Ayudantía 7
Concurrencia

Javier Larré

Pontificia Universidad Católica de Chile
javier.larre@uc.cl

15 de mayo de 2026

Ayudantía de hoy

1. Recordatorios de la T2
2. Repaso de Concurrencia
   - Ejercicios de la semana pasada
   - Ejercicios de esta ayudantía
3. Monitors
4. Compare And Swap
5. Coroutines
6. Ejercicios propuestos
7. Referencing

Recordatorios de la T2

- **Recordatorios sobre la T2:**



Recordatorios de la T2

- **Recordatorios sobre la T2:**
- ¿Cómo van?.



Recordatorios de la T2

- **Recordatorios sobre la T2:**
- ¿Cómo van?.
- Cada parte de la tarea vale 50% de la nota.



Recordatorios de la T2

- **Recordatorios sobre la T2:**
- ¿Cómo van?.
- Cada parte de la tarea vale 50% de la nota.
- No pueden usar loops `for` ni `while` **solamente** en la primera pregunta de la primera parte. Pueden usar loops en el resto de la tarea.



Recordatorios de la T2

- **Recordatorios sobre la T2:**
- ¿Cómo van?.
- Cada parte de la tarea vale 50% de la nota.
- No pueden usar loops `for` ni `while` **solamente** en la primera pregunta de la primera parte. Pueden usar loops en el resto de la tarea.
- Publicaremos más tests durante este fin de semana. (aprovecharé de actualizar el enunciado para incluir estas consideraciones)

Recordatorios de la T2

- **Recordatorios sobre la T2:**
- ¿Cómo van?.
- Cada parte de la tarea vale 50% de la nota.
- No pueden usar loops `for` ni `while` **solamente** en la primera pregunta de la primera parte. Pueden usar loops en el resto de la tarea.
- Publicaremos más tests durante este fin de semana. (aprovecharé de actualizar el enunciado para incluir estas consideraciones)
- Pueden editar los archivos que vienen con la tarea, pero cuiden que su tarea compile junto con los tests.

Recordatorios de la T2

- **Recordatorios sobre la T2:**
- ¿Cómo van?.
- Cada parte de la tarea vale 50% de la nota.
- No pueden usar loops `for` ni `while` **solamente** en la primera pregunta de la primera parte. Pueden usar loops en el resto de la tarea.
- Publicaremos más tests durante este fin de semana. (aprovecharé de actualizar el enunciado para incluir estas consideraciones)
- Pueden editar los archivos que vienen con la tarea, pero cuiden que su tarea compile junto con los tests.
- Por favor, no copien y peguen de ChatGPT. Si lo hacen serán penalizados.

Recordatorios de la T2

- **Recordatorios sobre la T2:**
- ¿Cómo van?.
- Cada parte de la tarea vale 50% de la nota.
- No pueden usar loops `for` ni `while` **solamente** en la primera pregunta de la primera parte. Pueden usar loops en el resto de la tarea.
- Publicaremos más tests durante este fin de semana. (aprovecharé de actualizar el enunciado para incluir estas consideraciones)
- Pueden editar los archivos que vienen con la tarea, pero cuiden que su tarea compile junto con los tests.
- Por favor, no copien y peguen de ChatGPT. Si lo hacen serán penalizados.
- Pueden usar las librerías built in que quieran siempre y cuando no tengan que descargarlas externamente.

Repaso de Concurrencia



Ejercicios de concurrencia

- Describe el problema productor-consumidor.
- Cuáles son los 5 estados en el que puede estar un thread?
- Cuál es la diferencia entre concurrencia y paralelismo.
- Cuál es el problema de *busy waiting*?
- Qué es lo que indica la variable `counter` en un semáforo?
- Escribe en pseudocódigo las 2 operaciones de un semáforo (wait y release)
- Qué desventaja trae usar semáforos?
- Qué ocurriría si las operaciones de un semáforo fueran **no atómicas**?

Repaso de Concurrencia

Ejercicios de concurrencia

- Describe el problema productor-consumidor.

Ejercicios de concurrencia

- Describe el problema productor-consumidor.
  1. Es un problema de coordinación entre dos programas (u otro tipo de concurrencia) donde uno usa los datos o recursos producidos por otro programa.

