Sistemas Operativos

Escuela de Ingeniería Civil Informática

- Administración de Procesos
  Sincronización con Locks

UNIVERSIDAD DEL BÍO-BÍO

TEMARIO

- MECANISMOS DE SINCRONIZACIÓN
  - Locks (candados)
  - Semáforos
  - Mensajes
  - Monitores

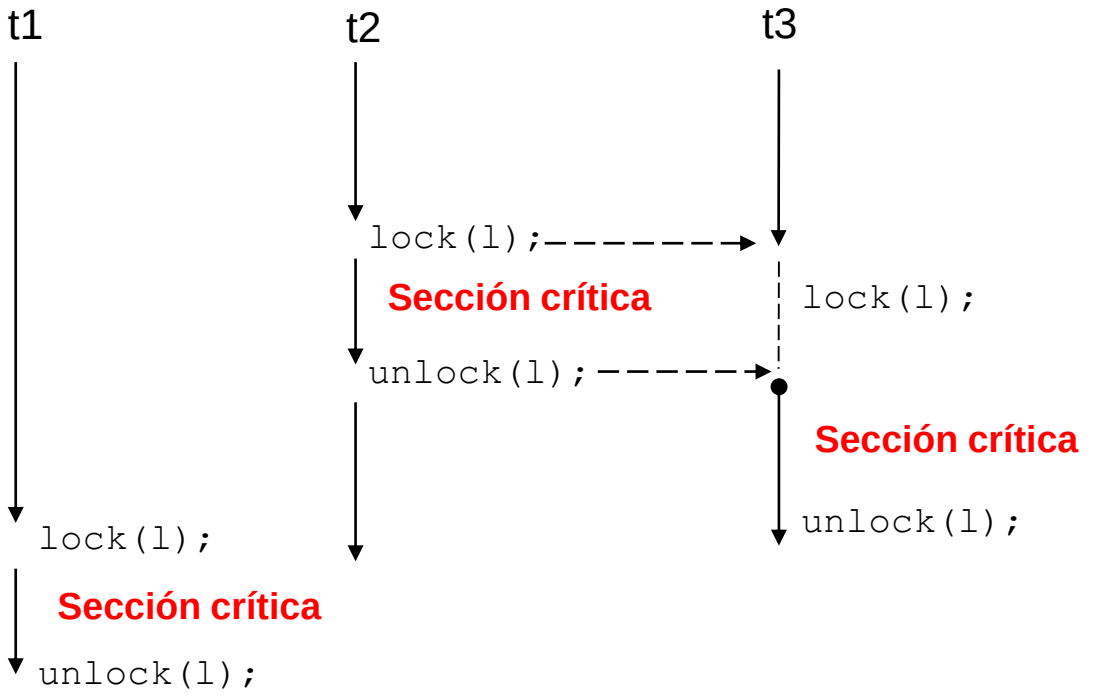
- PROBLEMAS CLÁSICOS
  - Productor/Consumidor
  - Filósofos comensales
  - Lectores/Escritores

- PROBLEMAS DE SINCRONIZACIÓN
  - Data race
  - Deadlock
  - Starvation

CANDADOS O LOCKS

- Son también conocidos como mutex o semáforos binarios.
- Corresponden a un mecanismo muy primitivo de sincronización.
- Sirven para garantizar la exclusión mutua.
- Ejemplo de uso:

```
Lock l;

nMain(...) {
   l = makeLocks();
   ...
}
```



CANDADOS O LOCKS

- La solución al ejercicio del factorial concurrente se puede apreciar a continuación:

```
double r;
Lock l= makeLocks();

int pmul(...) {
  if (...) {
    double v;
    ...
    lock(l);
    r = r * v;
    unlock(l);
  }
  else {
    ...
  }
}
```

Sección crítica, variable r es compartida por otras tareas

CANDADOS EN NANO SYSTEM

- NSYSTEM permite el uso de Locks.
- La API es la siguiente:
  - ***Lock makeLock();***
    Crea un candado inicialmente abierto.
  - ***void lock(Lock l);***
    Solicita cerrar el candado (si está cerrado, el proceso debe esperar)
  - ***void unlock(Lock l);***
    Abre el candado

EJEMPLO: DICCIONARIO CONCURRENTE

- Este diccionario posee las siguiente operaciones:
  - `void newDef(char *k, char *d);`
  - `char *query(char *k);`
  - `void delDef(char *k);`

- Forma de uso:

```
newDef(“a”, “123”);
printf(“%s”, query(“a”));   /*123*/
newDef(“b”, “456”);
delDef(“a”);
query(“a”);    /*NULL*/
```

