CS 342 - Operating Systems
Fall, 2024-2025

Project 3
03.12.2024

Emre Can Yologlu - 22102542
Burak Efe Ogut - 22103449

Number of Threads vs Time Performance of Deadlock Detection:
Setup:
We've fixed number resources to 10 and the main function runs a for loop for 10 iterations and sleeps for 2 seconds:

Number of Threads	Average Time Elapsed For A Deadlock Check(seconds)
2	0.000006169
4	0.000009275
8	0.000017473

Figure 1: Thread Count vs Elapsed Time Table

Explaining the experiment and evaluating the results:

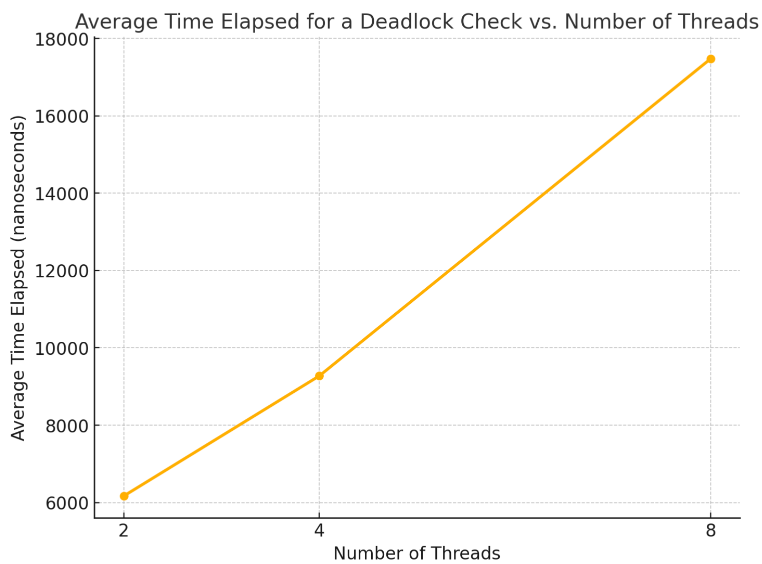

Figure 2: Thread Number vs Elapsed Time Graph

We wanted to see how the number of concurrent threads will affect the performance of the deadlock detection algorithm. We guessed that there would be a linear relation between them since increasing the thread count means we are increasing the row count and therefore causing the matrix search to take longer. Our graph and data supports our claim as the data increases very close to linear when the number of threads doubles(since matrix size doubles as well).

Number of Resources vs Time Performance of Deadlock Detection:
Setup: number of threads is fixed to 4 and the main function runs a for loop for 10 iterations and sleeps for 2 seconds:

Number of Resources	Average Time Elapsed For A Deadlock Check(seconds)
5	0.000002552
10	0.000003450
20	0.000004232

Figure 3: Resource Count vs Elapsed Time Table

Explaining the experiment and evaluating the results:

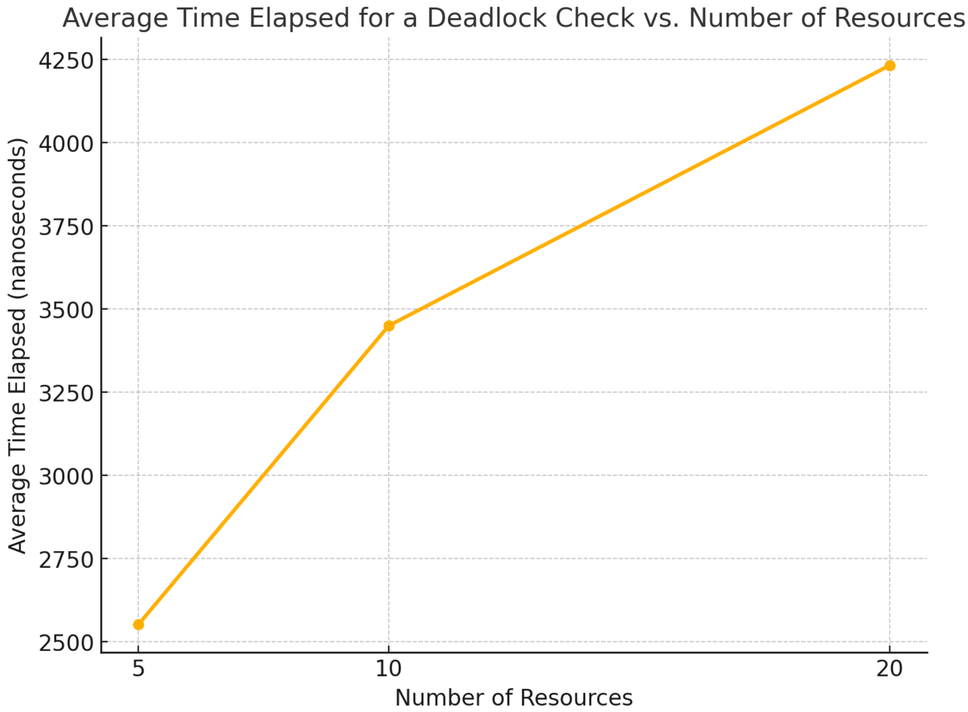

Figure 4: Resource Count vs Elapsed Time Graph

Similar to how the thread count would affect the detection search, we guessed that resource count would cause the time to increase linearly as it would cause the number of columns of the search matrix to increase. We have a more spiked increase from 5 to 10, more than linearly, and we suspect that to some hardware uncertainties of the experiment environment. Other than that our graph and data supports our claim.