Repaso de Concurrencia

Ejercicios de concurrencia

- Describe el problema productor-consumidor.
  1. Es un problema de coordinación entre dos programas (u otro tipo de concurrencia) donde uno usa los datos o recursos producidos por otro programa.
  2. Los datos son puestos por el productor en un buffer (cualquier tipo de estructura) compartido y retirados por el consumidor.

Repaso de Concurrencia

Ejercicios de concurrencia

- Describe el problema productor-consumidor.
  1. Es un problema de coordinación entre dos programas (u otro tipo de concurrencia) donde uno usa los datos o recursos producidos por otro programa.
  2. Los datos son puestos por el productor en un buffer (cualquier tipo de estructura) compartido y retirados por el consumidor. Estas operaciones deben ser atómicas.

Repaso de Concurrencia

Ejercicios de concurrencia

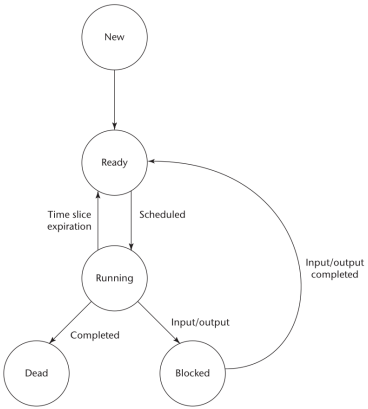
- Describe el problema productor-consumidor.
  1. Es un problema de coordinación entre dos programas (u otro tipo de concurrencia) donde uno usa los datos o recursos producidos por otro programa.
  2. Los datos son puestos por el productor en un buffer (cualquier tipo de estructura) compartido y retirados por el consumidor. Estas operaciones deben ser atómicas.
  3. El consumidor debe prevenir *poppear* del buffer si está vacío y viceversa con el productor.

Repaso de Concurrencia

Ejercicios de concurrencia

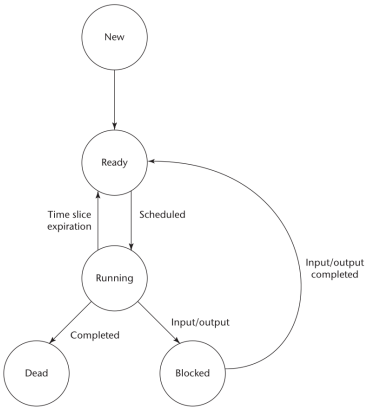
- Describe el problema productor-consumidor.
  1. Es un problema de coordinación entre dos programas (u otro tipo de concurrencia) donde uno usa los datos o recursos producidos por otro programa.
  2. Los datos son puestos por el productor en un buffer (cualquier tipo de estructura) compartido y retirados por el consumidor. Estas operaciones deben ser atómicas.
  3. El consumidor debe prevenir *poppear* del buffer si está vacío y viceversa con el productor.
- Cuáles son los 5 estados en el que puede estar un thread?

Repaso de Concurrencia



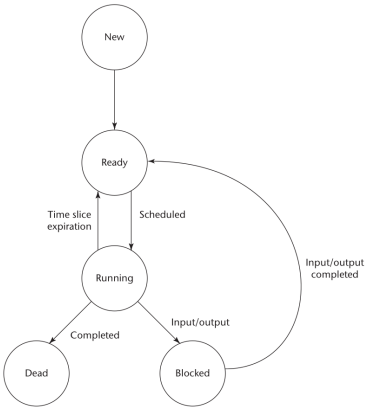
- ¿Cuáles son los 5 estados en el que puede estar un thread?

Repaso de Concurrencia



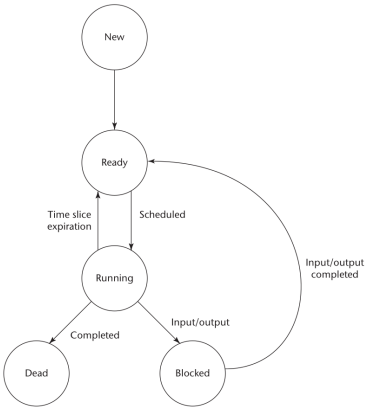
- ¿Cuáles son los 5 estados en el que puede estar un thread?
- Cuál es la diferencia entre concurrencia y paralelismo.

