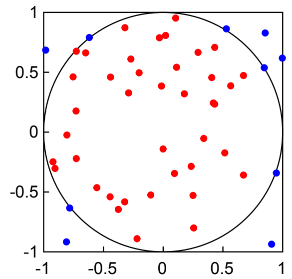
ID1206 HT24
Review Lab for Module #2
Deadline: 2024-11-22

---

1. Multi-threaded Monte Carlo Integration

Monte Carlo Integration is a numerical integration method. It can be used for determining the surface of a circle with radius r. This circle fits into a square of size $2 \cdot r$ as shown in the following figure:



The surface of the square is known to be $S_{\text{square}} = 4 \cdot r^2$.

Assume we randomly choose N_{try} tuples (x, y) where $-r \leq x, y \leq +r$. Let N_{hit} be the number tuples with $x^2 + y^2 \leq r^2$. (In the figure above, these tuples are shown as red bullets.) Using the Monte Carlo Integration method we can estimate the surface of the circle as follows:

$$S_{\text{circle}} \simeq \frac{N_{\text{hit}}}{N_{\text{try}}} S_{\text{square}} \,. \tag{1}$$

You can compare this result with the known result: $S_{\text{circle}} = \pi \, r^2$

Your task is to implement a multi-threaded program for computing S_{circle} for $r = 1$ with the following properties:

- Use POSIX threads
- Allow for the number of threads to be $N_{\text{thrd}} > 1$
- Use a Pthreads mutex for the concurrent update of N_{try} and N_{hit}
- Let the worker threads run forever while using the main thread to periodically print the current estimate for S_{circle}

Note that you need a pseudo-random number generator (RNG) that can be used in the context of multi-threading. `drand48_r` is a suitable choice as it allows to keep the RNG state in a thread-local variable. You can use `pthread_self()` to seed the RNG differently for all threads.

The use of a mutex is likely to introduce a high overhead. How could the overhead be reduced?

2. Implementation of a binary semaphore

Implement your own binary semaphore lock using atomic functions like `atomic_flag_test_and_set()` to make the following program to run:

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <pthread.h>

int counter = 0;
const int nthr = 1000;

typedef struct {
    /* MEMBERS TO BE DEFINED */
} mysemaphore_t;

mysemaphore_t sem;

int sem_init(mysemaphore_t *s) {
    /* FUNCTION TO BE IMPLEMENTED */
    return 0;
}

int sem_wait(mysemaphore_t *s) {
    /* FUNCTION TO BE IMPLEMENTED */
    return 0;
}

int sem_post(mysemaphore_t *s) {
    /* FUNCTION TO BE IMPLEMENTED */
    return 0;
}

void* func() {
    sleep(1);
    sem_wait(sem);
    counter++;
    sem_post(sem);
}

int main() {
    pthread_t thr[nthr];

    sem_init(&sem);

    for (int i = 0; i < nthr; i++)
        pthread_create(&thr[i], NULL, &func, NULL);

    for (int i = 0; i < nthr; i++)
        pthread_join(thr[i], NULL);

    printf("counter = %d\n", counter);

    return 0;
}
```

Test that the correct results is created using a large number of threads.

3. Semaphore with mitigated busy wait

To mitigate the impact of busy waiting, Tom Anderson[1] suggested to add a delay

[1]T. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors", 1990

after each attempt to acquire a lock. This delay may be incremented in each iteration as shown in the following pseudo-code:

```
type lock = (unlocked, locked)

procedure acquire_lock (L : ^lock)
    delay : integer := 1
    while test_and_set (L) = locked     // returns old value
        pause (delay)                   // consume this many units of time
        delay := delay * 2

procedure release_lock (L : ^lock)
    lock^ := unlocked
```

Update the code implemented in the previous task by adding a delay using the POSIX function `usleep()`. Check whether performance differences can be observed.

(doi: 10.1109/71.80120).