CC4302
Sistemas Operativos
Profesor: Luis Mateu

- Uso de spin-locks
- Implementación básica de spin-locks
- El problema de la memoria caché con los spin-locks
- Procolo MESI
- Implementación eficiente de spin-locks
- Núcleo clásico vs. núcleo moderno
- Núcleo monolítico vs. micro núcleo

Spin-locks

- Mutex que se implementa con *busy-waiting*
- Uso:

```
int sl= OPEN;

spinLock(&sl);
… // sl es CLOSED
spinUnlock(&sl);
```

- Es funcionalmente equivalente a un semáforo que puede almacenar máximo 1 ticket

Implementación incorrecta de spin-locks

```
void spinLock(int *plock) {
  while (*plock==CLOSED)
    ;  // ¡Incorrecta!
  *plock=CLOSED;
}
void spinUnlock(int *plock) {
  *plock= OPEN;
}
```

- Se necesita ayuda del hardware para poder implementar correctamente un spin-lock
- Todo los procesadores poseen una instrucción swap que intercambia ***atómicamente*** el valor almacenado en un registro por el valor almacenado en una dirección de memoria.

La instrucción swap

En assembler: swap [*R1*], *R2*

- R1 contiene la dirección de entero en memoria
- Ese entero puede contener OPEN o CLOSED
- R2 contiene el valor CLOSED
- Swap intercambia los valores almacenados en R2 y el entero en memoria
- Si el entero en memoria contenía OPEN, queda en CLOSED y R2 queda en OPEN
- Considere una función *swap* que hace lo mismo que esta otra función *swap* escrita en C:

```
int swap(int *psl, int val) {
   int ret= *psl;
   *psl= val;
   return ret;
}
```

- Pero está escrita en assembler para usar la instrucción de máquina swap y por lo tanto **es atómica**

Implementación ineficiente de spin-locks

```
void spinLock(int *psl) {
  while (swap(psl, CLOSED)==CLOSED)
    ;
}

void spinUnlock(int *psl) {
  *psl= OPEN;
}
```

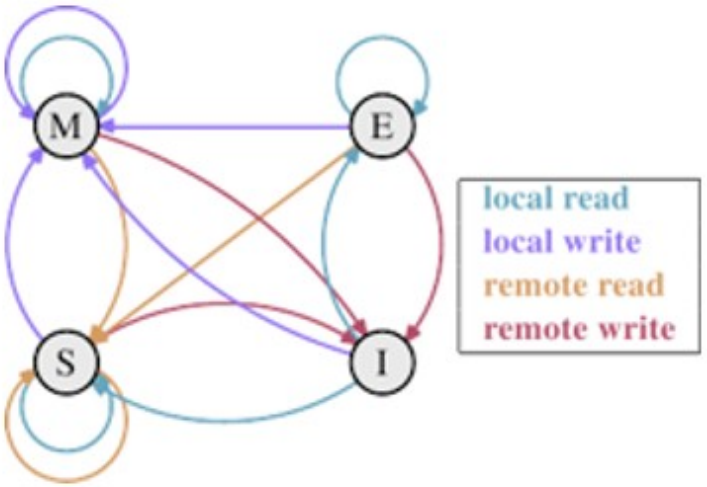
- Esta implementación es correcta
- Pero ineficiente porque puede producir exceso de tráfico en el bus de datos que es compartido por todos los procesadores para comunicarse entre sí y para llegar a la memoria
- No hay problema si se pide el spin-lock y está libre
- Tampoco hay problema si un solo procesador pide el spin-lock que está siendo ocupado por otro procesador
- El procesador en espera hará busy-waiting llamando a swap hasta que se invoque *spinUnlock*
- El acceso a la memoria de *swap* ocurrirá en la memoria cache del procesador en espera, sin interferir con la ejecución de otros procesadores