Repaso de Concurrencia



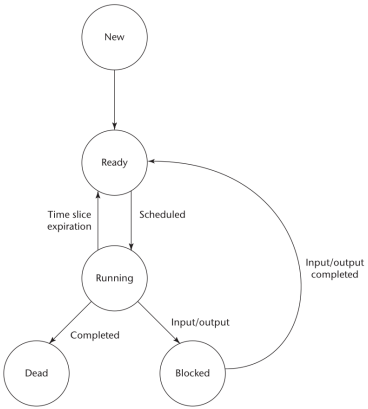
- ¿Cuáles son los 5 estados en el que puede estar un thread?
- Cuál es la diferencia entre concurrencia y paralelismo.
  Paralelismo es cuando un programa usa más de un core de CPU, mientras que la concurrencia permite tener tareas independientes de si misma sin necesidad de múltiples cores.

Repaso de Concurrencia



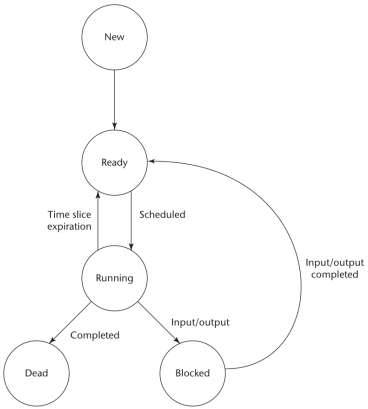
- ¿Cuáles son los 5 estados en el que puede estar un thread?
- Cuál es la diferencia entre concurrencia y paralelismo.
  Paralelismo es cuando un programa usa más de un core de CPU, mientras que la concurrencia permite tener tareas independientes de si misma sin necesidad de múltiples cores.
- ¿Cuál es el problema de *busy waiting*?

Repaso de Concurrencia



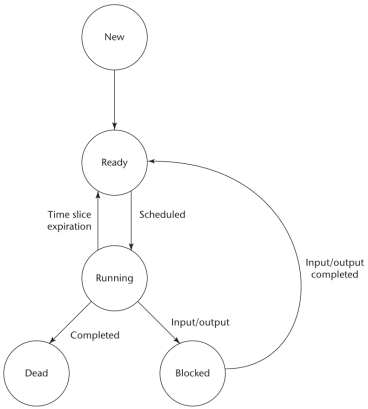
- ¿Cuáles son los 5 estados en el que puede estar un thread?
- Cuál es la diferencia entre concurrencia y paralelismo.
  Paralelismo es cuando un programa usa más de un core de CPU, mientras que la concurrencia permite tener tareas independientes de si misma sin necesidad de múltiples cores.
- ¿Cuál es el problema de *busy waiting*?
  Una tarea consultando constantemente consume tiempo de CPU que puede ser usado por otra tarea.

Repaso de Concurrencia



- ¿Cuáles son los 5 estados en el que puede estar un thread?
- Cuál es la diferencia entre concurrencia y paralelismo.
  Paralelismo es cuando un programa usa más de un core de CPU, mientras que la concurrencia permite tener tareas independientes de si misma sin necesidad de múltiples cores.
- ¿Cuál es el problema de *busy waiting*?
  Una tarea consultando constantemente consume tiempo de CPU que puede ser usado por otra tarea.
- ¿Qué es lo que indica la variable `counter` en un semáforo?

Repaso de Concurrencia



- ¿Cuáles son los 5 estados en el que puede estar un thread?
- Cuál es la diferencia entre concurrencia y paralelismo.
  Paralelismo es cuando un programa usa más de un core de CPU, mientras que la concurrencia permite tener tareas independientes de si misma sin necesidad de múltiples cores.
- ¿Cuál es el problema de *busy waiting*?
  Una tarea consultando constantemente consume tiempo de CPU que puede ser usado por otra tarea.
- ¿Qué es lo que indica la variable `counter` en un semáforo?
  La cantidad de procesos que pueden entrar al recurso crítico.

