Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-322 — Sistemas Operativos I

Práctica 2

Multiprocesador y Exclusión Mútua

[Guido: Muchos de estos ejercicios los saqué del apunte de peterson]

Nota: Si desea limitar los procesadores en los que puede correr un proceso, puede usar el comando `taskset`.

Ej. 1. Suponga que queremos detectar si un array A contiene el entero 42 iterando por el mismo, y si es así, prender la bandera `encontrado`. ¿Hay diferencia entre los dos fragmentos siguientes? ¿En qué casos?

```
if (A[i] == 42)
    encontrado = true;
```

```
encontrado = encontrado || (A[i] == 42);
```

> **Solución.** Asumiendo que el compilador no cambia el significado esperado, la segunda versión siempre escribe a la variable, mientras que la primera sólo la "sube" a true cuando lo encuentra. Entonces, si estamos buscando con varios hilos, la segunda puede tener un race que pisa un true con un false, pero la primera no!

Ej. 2. ¿Puede fallar la siguiente aserción? ¿Bajo qué condiciones? Explique. Si puede fallar, arregle el programa.

```
int x = 0, y = 0, a = 0, b = 0;
void * foo(void *arg) { x = 1; a = y; return NULL; }
void * bar(void *arg) { y = 1; b = x; return NULL; }
int main() {
    pthread_t t0, t1;
    pthread_create(&t0, NULL, foo, NULL);
    pthread_create(&t1, NULL, bar, NULL);
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);
    assert (a || b);
    return 0;
}
```

> **Solución.** Sí puede fallar por las mismas razones que para Peterson: puede ser que 1) los reads se reordenen antes de los write y/o 2) que los writes queden en los store-buffer (caché) y no sean visibles por el otro proc. La solución es poner un mfence luego de cada asignación a x/y. Eso es en x86 por su modelo de memoria. En otras arquitecturas la respuesta puede cambiar. Con consistencia secuencial, esto anda bien.

Ej. 3. ¿Puede fallar la siguiente aserción (`wr` y `rd` corren en un thread cada uno)? Explique. Si puede fallar, arregle el programa.

```
volatile int x = 0;
volatile int y = 0;
void * wr(void *arg) { x = 123; y = 1; }
void * rd(void *arg) {
    while (!y)
        ;
    assert(x == 123);
}
```

> **Solución.** Sorprendentemente, en x86 anda bien por las garantías de memoria (los stores no se reordenan entre ellos)!. Sin embargo es posible que otra arquitectura requiera barreras de memoria.

Ej. 4. Implemente el algoritmo de Peterson para solucionar el problema del jardín ornamental. Tenga en cuenta lo discutido sobre barreras de memoria.

Ej. 5. Considere el problema del jardín ornamental en un sistema con un **único** procesador.

a) ¿Sigue habiendo un problema? Justifique.

b) Si implementa el algoritmo de Peterson, ¿son necesarias las barreras de memoria?

c) Si el incremento se hace con la instrucción `incl` de x86, ¿hay problema? Puede aprovechar la siguiente función:

```
static inline void incl(int *p) {
    asm("incl %0" : "+m"(*p) : : "memory");
}
```

d) ¿Qué pasa con la implementación con `incl` al tener más de un procesador?

e) Repita el experimento con esta versión de `incl`:

```
static inline void incl(int *p) {
    asm("lock; incl %0" : "+m"(*p) : : "memory");
}
```

> **Solución.**
>
> **a)** Sí, porque el thread se puede interrumpir en cualquier momento por varias razones (agotó quantum de scheduling, interrupción, etc).
>
> **b)** No hacen falta, porque es el mismo proc y no tiene problemas de consistencia con sí mismo. De hecho algunos sistemas Linux hacen patcheo en caliente para apagar las barreas de memoria (costosas) cuando se está corriendo con un solo proc.
>
> **c)** Con un sólo proc, `incl` basta para prevenir el problema, porque las interrupciones ocurren siempre antes o después de una instrucción, nunca en el medio. Dentro del mismo procesador, nunca vamos a ver una ejecución parcial de una instrucción.
>
> **d)** Con varios procesadores, ya no tenemos garantías de atomicidad, por más que sea una única instrucción. El problema es por store-buffering.
>
> **e)** Con `lock;incl` anda joya, porque el prefijo `lock` garantiza exclusión mútua via hardware.

