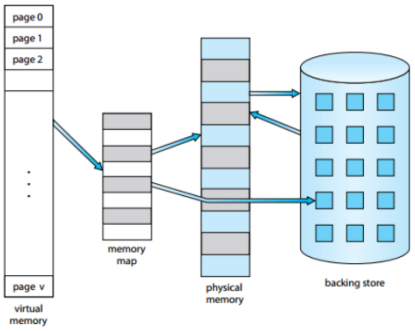
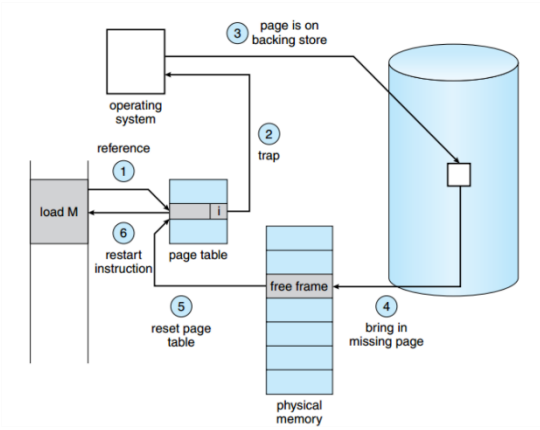
La virtualización de CPU y memoria es fundamental para la gestión eficiente de recursos en sistemas operativos, permitiendo la ejecución concurrente de múltiples hilos dentro de un mismo proceso. La creación y manejo de hilos es esencial para mejorar el rendimiento de aplicaciones, especialmente en entornos multicore.





Un proceso incluye:

- **CPU context** (Registers)
- **OS Resources** (Address Space, Open files, etc).
- **Otra información** (PID, state, owner, etc).

Un proceso con un solo hilo se caracteriza por que:

- Tiene un único hilo de ejecución (a cuyo estado se asocian los registros de la CPU: PC, SP, PGs, etc.).
- Un espacio de direcciones único (code, stack y heap).

La virtualización de CPU y memoria permite la abstracción de procesos y la gestión eficiente de recursos. Un proceso puede incluir múltiples hilos de ejecución, donde cada hilo comparte el mismo espacio de direcciones, lo que facilita la comunicación y la ejecución concurrente. La tendencia actual en CPUs es hacia más núcleos y la necesidad de escribir aplicaciones que aprovechen esta capacidad. Los hilos permiten la paralelización de tareas, mejorando el rendimiento en operaciones intensivas como la suma de arreglos grandes. Sin embargo, el uso de hilos también introduce desafíos como condiciones de carrera y la necesidad de gestionar secciones críticas. La API Pthreads proporciona herramientas para la creación y gestión de hilos, incluyendo funciones para crear, unir y manejar bloqueos. La exclusión mutua es crucial para evitar conflictos en el acceso a variables compartidas. La comprensión de estos conceptos es esencial para el desarrollo de aplicaciones eficientes en sistemas operativos modernos.

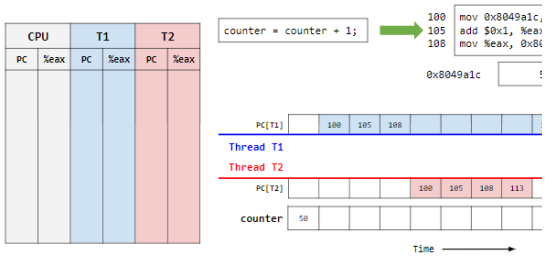
La virtualización de CPU y memoria permite la ejecución eficiente de múltiples hilos, mejorando el rendimiento de las aplicaciones. La gestión adecuada de hilos y la prevención de condiciones de carrera son esenciales para el desarrollo de software concurrente.

Virtualización de la memoria

La virtualización y el manejo de hilos son fundamentales para la eficiencia en sistemas operativos, permitiendo la ejecución concurrente y el acceso controlado a recursos compartidos. La implementación de locks es esencial para evitar condiciones de carrera y garantizar la exclusión mutua en secciones críticas.

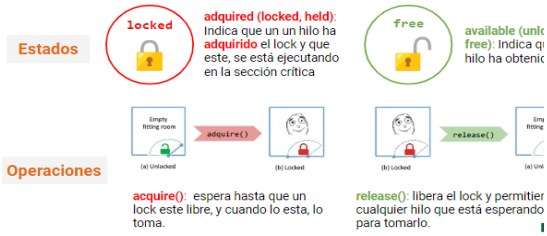
- Condición que se da cuando múltiples hilos de ejecución ingresan en la sección crítica a la misma vez.
- Cuando los hilos que se encuentran en la región crítica intentan modificar datos compartidos **los resultados son inesperados**.