EJEMPLO: DICCIONARIO CONCURRENTE

- Implementación de las operaciones:

```
#define MAX 100

char *keys[MAX]; /*asumimos que se inicializó en NULL*/
char *defs[MAX]; /*asumimos que se inicializó en NULL*/


int getSlot() {
  int i;
  for(i=0; i<MAX; i++)
    if (keys[i]==NULL)
       return i;
  error(); /*asumimos que no retorna*/
  return 0;
}


void newDef(char *k, char *d) {
  int i = getSlot();
  keys[i]= k;
  defs[i]= d;
}
```

EJEMPLO: DICCIONARIO CONCURRENTE

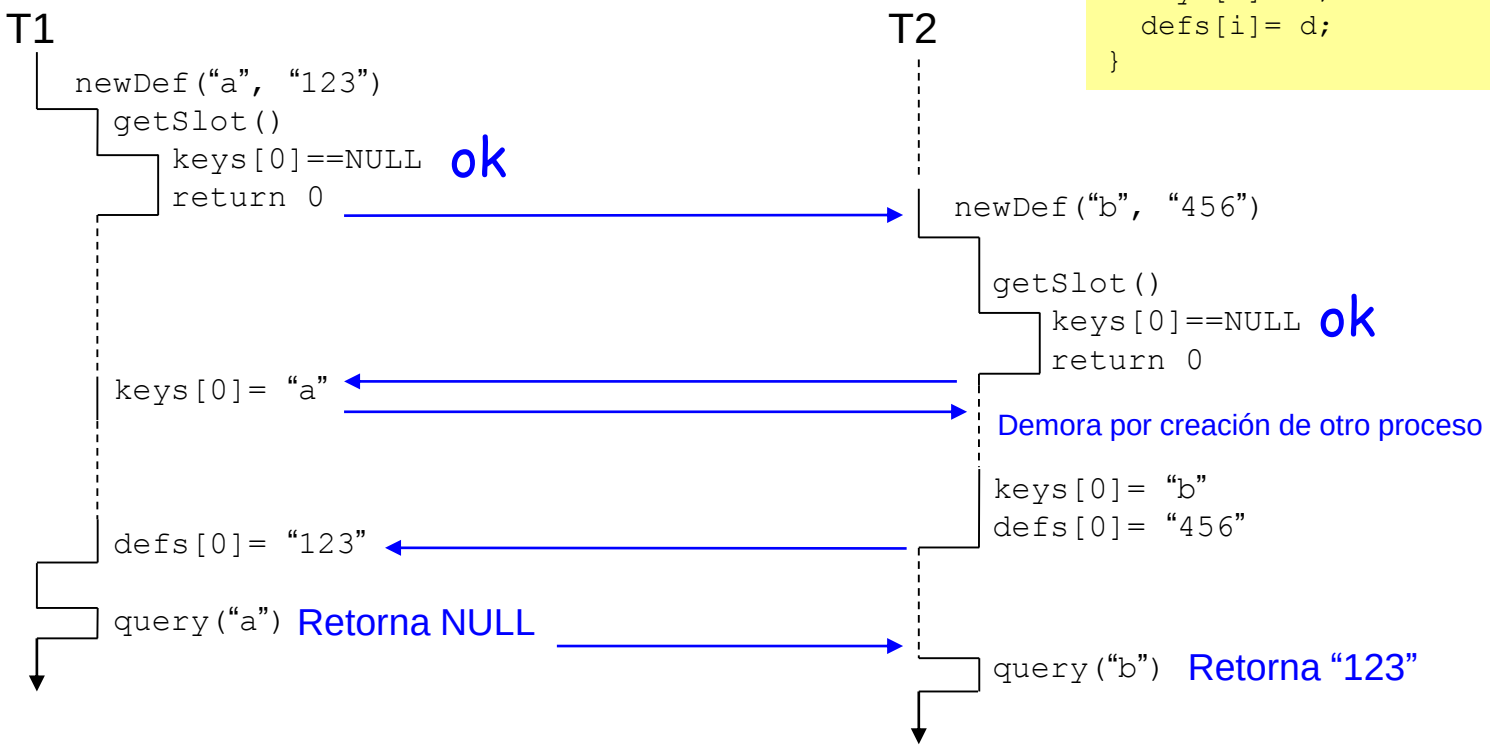
```
int getIdx(char *k) {
  int i;
  for(i=0; i<MAX; i++)
    if (keys[i]!=NULL && strcmp(k, keys[i]) == 0)
       return i;
  return -1;
}


char *query(char *k) {
  int i= getIdx(k);
  return (i==-1 ? NULL: defs[i]);
}


void delDef(char *k) {
  int i= getIdx(k);
  if (i!=-1) {
    keys[i]= NULL;
    defs[i]= NULL;
  }
}
```

