Contenido 5: Sincronización

Lesly Reyes
lareyes2@uc.cl

Álvaro Quintana
alvaroq@uc.cl

Departamento de Ciencias de la Computación
Pontificia Universidad Católica Chile

August 26, 2024

Contenido

Sincronización
- Threads y Multicores
- Race Condition
- Soluciones condiciones de carrera
- Sincronización por Hardware

Abstracciones de Sincronización

Problema del productor-consumidor
- Problema del bounded-buffer

Semáforos
- Implementación Semáforo

Variables de condición

Problema de los Lectores y escritores

Problema de los filósofos comensales

Material Adicional

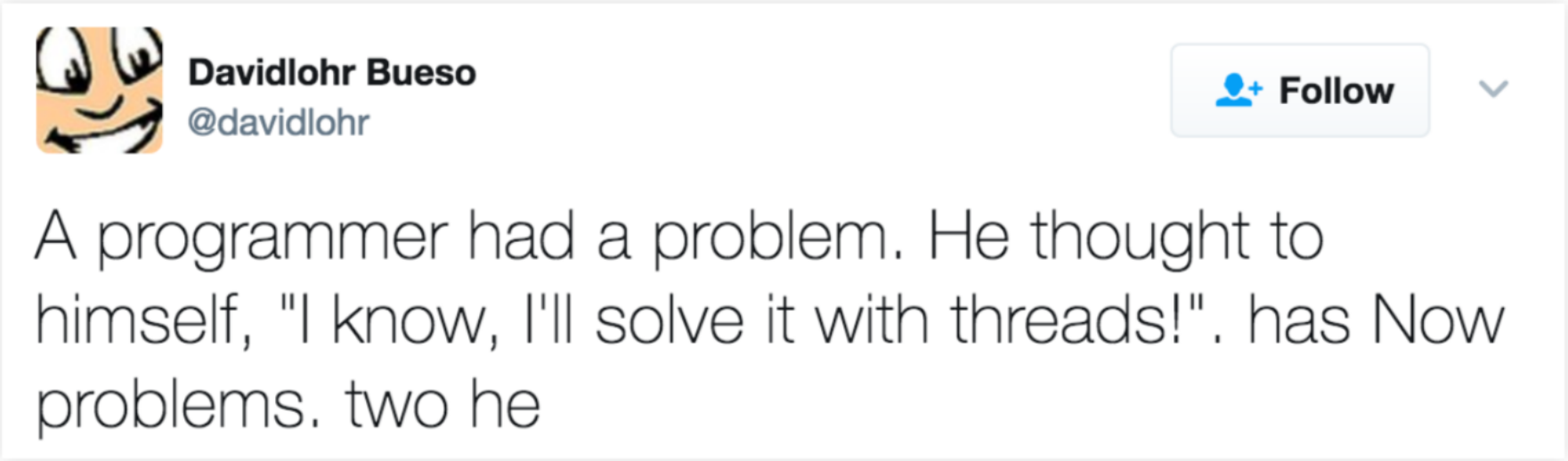
Sincronización

Threads como **unidad básica** de procesamiento

- Threads en un proceso comparten memoria
- Threads son concurrentes (compiten por uso de CPU)
- Threads en un sistema multicore pueden ejecutar en paralelo

Pero deben ser sincronizados... o puede pasar ...

Sincronización



Esta situación ilustra una **race condition**

- Situación en que la salida de una operación depende del orden temporal de sus operaciones internas, el cual no está bajo control del programador.
- El resultado depende de la entremezcla (interleaving) de sus threads

Hay que evitarlas

Sincronización

Sección Crítica

El segmento de código en que el thread accede a recursos compartidos se conoce como **sección crítica**

- Necesitamos un protocolo que no permita que dos o más threads se encuentren en su sección crítica al mismo tiempo