Repaso de Concurrencia

Escribe en pseudocódigo las dos operaciones de un semáforo (wait y release)

Repaso de Concurrencia

Escribe en pseudocódigo las dos operaciones de un semáforo (wait y release)

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
      decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task-ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
      increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

Repaso de Concurrencia

Escribe en pseudocódigo las dos operaciones de un semáforo (wait y release)

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
      decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task-ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
      increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

Fun fact: Estas operaciones fueron originalmente nombradas P y V por Dijkstra a partir de las palabras *passeren* y *vrygeren* que significan *to pass* y *to release*

Repaso de Concurrencia

Escribe en pseudocódigo las dos operaciones de un semáforo (wait y release)

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
      decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task-ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
      increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

Fun fact: Estas operaciones fueron originalmente nombradas P y V por Dijkstra a partir de las palabras *passeren* y *vrygeren* que significan *to pass* y *to release*

- Qué desventaja o problema trae usar semáforos?

Repaso de Concurrencia

Escribe en pseudocódigo las dos operaciones de un semáforo (wait y release)

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
    decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task-ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
    increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

Fun fact: Estas operaciones fueron originalmente nombradas P y V por Dijkstra a partir de las palabras *passeren* y *vrygeren* que significan *to pass* y *to release*

- Qué desventaja o problema trae usar semáforos?
  Aún así puede suceder **Deadlock**.

Repaso de Concurrencia

Escribe en pseudocódigo las dos operaciones de un semáforo (wait y release)

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
    decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task-ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
    increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

Fun fact: Estas operaciones fueron originalmente nombradas P y V por Dijkstra a partir de las palabras *passeren* y *vrygeren* que significan *to pass* y *to release*

- Qué desventaja o problema trae usar semáforos?
  Aún así puede suceder **Deadlock**.
- Qué ocurriría si las operaciones de un semáforo fueran **no atómicas**?

Repaso de Concurrencia

Escribe en pseudocódigo las dos operaciones de un semáforo (wait y release)

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
    decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task-ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
    increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

Fun fact: Estas operaciones fueron originalmente nombradas P y V por Dijkstra a partir de las palabras *passeren* y *vrygeren* que significan *to pass* y *to release*

- Qué desventaja o problema trae usar semáforos?
  Aún así puede suceder **Deadlock**.
- Qué ocurriría si las operaciones de un semáforo fueran **no atómicas**?
  Se puede romper la propiedad de *safeness*. Si la cola de procesos no es *thread safe* entonces se va a corromper.

Repaso de Concurrencia

Escribe en pseudocódigo las dos operaciones de un semáforo (wait y release)

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
    decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task-ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
    increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

Fun fact: Estas operaciones fueron originalmente nombradas P y V por Dijkstra a partir de las palabras *passeren* y *vrygeren* que significan *to pass* y *to release*

¿Qué problema hay en esta implementación de u semáforo?

```
struct semaphore access, fullspots, emptyspots;
access.count = 1;
fullspots.count = 0;
emptyspots.count = BUFLEN;

void producer(){
  while (true) {
    wait(emptyspots);
    wait(access);
    DEPOSIT(VALUE);
    release(access);
    release(fullspots);
  }
}

void consumer(){
  while (true) {
    wait(fullspots);
    wait(access);
    FETCH(VALUE);
    wait(access);
    release(emptyspots);
  }
}
```

Repaso de Concurrencia

```
struct semaphore access, fullspots, emptyspots;
access.count = 1;
fullspots.count = 0;
emptyspots.count = BUFLEN;

void producer(){
  while (true) {
    wait(emptyspots); // wait for a space
    wait(access); // wait for access
    DEPOSIT(VALUE);
    release(access); // relinquish access
    release(fullspots); // increase filled spac
  }
}

void consumer(){
  while (true) {
    wait(fullspots); // make sure it is not emp
    wait(access); // wait for access
    FETCH(VALUE);
    release(access); // relinquish access
    release(emptyspots); // increase empty spac
  }
}
```