EJEMPLO: DICCIONARIO CONCURRENTE

- Ejemplos de “data races” (interferencias entre tareas).

i) newDef con newDef

```
void newDef(char *k, char *d) {
  int i = getSlot();
  keys[i]= k;
  defs[i]= d;
}
```



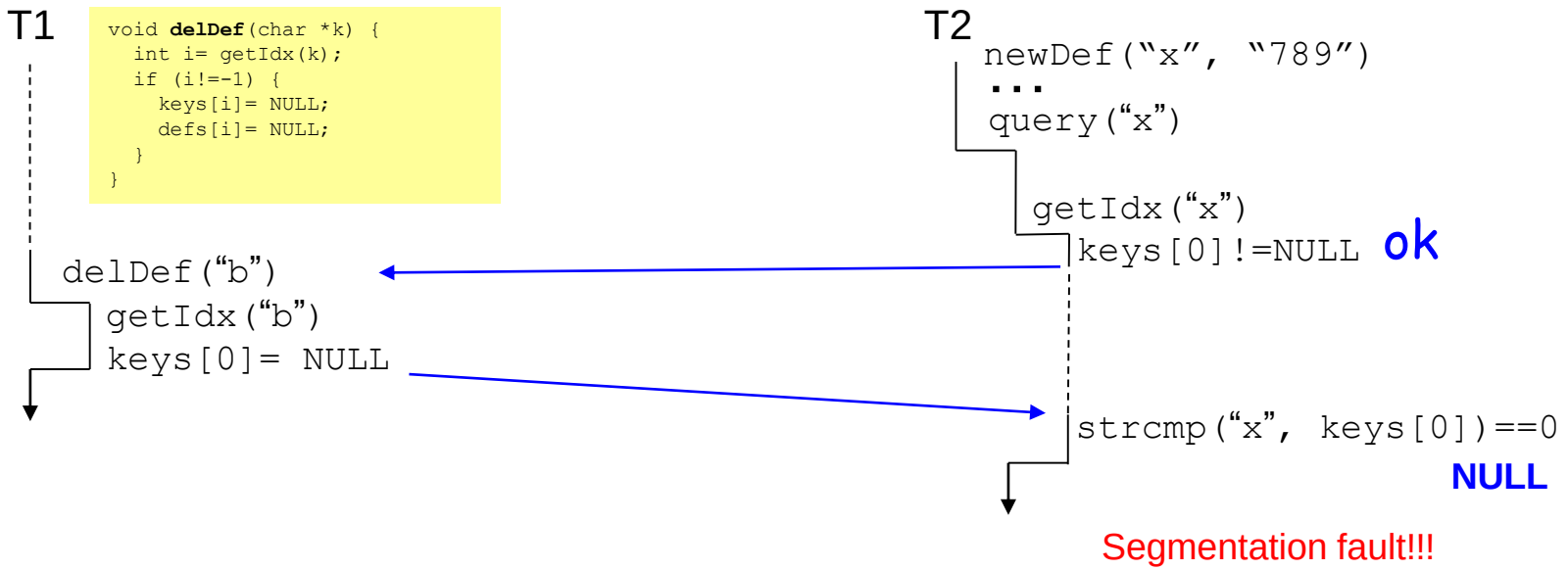
No es razonable!!!

EJEMPLO: DICCIONARIO CONCURRENTE

```
int getIdx(char *k) {
  int i;
  for(i=0; i<MAX; i++)
    if (keys[i]!=NULL && strcmp(k, keys[i]) == 0)
       return i;
  return -1;
}
```

ii) delDef con query

```
void delDef(char *k) {
  int i= getIdx(k);
  if (i!=-1) {
    keys[i]= NULL;
    defs[i]= NULL;
  }
}
```