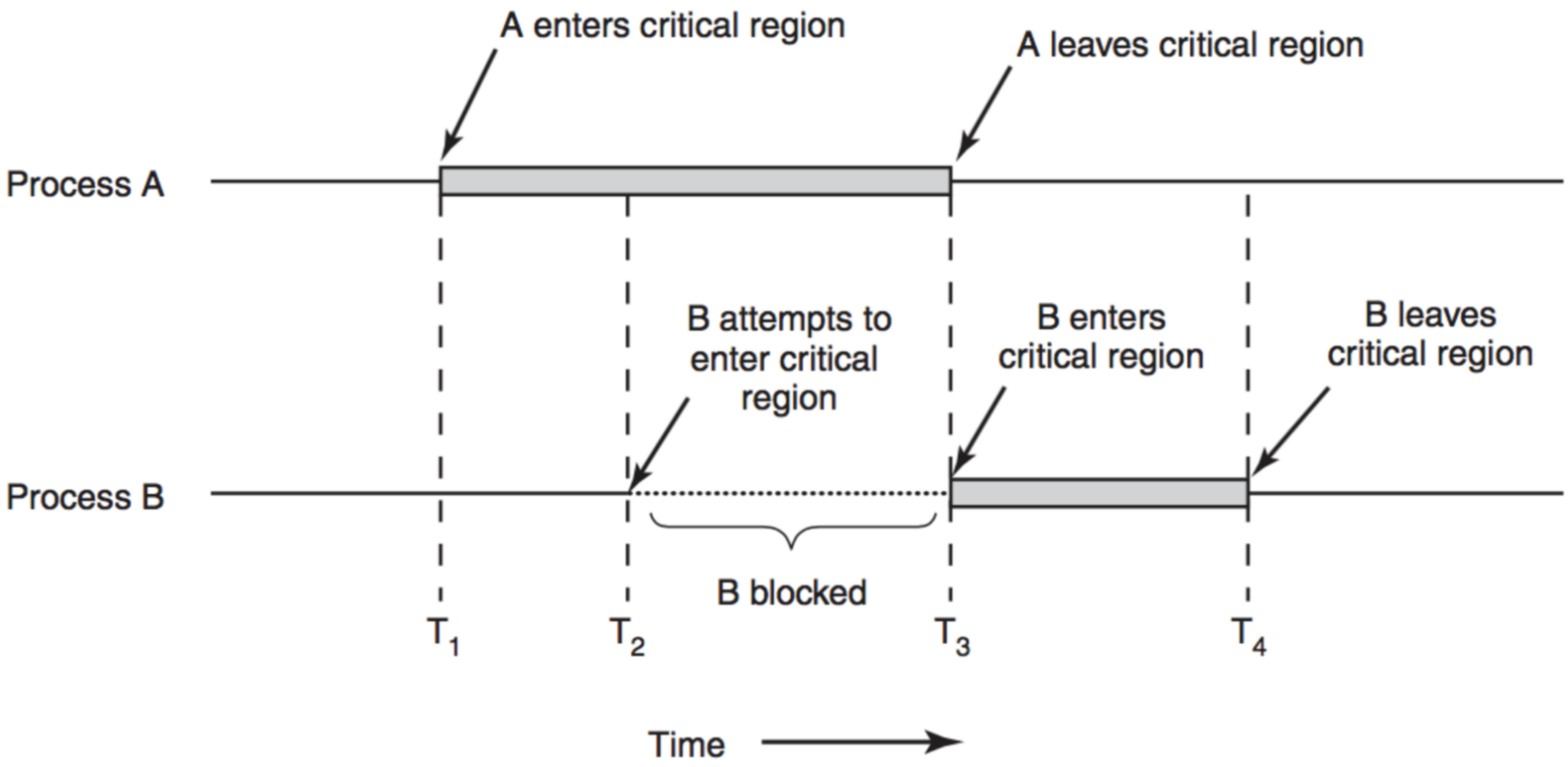
```
void mess_with_shared_things() {
  entry();
  /* do stuff with shared resources */
  leave();

  /* moar stuff */
}
```

Una solución al problema de la sección crítica (SC) debe cumplir:

- **Exclusión Mutua**: A lo más un thread está en su SC.
- **Progreso**: Al menos un thread puede entrar a su SC. Si ningún thread está en su SC, y hay threads que desean entrar, entonces los que quieren entrar deben decidir quién entra, y decidirlo en un tiempo acotado.
- **Espera acotada**(ausencia de inanición): Si un proceso quiere entrar a su SC, podrá hacerlo luego de una cantidad finita de tiempo.