Ej. 6. Para la versión ingenua (sin exclusión mútua) del jardín ornamental, ¿qué pasa cuando compilamos con optimizaciones? Pista: ver el assembler generado.

> **Solución.** Lo más factible es que `gcc` optimice el loop que suma de a 1 a una sola suma que suma `N_VISITANTES`. Sigue teniendo problemas de concurrencia, pero vamos a ver valores más "redondos" y que falla menos seguido.

Ej. 7. En la siguiente implementación del jardín ornamental (asumiendo dos molinetes), agregue estratégicamente algunos `sleep()` para obtener el mínimo valor posible de `visitantes`. Puede usar condicionales.

```
void * proc(void *arg) {
        int i;
        int id = arg - (void*)0;
        for (i = 0; i < N; i++) {
                int c;
                /* sleep? */
                c = visitantes;
                /* sleep? */
                visitantes = c + 1;
                /* sleep? */
        }
        return NULL;
}
```

> **Solución.** Esta es una solución robusta, siempre da 2:
>
> ```
> void * proc(void *arg) {
>     int i;
>     int id = (long)arg;
>     for (i = 0; i < N; i++) {
>         int c;
>
>         if (id == 1 && i == 1)   sleep(2);
>         if (id == 2 && i == 0)   sleep(1);
>         if (id == 2 && i == N-1) sleep(2);
>
>         c = visitantes;
>
>         if (id == 1 && i == 0)   sleep(2);
>         if (id == 2 && i == N-1) sleep(2);
>
>         visitantes = c + 1;
>     }
>     return NULL;
> }
> ```

Ej. 8. Compare la performance del jardín ornamental (para una misma cantidad de visitantes totales) para las siguiente implementaciones. Explique las diferencias (si las hay).

a) sin sincronización

b) usando el algoritmo de Peterson

c) usando `incl`

d) usando un `pthread_mutex_t`

e) usando un solo molinete sin multithreading.

> **Solución.** La última anda mucho más rápido que el resto porque no hay rebotes de la cache line que contiene a `cantidad` entre los procesadores. Es esperado. El resto de las diferencias son esperables.

Ej. 9 (extra). Usando CAS, implemente una variante del jardín ornamental sin usar locks. Compare la performance de esta variante lock-free con la variante que implementa un mutex via CAS, especialmente al aumentar mucho el número de hilos (ej. 100).

> **Solución.** Hay que usar básicamente esto:
>
> ```
> int old, new;
> do {
>         old = visitantes;
>         new = old + 1;
> } while (!casX86(&visitantes, old, new));
> ```
>
> para hacer un +1 atómico. La performance es similar para Nthreads bajo. Para muchos threads, es muchísimo mejor, porque no hay locks. El problema de la otra variante es que un thread puede tener el lock tomado e irse a dormir, forzando a todo el resto a spinnear. Con esta variante eso no pasa nunca.

Ej. 10. Analice y explique el comportamiento del siguiente programa.

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t a = PTHREAD_MUTEX_INITIALIZER;

void * foo(void *_arg)
{
        pthread_mutex_lock(&a);
        printf("Foo!\n");
        sleep(1);
        pthread_mutex_unlock(&a);
        return NULL;
}

int main()
{
        pthread_t th;
        pthread_create(&th, NULL, foo, NULL);

        pthread_mutex_t b = a;
        pthread_mutex_lock(&b);
        printf("Bar!\n");
        sleep(1);
        pthread_mutex_unlock(&b);