Race condition (condición de carrera)



Locks

Un lock es un objeto (en memoria) que proporciona exclusión mutua.



Evaluación de locks

Para evaluar la efectividad en la implementación de un lock se utilizan las siguientes métricas:

Mutual exclusion (Exclusión mutua): Sólo un hilo en la sección crítica.

Fairness (Equidad): Cada hilo tiene igual probabilidad de adquirir el lock.

Performance (Desempeño): Relacionado con el time overhead (tiempo tomado en la adquisición/liberación con o sin competencia).

Implementación mediante el control de interrupciones:

- Deshabilitar interrupciones al entrar a las regiones críticas.
    - Cuando se deshabilitan las interrupciones se bloquean eventos externos que podrían generar un cambio de contexto.
    - El código dentro de la sección crítica no será interrumpido.
    - No hay un estado asociado al lock.

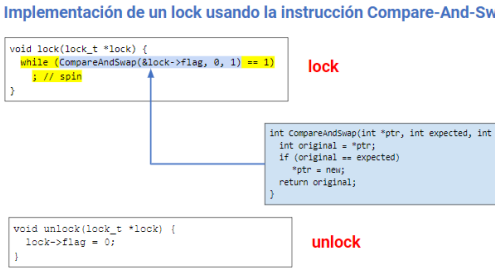
Implementación de un lock usando la instrucción test-and-set

```
typedef struct __lock_t {
  int flag;
} lock_t;

void init(lock_t *mutex) {
  // 0 -> lock is available, 1 -> held
  mutex->flag = 0;
}

void lock(lock_t *mutex) {
  while (mutex->flag == 1) // TEST the flag
  ; // spin-wait (do nothing)
  mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

```
typedef struct __lock_t {
  int flag;
} lock_t;

void init(lock_t *lock) {
  // 0: lock is available, 1: lock is held
  lock->flag = 0;
}

void lock(lock_t *lock) {
  while (TestAndSet(&lock->flag, 1) == 1)
  ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
  lock->flag = 0;
}
```

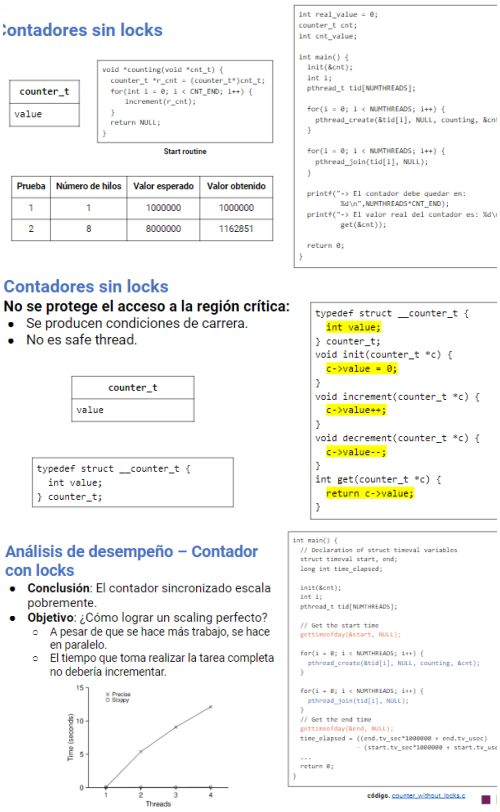
Implementación de un lock usando la instrucción Compare-And-Swap

- Esta instrucción atómica actualiza una localización de memoria con un valor anterior sólo cuando este es igual al esperado. (x86: cmpxchg).
- Esta instrucción trabaja de manera similar al test-and-set pero es más poderosa.



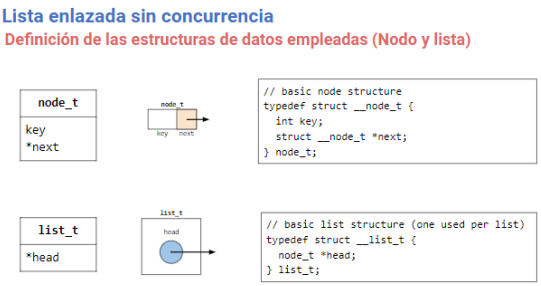
LL (Load linked): Opera como una instrucción load típica trayendo simplemente un valor de memoria y poniéndolo en un registro.
SC (Store conditional): El almacenamiento solo funciona si no se ha realizado otro almacenamiento previo en el mismo lugar:

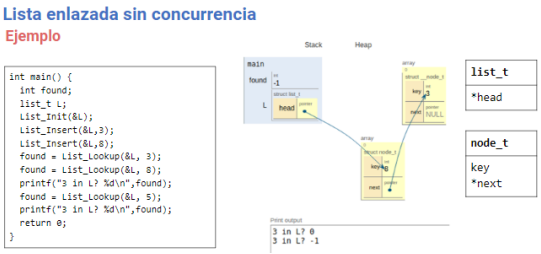
Controladores concurrentes:



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).

Listas enlazadas concurrentes:





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.

Colas concurrentes:

- **Se usan dos locks (Michael and Scott)**
    - Uno para el nodo cabeza (head) de la cola.
    - Otro para el nodo final (tail) de la cola.
    - El objetivo es habilitar concurrencia para encolar y desencolar elementos.
- **Se incluye un nodo Dummy**
    - Asignado en el código de inicialización de la cola.
    - Permite separación de operaciones en la cabeza y el final de la cola.

Operaciones básicas sobre la cola concurrente

Inicialización: **Queue_Init**