Situación ideal



Atacar al problema de raíz: **Deshabilitar las interrupciones**

- Problema ocurre cuando un thread es interrumpido
- Si no hay interrupciones, no hay problema
- ¿Es suficiente?
- ¿Quién puede hacer esto?

Sincronización

Soluciones condiciones de carrera

En el mundo del usuario estamos a merced de las interrupciones
Solución con variable compartida lock

```
int lock = 0;  // 0==free, 1==busy
```

Código de cada thread:

```
while(TRUE) {
    while(lock);     // busy waiting (spinlock)
    lock = 1;
    /* ... SC ... */
    lock = 0;
    /* ... out of SC ... */
}
```

¿Problemas?

Sincronización

Solución con TURN

Solución con turn, para alternar entre dos threads
Variable compartida turn determina quién puede entrar.

```
int turn = 1;  // o 0 ...
```

Código para thread 0 y código para thread 1:

```
/* Thread 0 */
while(TRUE) {
    while(turn != 0);
    /* ... SC ... */
    turn = 1;
    /* ... out of SC ... */
}
```

```
/* Thread 1 */
while(TRUE) {
    while(turn != 1);
    /* ... SC ... */
    turn = 0;
    /* ... out of SC ... */
}
```

¿Cumple con las condiciones?

- Exclusión mutua: Sí
- Progreso: No
- Porque hay un thread que puede entrar y no quiere. Y otro que quiere entrar y no puede.

Sincronización

Solución de G.L.Peterson, 1981 (no la primera, pero la más simple)

Variable compartida **turn determina quién puede entrar.**

```
int turn = 1;                  // o 0 ...
int flag[2] = {false, false}; // indica t_i está interesado en entrar
```

Variable local me puede ser 0 ó 1:

```
int me = // ... 0 ó 1;    /* Peterson es para dos threads */
int other = 1-me;
while('o') {
    flag[me] = true;  /* wants */
    turn = other;
    while(flag[other] && turn == other); /* wait */
    /* ... SC ... */
    flag[me] = false;
    /* ... out of SC .../ */
}
```

¿Cumple los requisitos? ¿Por qué? ¿Se puede demostrar?

- Exclusión mutua: Sí
- Progreso: Sí
- Espera Acotada: Si un thread 1 quiere entrar va estar bloqueado, y solo va a poder entrar solo cuando el thread 0 salga de SC. Si el thread 0 es muy rápido va esperar infinito.
- Turn == other hace la magia

Soluciones por software dependen de que existan **instrucciones atómicas** provistas por el hardware.

```
int lock = 0;  // 0==free, 1==busy
```

```
while(TRUE) {
    while (lock);
    lock = 1;
    /* Critical Section */
    lock = 0;
    /* ... */
```

Esto no era suficiente. ¿Qué instruccion(es) nos gustaría que fuese(n) atómica(s)?

Sincronización

Sincronización por Hardware: *test_and_set*()(TSL)

test_and_set()(TSL)

Algunas arquitecturas proveen instrucciones atómicas del tipo *test_and_set*

La instrucción, que se ejecuta atómicamente, es equivalente a este código:

```
bool SOFTWARE_test_and_set (bool *target) {
    bool old_value = *target;
    *target = true;

    return old_value;
}
```

¿Cómo usarlo?

Sincronización

Sincronización por Hardware: *test_and_set*()(TSL)

Código equivalente a *test_and_set*

```
bool SOFTWARE_test_and_set (bool *target) {
    bool old_value = *target;
    *target = true;

    return old_value;
}
```

Ahora se puede implementar la versión "ingenua"

```
bool lock = false;  // false==free, true==busy
```

```
while(true) {
    /* ... ... ... */
    while (test_and_set(&lock));
    /* ... SC ... */
    lock = false;
    /* ... Out of SC ... */
}
```

Sincronización

Sincronización por Hardware: *compare_and_swap*() (XCHG, CAS)

Otra instrucción atómica *compare_and_swap*()(XCHG, CAS)

```
bool SOFTWARE_compare_and_swap (int *value, int expected, int new_value) {
  int previous = *value;

  if(*value == expected)
    *value = new_value;

  return previous;
}
```

¿Cómo usarlo?