¿Qué problema hay en esta implementación de u semáforo?

```
struct semaphore access, fullspots, emptyspots;
access.count = 1;
fullspots.count = 0;
emptyspots.count = BUFLEN;

void producer(){
  while (true) {
    wait(emptyspots);
    wait(access);
    DEPOSIT(VALUE);
    release(access);
    release(fullspots);
  }
}

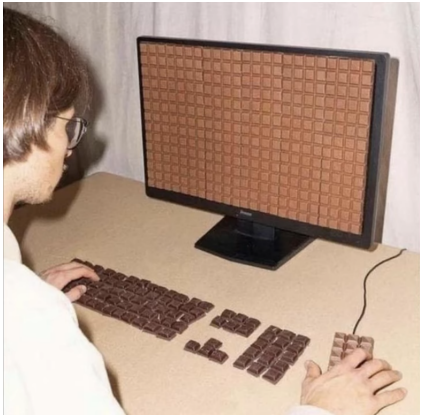
void consumer(){
  while (true) {
    wait(fullspots);
    wait(access);
    FETCH(VALUE);
    wait(access);
    release(emptyspots);
  }
}
```

Monitors

Quieres que una tabla de hash pueda ser compartida por varios thread concurrentes (muchos, más de los que imaginas) entonces implementas las operaciones `get_item` y `set_item` de tal manera que son atómicas.
Ahora solo queda sincronizar el acceso para proteger la tabla: podrías usar un lock entero para toda la tabla o ingeniar una estrategia utilizando varios locks. ¿Cuál estrategia funcionará mejor? ¿Cuáles pueden ser nuestros supuestos?

Monitors

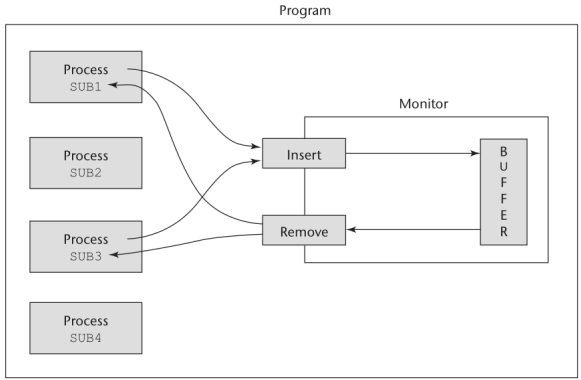
Monitors!



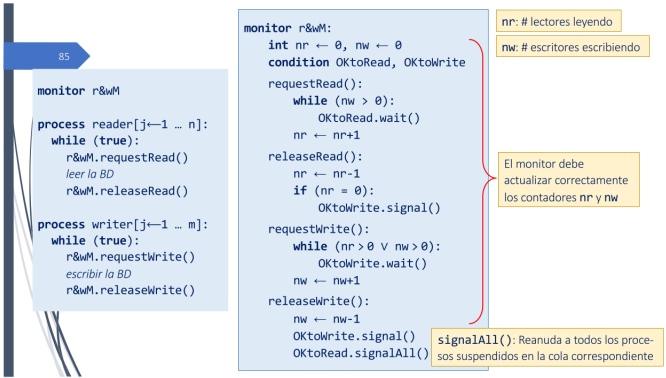
Monitors!

Figure 13.3

A program using a monitor to control access to a shared buffer



Monitors!


85
monitor r&wM
process reader[j←1 … n]:
while (true):
r&wM.requestRead()
leer la BD
r&wM.releaseRead()
process writer[j←1 … m]:
while (true):
r&wM.requestWrite()
escribir la BD
r&wM.releaseWrite()
monitor r&wM:
int nr ← 0, nw ← 0
condition OKtoRead, OKtoWrite
requestRead():
while (nw > 0):
OKtoRead.wait()
nr ← nr+1
releaseRead():
nr ← nr-1
if (nr = 0):
OKtoWrite.signal()
requestWrite():
while (nr > 0 ∨ nw > 0):
OKtoWrite.wait()
nw ← nw+1
releaseWrite():
nw ← nw-1
OKtoWrite.signal()
OKtoRead.signalAll()
nr: # lectores leyendo
nw: # escritores escribiendo
El monitor debe actualizar correctamente los contadores nr y nw
signalAll(): Reanuda a todos los procesos suspendidos en la cola correspondiente

Monitors

- Queremos permitir acceder a la tabla siempre y cuando no esté siendo editada.