```
void Queue_Init(queue_t *q) {
  node_t *tmp = malloc(sizeof(node_t));
  tmp->next = NULL;
  q->head = q->tail = tmp;
  pthread_mutex_init(&q->head_lock, NULL);
  pthread_mutex_init(&q->tail_lock, NULL);
}
```

Encolar: **Queue_Enqueue**

```
void Queue_Enqueue(queue_t *q, int value) {
  node_t *tmp = malloc(sizeof(node_t));
  assert(tmp != NULL);
  tmp->value = value;
  tmp->next = NULL;

  pthread_mutex_lock(&q->tail_lock);
  q->tail->next = tmp;
  q->tail = tmp;
  pthread_mutex_unlock(&q->tail_lock);
}
```

node_t
key
*next

queue_t
*head
head_lock
*tail
head_lock

Desencolar: **Queue_Dequeue**

```
int Queue_Dequeue(queue_t *q, int *value) {
  pthread_mutex_lock(&q->head_lock);
  node_t *tmp = q->head;
  node_t *new_head = tmp->next;
  if (new_head == NULL) {
    pthread_mutex_unlock(&q->head_lock);
    return -1; // queue was empty
  }
  *value = new_head->value;
  q->head = new_head;
  pthread_mutex_unlock(&q->head_lock);
  free(tmp);
  return 0;
}
```

Variables de condición:

Las variables de condición (CV) se usan para esperar hasta que alguna variable cumpla alguna condición (evento).
Variable de condición (CV): Cola de hilos en espera.

Operaciones de las variables de condición

`wait(cond_t *cv, mutex_t *mutex)`

- Un hilo se pone en una cola (a dormir) a la espera de que un estado deseado suceda.
- Recibe un mutex como parámetro.
- Cuando se llama el wait el lock es liberado y se pone el hilo a dormir.

`signal(cond_t *cv)`

- Despierta un único hilo de los que se encuentran en la cola de espera (**if >= 1** el hilo está esperando).
- Si no hay hilos esperando, solo retorna sin hacer nada.
- Cuando el hilo se despierte se debe adquirir de nuevo el lock.

Implementación de Join:

Padre	Hijo	Anotaciones
`Pthread_create(&p, NULL, child, NULL);`		Crea el hilo hijo
	`call child()`	El Hilo hijo empieza su ejecución llamando la función **child()**
	`printf("child\n");`	Se imprime el mensaje: **child**
	`call thr_exit();`	Llama función thr_exit() para despertar al hijo padre
	`pthread_mutex_lock(&m);`	Obtiene el lock.
	`done = 1;`	Se fija el valor de la variable done (done = 1)
	`pthread_cond_signal(&c);`	Llama signal para despertar al hilo padre. Como este no esta dormido, no hace nada, solo retorna.
	`Pthread_mutex_unlock(&m);`	Se libera el lock. Luego se pasa la ejecución al hilo padre.
`call thr_join();`		El hilo padre llama la función join
`pthread_mutex_lock(&m);`		El padre adquiere el lock. Tengase en cuenta que el lock se encuentra en 1 por lo que no se mete al ciclo que pone a dormir al hilo
`pthread_mutex_unlock(&m);`		Libera el lock.
`printf("parent: end\n");`		Imprime el mensaje **parent**

Semáforos:

Primitiva de sincronización inventada por Dijkstra en 1968.
Objeto con un valor entero no negativo (asociado al estado).

Un **semáforo**, solo puede ser ser manipulado por a través de dos operaciones.

`sem_wait()`: También conocida como **`wait()`**, **`P()`** o **`down()`**.