Sincronización

Sincronización por Hardware: *compare_and_swap*() (XCHG, CAS)

Código equivalente a *compare_and_swap*()(XCHG, CAS)

```
bool SOFTWARE_compare_and_swap (int *value, int expected, int new_value) {
  int previous = *value;

  if(*value == expected)
    *value = new_value;

  return previous;
}
```

Otra implementación simple, gracias a una instrucción atómica

```
int lock = 0;  // 0==free, 1==busy
```

```
while(true) {
    /* ... ... ... */
    while (compare_and_swap(&lock, 0, 1));
    /* ... SC ... */
    lock = 0;
    /* ... Out of SC ... */
}
```

Abstracciones de Sincronización

En general no se usan directamente soluciones por hardware
Se construyen **primitivas de software** sobre ellas.

Abstracciones de Sincronización

Mutex Locks

La más simple: **MUT**ual **EX**clusion locks (mutex locks)

- Lock::acquire(); toma el lock
- Lock::release(); libera el lock (error si no está tomado)

```
struct Lock {
  bool value = false; // false==free, true==busy
}
```

Implementación sin (mucho) busy waiting

```
void acquire() {
  while(test_and_set(&value))
    thread_yield();            // otras implementaciones usan sleep() y colas de espera
}
void release() {
  value = false;
}
```

Abstracciones de Sincronización

Mutex Locks

Solución a Sección Crítica

```
while(true) {
    /* ... ... ... */
    lock.acquire();
    /* ... SC ... */
    lock.release();
    /* ... Out of SC ... */
}
```

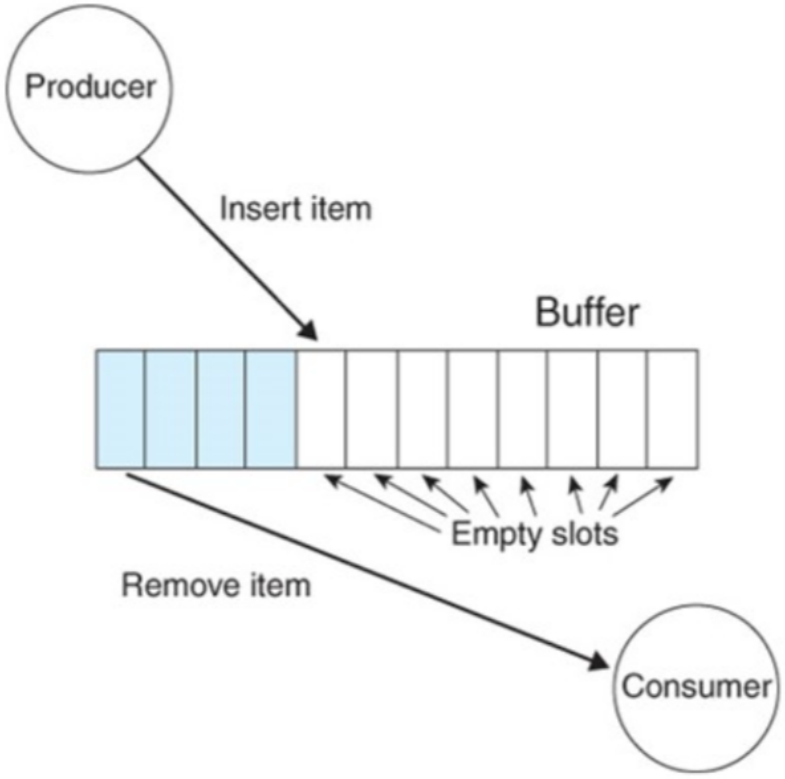
Contador con acceso exclusivo

```
struct Lock l; // compartido
```

```
/* ... ... ... */
for(int i=0; i<max; i++) {
  l.acquire();
  counter = counter + 1;
  l.release();
}
```