Monitors

- Queremos permitir acceder a la tabla siempre y cuando no esté siendo editada.
- Los *bucket*s son independientes entre sí, por lo que las ediciones en uno no deberían afectar otro.

Monitors

- Queremos permitir acceder a la tabla siempre y cuando no esté siendo editada.
- Los *bucket*s son independientes entre sí, por lo que las ediciones en uno no deberían afectar otro.
- El monitor debe tener almacenado la tabla entera y manejar todos los locks, o bien que cada *bucket* de la tabla sea un monitor que almacena una lista.

Monitors

- Queremos permitir acceder a la tabla siempre y cuando no esté siendo editada.
- Los *bucket*s son independientes entre sí, por lo que las ediciones en uno no deberían afectar otro.
- El monitor debe tener almacenado la tabla entera y manejar todos los locks, o bien que cada *bucket* de la tabla sea un monitor que almacena una lista.
- ¿Cuáles serían las variables condiciones y la invarianza de nuestro monitor/es?

Monitors

- Queremos permitir acceder a la tabla siempre y cuando no esté siendo editada.
- Los *bucket*s son independientes entre sí, por lo que las ediciones en uno no deberían afectar otro.
- El monitor debe tener almacenado la tabla entera y manejar todos los locks, o bien que cada *bucket* de la tabla sea un monitor que almacena una lista.
- ¿Cuáles serían las variables condiciones y la invarianza de nuestro monitor/es?

Monitors

- Queremos permitir acceder a la tabla siempre y cuando no esté siendo editada.
- Los *bucket*s son independientes entre sí, por lo que las ediciones en uno no deberían afectar otro.
- El monitor debe tener almacenado la tabla entera y manejar todos los locks, o bien que cada *bucket* de la tabla sea un monitor que almacena una lista.
- ¿Cuáles serían las variables condiciones y la invarianza de nuestro monitor/es?

Hint

¿Cuántos writers y readers queremos permitir?

Monitors

- Queremos permitir acceder a la tabla siempre y cuando no esté siendo editada.
- Los *bucket*s son independientes entre sí, por lo que las ediciones en uno no deberían afectar otro.
- El monitor debe tener almacenado la tabla entera y manejar todos los locks, o bien que cada *bucket* de la tabla sea un monitor que almacena una lista.
- ¿Cuáles serían las variables condiciones y la invarianza de nuestro monitor/es?

Hint

¿Cuántos writers y readers queremos permitir?

Variables condiciones

Nuestras variables condiciones pueden ser `okToRead` y `okToWrite`

Monitors

- Queremos permitir acceder a la tabla siempre y cuando no esté siendo editada.
- Los *bucket*s son independientes entre sí, por lo que las ediciones en uno no deberían afectar otro.
- El monitor debe tener almacenado la tabla entera y manejar todos los locks, o bien que cada *bucket* de la tabla sea un monitor que almacena una lista.
- ¿Cuáles serían las variables condiciones y la invarianza de nuestro monitor/es?

Hint

¿Cuántos writers y readers queremos permitir?

Variables condiciones

Nuestras variables condiciones pueden ser `okToRead` y `okToWrite`

invarianza

$nWriters > 0 \iff nReaders < 1 \land nWriters = 1$

Compare And Swap (CAS)



Compare And Swap

¿Es posible realizar la operación `x = edit(x)` de una manera `thread safe`?

[1] Esto fue la motivación principal para introducir instrucciones a nivel de assembly que permita CAS

Compare And Swap

¿Es posible realizar la operación `x = edit(x)` de una manera `thread safe`? ¿sin locks?