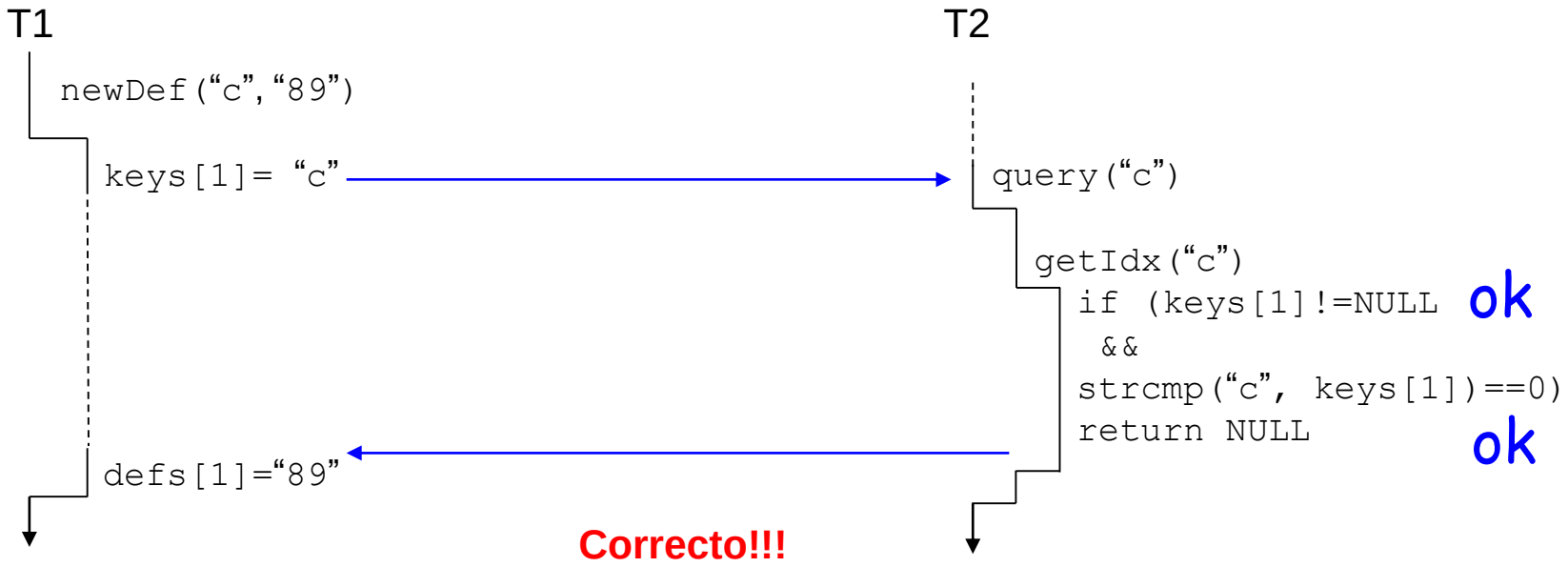


iii) query con query

No hay problema. Nunca hay problemas de datarace cuando no se modifica la estructura de datos compartida.

EJEMPLO: DICCIONARIO CONCURRENTE

iV) newDef con query




```
void newDef(char *k, char *d) {
  int i = getSlot();
  keys[i]= k;
  defs[i]= d;
}
```

```
char *query(char *k) {
  int i= getIdx(k);
  return (i==-1 ? NULL: defs[i]);
}
```

```
int getIdx(char *k) {
  int i;
  for(i=0; i<MAX; i++)
    if (keys[i]!=NULL && strcmp(k, keys[i]) == 0)
       return i;
  return -1;
}
```

EJEMPLO: DICCIONARIO CONCURRENTE

En general:

- 2 tareas no pueden modificar concurrentemente una misma estructura de datos
- 1 tarea no puede modificar una estructura de datos mientras otra la consulta
- 2 tareas si pueden consultar concurrentemente una misma estructura de datos.

EJEMPLO: DICCIONARIO CONCURRENTE

- Para evitar dataraces en el diccionario, usar:

```
Lock l;

void newDef(...) {
  lock(l);
  int i = getSlot();
  keys[i]= k;
  defs[i]= d;
  unlock(l);
}
```

```
int getIdx(char *k) {
  int i;
  for(i=0; i<MAX; i++)
    if (keys[i]!=NULL && strcmp(k, keys[i]) == 0)
       return i;
  return -1;
}
```

```
char *query(char *k) {
  char *d;
  lock(l);
  int i= getIdx(k);
  if (i==-1)
     d= NULL;
  else
     d= defs[i];
  unlock(l);
  return d;
}
```

```
void delDef(char *k) {
  lock(l);
  int i= getIdx(k);
  if (i!=-1) {
    keys[i]= NULL;
    defs[i]= NULL;
  }
  unlock(l);
}
```

En esta solución, ponemos los candados en la función que invoca a getIdx() debido a que dentro de ella corremos el riesgo de ejecutar la instrucción return antes de invocar a unlock() y dejar el candado cerrado para siempre,

DATARACE

- También llamado carrera de datos.
- Es uno de los errores más temibles de los programadores que trabajan en ambientes concurrentes.
- Es difícil de detectar
- Una de las mejores definiciones es la siguiente:

Datarace *corresponde al problema que se produce cuando dos o más tareas accesan simultáneamente las misma estructuras de datos compartidos y las dejan en un estado inconsistente.*

- Probablemente un programa con un datarace en alguna parte de su código se ejecutará sin problemas el **99%** de la veces, pero jamás sabremos cuando se dará el **1%** de error.