Problema del productor-consumidor

ó problema del bounded-buffer



Problema del productor-consumidor

ó problema del bounded-buffer

- Un buffer circular de tamaño limitado N
- Dos threads: productor (agrega) y consumidor (consume)
- Todos los elementos producidos deben ser consumidos en orden

```
int stock = 0;
T storage[N];
```

```
int in = 0;
while('p') {
    item = produce();
    while (stock == N);  // Storage full
    storage[in] = item;
    in = ++in%N;         // rotate index
    stock++;
}
```

```
int out = 0;
while ('c') {
    while(!stock);       // Wait for new items
    item = storage[out];
    out = ++out%N;       // rotate index
    stock--;
    consume(item);
} /* ' */
```

Problema del productor-consumidor

Podemos agregar locks en stock

```
int in = 0;
while('p') {
    item = produce();
    while (stock == N);  // Storage full
    storage[in] = item;
    in = ++in%N;         // rotate index
    lock.acquire();
    stock++;
    lock.release();
}
```

```
int out = 0;
while ('c') {
    while(!stock);       // Wait for new items
    item = storage[out];
    out = ++out%N;       // rotate index
    lock.acquire();
    stock--;
    lock.release();
    consume(item);
} /* ' */
```

¿Cómo eliminamos busy waiting?

Problema del productor-consumidor

- sleep() se bloquea en cola waiting
- wakeup(t) saca a t de la cola waiting (si está)

```
int in = 0;
while('p') {
    item = produce();
    while (stock == N) sleep(); // Storage full
    storage[in] = item;
    in = ++in%N;                // rotate index
    lock.acquire();
    stock++;
    lock.release();
    if(stock == 1) wakeup(consumer);
}
```

```
int out = 0;
while ('c') {
    while(!stock) sleep();      // Wait for new items
    item = storage[out];
    out = ++out%N;              // rotate index
    lock.acquire();
    stock--;
    lock.release();
    if(stock == N-1) wakeup(producer);
    consume(item);
} /* ' */
```

¿Problema? ¿Si el consumidor es interrumpido antes de dormir?

Semáforos

Primitiva de Sincronización

E.W.Dijkstra (1965) propone una nueva primitiva de sincronización: semáforos, y las introduce en Algol 68.

> Semáforos permiten un número limitado de threads en una sección crítica.
>
> **Un semáforo S** incluye un contador y dos operaciones:
>
> - **proberen P()** ó wait() ó down(). Intenta decrementar el valor.
> - **verhogen V()** ó signal() ó up(). Incrementa el valor.

E.W.Dijkstra era holandés, pero si no saben holandés es más fácil recordar Pedir/Votar

Semáforos

Implementación de semáforos

```
struct semaphore{
  int count;
  struct Lock l;
  struct process* slept = NULL;
}
```

```
void init(c) {
    count = c;
}

void P() {      /* wait: put into queue and sleep */
  l.acquire();
  while(count <= 0) {
      /* code to add itself to 'slept' */ // /
      l.release();
      sleep();
      l.acquire();
  }
  count--;
  l.release();
}
```

```
void V(S) {    /* signal */
  l.acquire();
  count++;
  wakeup(/* first from 'slept' */);
  // plus, remove from queue
  l.release();
}
```

Otras implementaciones usan busy waiting (para multiprocesadores)

Semáforos

Bounded-buffer con semáforos

```
struct lock l; int stock = 0; T storage[N];
struct semaphore full;  full.init(0);
struct semaphore empty; empty.init(N);
```

```
int in = 0;
while('p') {
    item = produce();
    empty.P();              // Wait one empty slot. P=='wait'=='down'
    l.acquire();
    storage[in] = item;
    in = ++in%N;                // rotate index
    stock++;
    lock.release();
    full.V();               // One more full slot. V=='signal'=='up'
}
```

```
int out = 0;
while ('c') {
    full.P()              // Wait one more item. P=='wait'=='down'
    lock.acquire();
    item = storage[out];
    out = ++out%N;              // rotate index
    stock--;
    lock.release();
    empty.V();            // One more empty slot. V=='signal'=='up'
    consume(item);
}
```

Variables de condición

Semáforos sirven para controlar cantidad de accesos a una región protegida

Mecanismo consisten en bloquearse (P()) bajo una condición (contador negativo) y avisar o señalar (V()) que una condición se ha cumplido (contador mayor a 0)

Variables de condición permitan bloquearse bajo condiciones arbitrarias.

