {
  double val;
  node *next;
  omp_lock_t l;
};
…
// create empty list with zero node always in front
node *L= new node;
L->val= 0;  L->next= NULL;

#pragma omp parallel shared(Seed,L,N)
{
  int tid = omp_get_thread_num();

  #pragma omp for
  for (int i=0; i<N; ++i)
  {
    double x = 1000*erand48( Seed[tid] );
    Insert ( L, x );
  }
}
```

```
void Insert ( node *L, double v)
{
  node *prev, *ptr;
  prev= L; ptr = L->next;

  node *t = new node;
  t->val  = v;

  // find insertion point
  while (ptr && ptr->val <= v)
  {
    prev = ptr;
    ptr  = ptr->next;
  }
  // insert
  t->next = ptr;
  prev->next= t;
}
```

Problema: Añadir una o varias regiones críticas a la función Insert para que el programa funcione siempre correctamente, aunque sea a costa de una reducción considerable del rendimiento del programa. Podéis cambiar solamente el código de la función Insert. Si en 10 minutos no habéis encontrado el problema, preguntad al profesor.

La solución correcta es guardar los accesos con **locks**. Abajo a la derecha se muestra el código con **locks**: las sentencias de color amarillo son las que se han añadido al programa original. Pero resulta que el programa no es correcto y tiene una condición de carrera.

Problema: Encontrar el problema de la versión con **locks** arreglarlo. Si en 10 minutos no habéis encontrado el problema, preguntad al profesor.

Pregunta: Evaluar el rendimiento de la ejecución (compilador gcc 8.2) de las dos versiones, la original (incorrecta) y la versión con **locks** (*thread-safe*). Explicar los resultados haciendo referencia a los datos obtenidos con la utilidad **perf**.

Opcional: (Dificultad Media) Escribir una versión especulativa con **locks** sólo en la parte final de la función en la que se produce la inserción del nuevo nodo en la lista, pero sin usar **locks** para buscar el lugar de inserción. **Cuidado**: el orden en el que se actualizan los apuntadores del nodo insertado y de la lista es crítico para el correcto funcionamiento del programa. Medir el rendimiento y explicarlo. Mirad la referencia que tenéis en el Campus Virtual en la que se explica la idea de la técnica de la especulación.

Opcional: (Difícil) Modificar la versión anterior para que funcione sin **locks** (**non-blocking**), utilizando la **operación atómica compare and swap**. Esta operación no está disponible como **#pragma** de openMP, y a cambio se debe usar la función built-in "**bool __atomic_compare_exchange_n**".

```
void InsertL ( node *L, double v)
{
  node *prev= L;
  node *ptr = L->next;
  node *t = new node;
  t->val  = v;

  // find insertion point
  omp_set_lock(&prev->l);
  while (ptr && ptr->val <= v)
  {
    omp_unset_lock(&prev->l);
    prev = ptr;
    omp_set_lock(&prev->l);
    ptr  = ptr->next;
  }
  // insert
  t->next = ptr;
  prev->next= t;
  omp_unset_lock(&prev->l);
}
```