Coherencia de la memoria cache

- Por razones de eficiencia, cada procesador tiene su propio cache
- Supongamos que un procesador P1 ocupa el spin-lock y otro procesador P2 espera
- P1 y P2 mantienen una copia en sus caches del spin-lock con valor CLOSED
- Cuando P1 invoca spinUnlock, si se escribe OPEN solo en el cache de P1, pero no en P2, P2 podría quedar esperando indefinidamente
- El problema ocurre porque hay una inconsistencia en los valores almacenados para el spin-lock en los caches de P1 y P2
- Todos los computadores multicore deben implementar *un protocolo que garantice la coherencia de la memoria cache*: todos los procesadores deben ver el mismo valor

Protocolo MESI

- El protocolo de coherencia de memoria cache más conocido se denomina MESI, por **M**odified, **E**xclusive, **S**hared e **I**nvalid
- Cada línea disponible en la memoria cache puede estar en uno de estos 4 estados:
  - Modified: el procesador local ha modicado la línea, también implica que es la única copia dentro de los caches
  - Exclusive: la línea no ha sido modicada pero se sabe que es la única copia
  - Shared: la línea no está modicada y podrá existir en otros caches
  - Invalid: la línea está libre, no almacena ninguna línea de la memoria



Implementación eficiente de spin-locks

```
void spinLock(volatile int *psl) {
  do {
    while (*psl==CLOSED)
      ;
  } while (swap(psl, CLOSED)!=OPEN);
}
```

- Es eficiente porque los procesadores que esperan ocupar el spin-lock solo leen, no escriben
- Permite que los procesadores en espera mantengan una copia del spin-lock en sus propios caches en estado *shared*
- Solo la primera lectura requiere el uso del bus de datos compartido, el resto de las lecturas no
- El busy-waiting no causará ningún impacto en el desempeño de los procesadores que hacen trabajo útil
- Cuando se invoque spinUnlock, se usará el bus de datos una sola vez para invalidar las copias del spin-lock en los procesadores en espera

Importante

- Este es el único caso en el curso en donde se usa *busy-waiting* legalmente
- Retomar otro proceso no es una opción porque para extraerlo de la cola *ready* se necesita ingresar al núcleo
- Nunca haga *busy-waiting* Ud. mismo, porque es complicado implementarlo eficientemente
- Si necesita esperar:
  - cree un spin-lock *w* cerrado
  - agregue *&w* a algún contenedor conocido
  - espere invocando *spinLock(&w)*
  - Cuando pueda continuar, otro procesador extraerá el valor *pw=&w* del contenedor e invocará

    spinUnlock(pw)

Núcleo clásico vs. núcleo moderno

Clásico	***Moderno***
fines 70: System V 3.x, BSD, Linux 1, Win. 95	años 90: System V 4.x, Solaris 2, Linux 2, Win. NT
Simple	Complejo
Liviano en uso de CPU y memoria	Consumidor de CPU y memoria
Secuencial	Paralelo
Ideal para mono-core	Ideal para multi-core
Un solo core activo en el núcleo con interrupciones inhibidas	Múltiples threads en paralelo con interrupciones permitidas
Un spin-lock controla la entrada al núcleo	Múltiples spin-locks: 1 por cada estructura crítica
No hay dataraces ni deadlocks	Errores de programación pueden llevar a dataraces y deadlocks
Lento para reaccionar a las interrupciones	Rápido para reaccionar a interrupciones: *tiempo real*

Núcleo monolítico vs. micro núcleo

Núcleo monolítico	***micro núcleo***
Muchas funciones en el núcleo	Solo funciones esenciales: procesos y memoria
Muchos servicios brindados por procesos	Casi todo los servicios brindados por procesos
Sistema de archivos, la red, drivers en el núcleo	El sistema de archivos, la red, drives en procesos
Sistema gráfico en un proceso (Ej.: X-windows)	%
Complejo	Simple
Inseguro	Seguro
Difícil lograr estabilidad	Simple de lograr estabilidad
Eficiente en cambios de contexto	Alto sobrecosto en cambios de contexto