- **Condition::wait()** bloquea siempre el thread
- **Condition::signal()** despierta a un thread bloqueado, si lo hay

Variables de condición

Bounded-buffer con condition variables

```
struct lock l;
struct condition condCons, condProd;
int stock = 0; T storage[N];
```

```
int in = 0;
while('p') {
    item = produce();
    l.acquire();                        // with mutex
    while(stock == N) condProd.wait(l); // wait if full
    storage[in] = item;  in = ++in%N;   // fill and rotate index
    stock++;
    condCons.signal();                  // signal new item for consumer
    l.release();
}
```

```
int out = 0;
while ('c') {
    l.acquire();
    while(!stock) condCons.wait(l);     // wait if empty
    item = storage[out]; out = ++out%N; // extract and rotate index
    stock--;
    condProd.signal();                  // signal new slot for producer
    l.release();
    consume(item);
}
```

Condition variables utilizan un lock y lo liberan antes de bloquearse.

Problema de los Lectores y escritores

Readers-Writers

Planteado por Courtois et al. (1971). Modela acceso a una base datos.

- Permitir acceso de n lectores simultáneos, pero solamente 1 escritor

```
struct semaphore rw; rw.init(1);
struct semaphore mutex; mutex.init(1);
int readers = 0;        // leen o quieren leer
```

Escritor (fácil) un semáforo **rw** compartido entre escritores y lectores

```
while ('r') {
    mutex.P();
    readers += 1;
    if(readers == 1)
        rw.P();
    mutex.V();
    /* ... READ ... */
    mutex.P();
    readers -= 1;
    if(readers == 0)
        rw.V();
    mutex.V();
}
```

Lectores un semáforo mutex compartido

```
while('w') {
    rw.P();
    /* ... WRITE ... */
    rw.V();
}
```

Problema de los filósofos comensales

5 filósofos se pasan la vida en dos actividades: comer y pensar

- Para comer deben tomar ambos palillos
- Sólo pueden tomar un palillo a la vez

Código para filósofo **i**

```
int left =  i;
int right = (i+1)%N;              // N==5
struct semaphore chopstick[N]; // inicializados en 1

do {
  chopstick[left].P();
  chopstick[right].P();

  /* eat */

  chopstick[left].V();
  chopstick[right].V();

  /* philosophise */

} while ('p');
```

Problema de los filósofos comensales

```
int N = 5;
enum {PHILOSOPHISE, HUNGRY, EATING} state[N];   /* free / waiting / using */
struct lock l;
struct semaphore philosopher[N]; // inicializados en 0
```

```
int left(i)  { return (i+N-1)%N; }
int right(i) { return (i+1)%N; }

void philosoph(int i) {
    while(TRUE) {
        think();
        take_forks();
        eat();
        release_forks();
    }
}

void take_forks(int i) {
    l.acquire();
    state[i] = HUNGRY;
    try_forks(i);
    l.release();
    philosopher[i].P();
}
```

```
void release_forks(int i) {
    l.acquire();
    state[i] = PHILOSOPHISE;
    try_forks(left(i));
    try_forks(right(i));
    l.release();
}

void try_forks(int i) {
    if(state[i] == HUNGRY && state[left(i)] != EATING
       && state[right] != EATING) {
        state[i] = EATING;
        philosopher[i].V();
    }
}
```

Material Adicional

- Video explicativo Sincronización y sección crítica: Link Video
- Video explicativo Soluciones para la sección crítica: Link Video
- Video explicativo Sincronización por Hardware: Link Video
- Video explicativo Primitivas de Sincronización Locks Link
- Video explicativo Primitivas de Sincronización Semáforos Link
- Video explicativo Primitivas de Sincronización Variables de condición Link

Próximo contenido a revisar será Administración de Memoria