        pthread_join(th, NULL);
```

```
        return 0;
}
```

> **Solución.** El mutex **se copia** y los lock/unlock son independientes. Puede verse que ambos entran a su RC, o (con menos probabilidad) que main copia el mutex ya tomado, y queda en deadlock. Si la impresión de bar fuera en otro thread "hijo" de main, lo más probable sería un deadlock, por lo que tarda en arrancar.

Ej. 11 (difícil). Para el algoritmo de Peterson, ¿puede ubicarse el `mfence` entre la asignación al flag y la asignación a `turn`? Justifique.

> **Solución.** No anda! Supongamos esta ejecución, donde P1 ya está en su RC.
>
> | (P1) | (P2) |
> |---|---|
> | CRIT | `f2 = true` |
> | CRIT | BARRIER |
> | CRIT | |
> | CRIT | |
> | CRIT | |
> | CRIT | (P1 ve `f2=true`) |
> | CRIT | `turn = 1` |
> | `f1 = false` | wait.. |
> | `f1 = true` | wait.. |
> | BARRIER | llegó f1=false, rompe espera |
> | | CRIT |
> | | CRIT |
> | (P2 ve f1=true) | CRIT (llega f1=true, tarde, ya entramos) |
> | `turn = 2` | CRIT |
> | wait.. | CRIT |
> | wait.. | CRIT |
> | **llega `turn=1`!** | CRIT |
> | CRIT | CRIT |
>
> Cada barrier espera a que los store sean globamente visibles. Lo que pasa es que cuando P1 sale de su RC, baja su bandera y habilita a P2 a entrar. Mientras, puede ser que la actualización del turno que escribió P2 no haya llegado a P1, porque no está atrás de la barrera. Cuando llegue, P1 va a entrar a su RC.

Ej. 12. La siguiente función recorre una cadena `s` de longitud `len` y guarda en el array `r` cuales caracteres aparecieron en la cadena (asuma, por un momento, que podemos usar esta construcción de `parallel for`). ¿Hay condición de carrera?

```
void charsof(char *s, int len, bool r[256])
{
        int i;
        for (i = 0; i < 256; i++)
                r[i] = false;
        parallel for (i = 0; i < len; i++)
                r[s[i]] = true;
}
```

Solución. Es una carrera benigna.

Ej. 13 (Servidor de Turnos). Esta vez vamos a usar threads...

a) Adapte su implementación de la práctica anterior para atender concurrentemente a todas las conexiones abiertas levantando un thread nuevo por cada conexión. **Nota**: todos los clientes deberán poder hacer pedidos sin esperar a otros, y siempre debe poder conectarse un nuevo cliente. Esta vez, se debe garantizar que dos pedidos nunca reciben el mismo entero.

b) Implemente una solución con `select`/`epoll`.

c) Compare la performance de ambas soluciones.

Ej. 14 (No-tan-mini memcached). Adapte su implementación de la práctica anterior para atender pedidos en simultáneo, con una cantidad de threads definida estáticamente (ej. 4). El servidor debe seguir siendo *correcto* y tener la misma funcionalidad. Obviamente, los accesos y modificaciones a estructuras internas deben sincronizar para que no se corrompan. Además, debe considerarse cuáles threads manejan los eventos.

Opcional: Usar `epoll()` para manejar las conexiones. Las funciones `strchr()` y `memchr()` pueden ser útiles para el parseo. ¿Qué ventaja trae usarla? (Puede usar la librería `pthread`.)

Solución. Ver `fuentes/mini-memcached-tcp` para una implementación.

Ej. 15 (Algoritmo de la Panadería, Lamport). Implemente el algoritmo de la panadería de Lamport para el problema del jardín ornamental. Compare esta solución con las vistas anteriormente. En particular, compare el uso de memoria. También, considere crear una *librería* que implemente el algoritmo de Lamport: ¿hay algún problema?

Solución. Lamport es $O(n)$ memoria, con CAS es $O(1)$. El $O(n)$ es óptimo si sólo puede usarse load/store sobre memoria compartida (buscar paper). Lamport además tiene la desventaja de necesitar identificar cada thread, y su cantidad, de antemano.

Ej. 16 (Mutexes Recursivos). Implemente una librería de mutexes recursivos. Puede asumir que cada thread cuenta con un identificador único (e.g. el devuelto por `gettid()`), pero no debe asumir una cantidad fija ni máxima de los mismos.

Solución. TODO