```
wait(S) {
  while (S <= 0)
    ; // busy wait
  S--;
}
```

`sem_post()`: También conocida como **`signal()`**, **`V()`** o **`up()`**.

```
signal(S) {
  S++;
}
```

Operaciones con semáforos:

Inicialización:

Antes de interactuar con cualquier semáforo es necesario declararlo e inicializarlo.

1. Declaración del semáforos
2. Inicialización del semáforo:
   El valor inicial es asignado a 1 (tercer argumento)
   El segundo argumento indica que el semáforo es compartido entre los hilos de un mismo proceso.

wait o test:

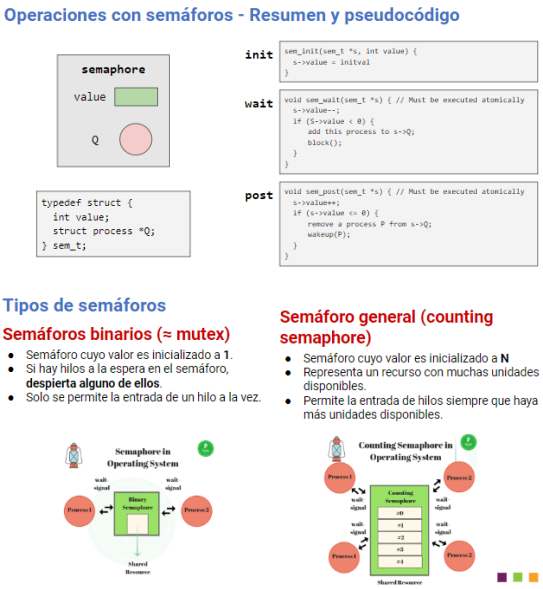
Si el valor del semáforo es:

- **Mayor o igual a uno (>= 1)**, la función retorna inmediatamente.

En caso contrario, el hilo llamador se suspende (wait) a la espera de una señal de post.

Post o signal o increment

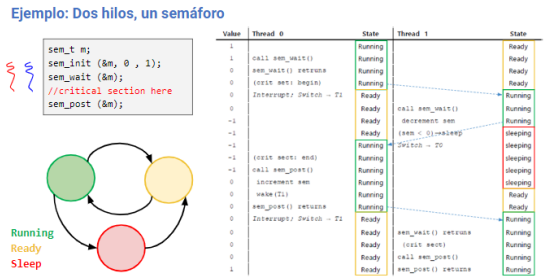
- **Incrementa** el valor del semáforo
- Si hay hilos a la espera en el semáforo, **despierta alguno de ellos**.



Semáforos binarios(locks):

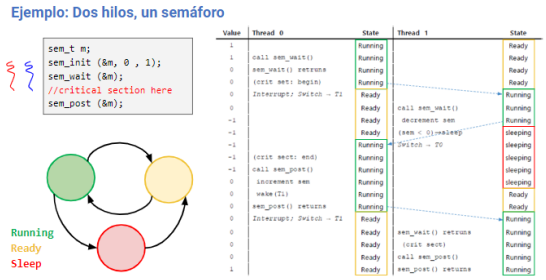
Un semáforo puede ser usado como lock.
La sección crítica va entre las operaciones wait() y post()
Escenario con dos hilos (Thread 0 y Thread 1)
De los dos hilos, solo uno de estos esta usa el semáforo.

- Escenario con dos hilos (**Thread 0** y **Thread 1**)
- De los dos hilos:
  - **Thread 0**: Ha llamado a **sem_wait()** pero aún no ha llamado a **sem_post()** - **holds the lock**.
  - **Thread 1:** Intenta ingresar en la **región crítica** llamando a **sem_post()**.



Semáforos de condición:

Los semáforos también son útiles para ordenar eventos en un programa concurrente lo que los hace aptos para ser usados como primitiva para controlar el orden de ejecución.



Los **semáforos** también son útiles para **ordenar eventos** en un programa concurrente lo que los hace aptos para ser usados como **primitiva para controlar el orden de ejecución**.