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

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);
```

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;
}
```

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);
}
```

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");
}
```

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

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;
}
```

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.

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).

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;
}
```

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

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;
}
```

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. (Puede usar la librería `pthread`.)

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?

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.