[1] Esto fue la motivación principal para introducir instrucciones a nivel de assembly que permita CAS

Compare And Swap

¿Es posible realizar la operación `x = edit(x)` de una manera `thread safe`? ¿sin locks?

```
start:
    r1 := x
    r2 := foo(r1)
    r2 := CAS(x, r1, r2)
    if !r2 goto start
```

[1] Esto fue la motivación principal para introducir instrucciones a nivel de assembly que permita CAS

Compare And Swap

¿Es posible realizar la operación `x = edit(x)` de una manera `thread safe`? ¿sin locks?

```
start:
    r1 := x
    r2 := foo(r1)
    r2 := CAS(x, r1, r2)
    if !r2 goto start
```

La respuesta es `CAS`: La variable `x` solamente se actualiza si el valor leído no ha cambiado y falla si detecta que el valor cambió durante la ejecución de la rutina.

[1] Esto fue la motivación principal para introducir instrucciones a nivel de assembly que permita CAS

Compare And Swap

¿Es posible realizar la operación `x = edit(x)` de una manera `thread safe`? ¿sin locks?

```
start:
    r1 := x
    r2 := foo(r1)
    r2 := CAS(x, r1, r2)
    if !r2 goto start
```

La respuesta es `CAS`: La variable `x` solamente se actualiza si el valor leído no ha cambiado y falla si detecta que el valor cambió durante la ejecución de la rutina. ¿Por qué esto siempre permite que al menos un proceso pueda actualizar `x` correctamente?

[1] Esto fue la motivación principal para introducir instrucciones a nivel de assembly que permita CAS

Compare And Swap

¿Es posible realizar la operación `x = edit(x)` de una manera `thread safe`? ¿sin locks?

```
start:
    r1 := x
    r2 := foo(r1)
    r2 := CAS(x, r1, r2)
    if !r2 goto start
```

La respuesta es `CAS`: La variable `x` solamente se actualiza si el valor leído no ha cambiado y falla si detecta que el valor cambió durante la ejecución de la rutina. ¿Por qué esto siempre permite que al menos un proceso pueda actualizar `x` correctamente?
Ejercicio: Implementa un stack de tamaño dinámico que sea *thread safe* pero no bloquee la ejecución de procesos. [1]

[1] Esto fue la motivación principal para introducir instrucciones a nivel de assembly que permita CAS

Compare And Swap

¿Es posible realizar la operación `x = edit(x)` de una manera `thread safe`? ¿sin locks?

```
start:
    r1 := x
    r2 := foo(r1)
    r2 := CAS(x, r1, r2)
    if !r2 goto start
```

La respuesta es `CAS`: La variable `x` solamente se actualiza si el valor leído no ha cambiado y falla si detecta que el valor cambió durante la ejecución de la rutina. ¿Por qué esto siempre permite que al menos un proceso pueda actualizar `x` correctamente?
Ejercicio: Implementa un stack de tamaño dinámico que sea *thread safe* pero no bloquee la ejecución de procesos. [1] Si se utilizan punteros para las operaciones con CAS entonces seguirá funcionando? Considera la situación donde tienes que manejar la memoria manualmente.

[1] Esto fue la motivación principal para introducir instrucciones a nivel de assembly que permita CAS

Ejercicio Propuesto

Una **Coroutine** es un tipo de subprogram donde la definición controla donde resumir la ejecución de este. Es decir, tras cada llamada la corutina puede elegir donde seguir ejecutando en la siguiente llamada. Los `Generator`s de Python son una implementación de corutinas.

Ejercicio Propuesto

Una **Coroutine** es un tipo de subprogram donde la definición controla donde resumir la ejecución de este. Es decir, tras cada llamada la corutina puede elegir donde seguir ejecutando en la siguiente llamada. Los `Generator`s de Python son una implementación de corutinas.
Explica por qué la sincronización competitiva no es un problema en un lenguaje (o cualquier enviroment) que tenga corutinas pero no concurrencia.
Sincronización competitiva es cuando dos procesos necesitan el uso de un recurso que no puede ser accedido simultaneamente.

Ejercicios propuestos

Problem Set del Capítulo 13 de Concepts of Programming Languages

- 1
- 4
- 6
- 9
- 10

Ejercicios del Capítulo 13 de Programming Languages Pragmatics

- 3
- 5
- 6
- 8
- 18
- 19
- 28

References


Robert Sebesta (2023)
Concepts of programming languages

The End

Questions? Comments?