Clase Semáforos y mapas y sumas de Erlang

Tomás Arjovsky

23 de Abril del 2018

Semáforos

Mecanismo de sincronización que se implementan con un contador c. **No** debería utilizarse para transmitir información.

- Si c > 0 el recurso está disponible.
- Si c <= 0 el recurso **no** está disponible. Puede ser negativo? No, es una invariante que no.
- c representa la cantidad de recursos disponibles.
- si c solo puede ser 0 o 1 es un semáforo binario, que básicamente es un lock.

NO puede usarse el contador hacia afuera. Es algo interno del semáforo para ver si está disponible o no un recurso.

Operaciones p(wait) resta 1, v(signal) suma 1. No las podemos usar en el TP, pero son las únicas operaciones. Solo para debuguear. Si se hace p y dejaría el contador en negativo debe haber un error.

Patrón barrera: Se separa en procesos. 2 procesos tienen puntos de sincronismo. Recién cuando llega el enésimo se activa el semáforo. ¿Cómo hacemos eso? Con un semáforo inicializado en n. Cada proceso hace p() y esperar a que sea 0.

Utilizaremos los semáforos de System V. Aquí no hay semáforos individuales, se manejan conjuntos, siempre. Si queremos uno solo, le decimos que queremos el conjunto de 1.

```
int semget(key_t key, int nsems, int semflg);
```

Crea u obtiene un semáforo/conjunto. Devuelve su id.

- key creada con ftok, como siempre.
- nsems: cantidad de semáforos. Tiene un máximo (ver).

- flags: tiene cosas como CREAT y EXCL como en memoria compartida.

El valor inicial del contador es basura.

```
int semctl(int semid, int semnum, int cmd, ...);
```

- semid: id del conjunto.
- semnum: número del semáforo dentro del conjunto.
- cmd: comando. Vamos a usar SETVAL, que dice qué valor se le da.
- resto: cúmulo de cosas en un union:

```
union semnum {
  int val; // 0 o mayor a 0!
  struct semid_ds* buf;
  ushort* array;
};
```

Cómo se usa? Ojo: para el tema de la creación podemos ver el error E_EXISTS para que todos intenten crearlo y si no existe simplemente lo piden. Problemón: alguien puede querer usarlo sin que haya sido inicializado el valor!!! Esto es un error de diseño de los semáforos de system V (esa separación de creación e inicialización). Una opción es crearlo antes de hacer el fork. Otra es hacer un lock exclusivo y listo. Una tercera está en la bibliografía, que es utilizando el atributo sem_otime que se modifica si ya se hizo la inicialización es un atributo interno.

GETVAL solo para contador.

SETALL es para todo semáforo.

IPC_RMID destruye todo el conjunto. Lo llama un solo proceso.

IPC_STAT: info del estado del conjunto.

Operaciones atómicas sobre Semáforos

```
int semop(int semid, struct sembuf* sops, unsigned nsops);
```

- semid: id.
- sops: lista de operaciones. El conjunto entero de esas operaciones será atómico, independientemente de su número. Puede fallar cualquiera. En cada struct se especifica sobre qué semáforo.
- nsops: cantidad de elementos del vector.

```
struct sembuf {
  ushort sem_num; // número de semáforo.
  short sem_op;
  short sem_flg;
}
```

- short semop: positivo aumenta, negativo decrementa, cero espera a que sea 0.
- short sem_flg:
  - IPC_NOWAIT: no se bloquea si es cero. Es caer fácilmente en busywait.
  - SEM_UNDO: se deshacen las operaciones cuando el programa termina de forma abrupta. Puede ser útil para los procesos que terminen de forma abrupta.

Opera sobre todo el conjunto de semáforos.

Ejemplos

Lectores y escritores

Recordemos que en memoria compartida los accesos no estaban sincronizados. Muchas veces eso puede resolverse con semáforos. En esta clase si el lector hace p y está en 0 se bloquea hasta que alguien (por ejemplo un escritor) haga v.

VER EJEMPLOS

hacemos un algoritmo en clase.

Parte de Erlang

Hoy vemos un toque del aspecto funcional de Erlang. Los que hayan trabajado en python conocerán sum y map. La aridad de la función sum es 1 (porque tiene un solo argumento). Map toma dos argumentos, una lista y una función. La función es como una función matemaica. Dado un x devuelve, por ejemplo, 2x. La aplica a todos los elementos. Map tiene aridad 2. En erlang los argumentos van al revés: primero va la función y luego la lista (por algo que se llama currying (no lo vemos aquí)).

lists_sum ya está programado en erlang.

```
lists:sum([1,2,3]) % devuelve 6.
```

En un map se puede pasar la definición de la funciónahí mismo.

```
lists:map(fun(X)-> 2*X end, [1,2,3]).
```

Se puede usar list_comprehension. No hay diferencias de performance grandes entre

```
[2*X || X <- [1,2,3]]
```

Cómo mejoramos el total del otro día sabiendo esto?

queríamos hacer la suma de totales parciales (cantidad de cada elemento por su precio).

```
total(L) ->
  lists:sum([Cant*precio(X) || {X, Cant} <- Lista]).
```

¿Cómo sería con maps? Usamos pattern matching para que en la misma definición de la función deconstruya la tupla.

```
totalConMap(L) ->
  lists:sum(
    lists:map(fun({X,Y})->Y*precio(X) end), L)
  ).
```

(recordar, las cosas que se exportan son las funciones y se aclara su aridad)-

Ojo, Erlang no maneja strings de forma nativa. Si ve que una lista cuyos elementos son todos códigos ASCIIs de caracterse imprimibles, automáticamente interpreta que esa lista es una cadena y la imprime como tal. Internamente sigue siendo una lista de enteros, pero su representación impresa cambia.

Elixir es un lenguaje implementado sobre BEAM (la máquina virtual de Erlang) que le quita las pequeñas odiosidades como terminar todo con ".", la falta de strings, etc.

Problema: puede que llamar a una función del map tarde un tiempo. Nos ayudaremos para hacerlo mejor el con el concepto similar a promesa. PMAP. Paralel Map. Se crean procesos de Erlang (de la máquina virtual BEAM, no del sistema, que pueden ocupar muy poco y ser millones).