Frequency of Deadlock Checking vs Time Performance of The Application:
Setup: number of threads is fixed to three and number of resources is fixed to 10. Number of iterations is fixed to 20.

Number of checks for a single iteration(frequency of deadlock checking)	Elapsed time for finishing the whole application(seconds)
1	20.094551807

2	40.106705323
4	80.294149239

Figure 4: Check Frequency vs Elapsed Time Table

Explaining the experiment and evaluating the results:

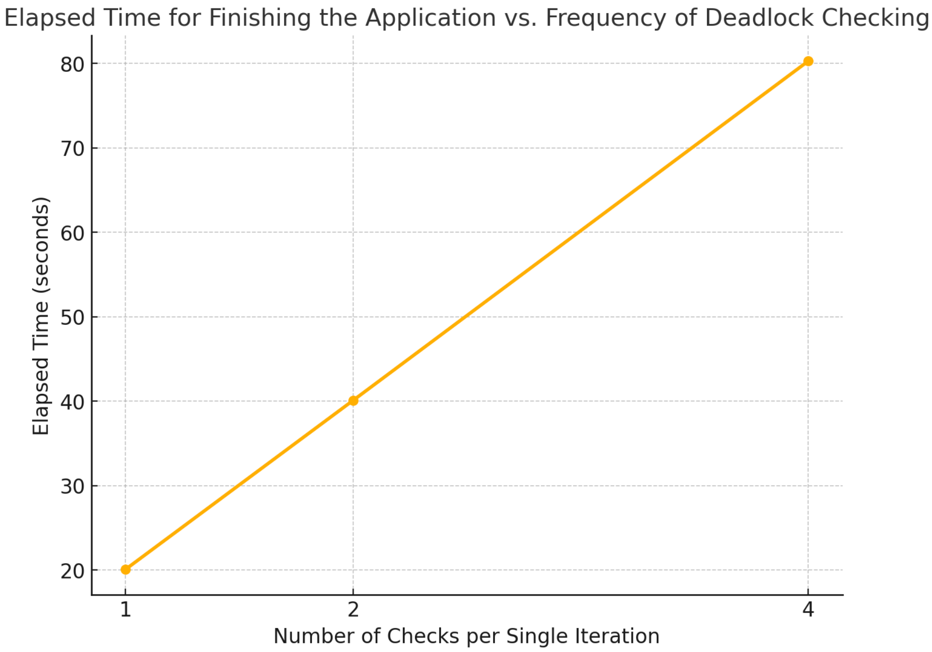

Figure 5: Check Frequency vs Elapsed Time Graph

In real life scenarios one issue regarding deadlock avoidance is how frequently the operating system should perform state checking. We wanted to simulate how the frequency of deadlock checking would affect the time elapsed for the application to finish. We hypothesized that as the frequency increases, the elapsed time would increase linearly as well. But we weren't sure about the degree of the correlation so we performed simulation experiments. As the data and the graph shows, there is a linear relationship between the deadlock checking frequency and application time.

Appendix:

Testing Environment Used:

```
#define _POSIX_C_SOURCE 199309L

#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdarg.h>
#include <time.h>
#include "reman.h"

#define NUMR 5 // Number of resources
#define NUMT 4  // Number of threads
#define ITERATIONS 3 // Number of detection checks


// Utility function to print resource requests and holdings
void pr(int tid, char astr[], int m, int r[]) {
   int i;
   printf("Thread %d, %s, [", tid, astr);
   for (i = 0; i < m; ++i) {
       if (i == (m - 1))
           printf("%d", r[i]);
       else
           printf("%d,", r[i]);
   }
   printf("]\n");
}

// Utility function to set arrays
void setarray(int r[], int m, ...) {
   va_list valist;
   int i;
   va_start(valist, m);
   for (i = 0; i < m; i++) {
       r[i] = va_arg(valist, int);
   }
   va_end(valist);
}
```

```c
// Generate a random array of resource requests (only 0 or 1 per
resource)
void generate_random_request(int r[], int m) {
    for (int i = 0; i < m; i++) {
        r[i] = rand() % 2; // Each resource request is 0 or 1 (1
instance per resource)
    }
}

// Thread function for Thread 1
void *threadfunc1(void *a) {
    int tid = *((int *)a);
    int claim[NUMR];
    int request[NUMR];
    int release[NUMR];

    reman_connect(tid);

    // Declare maximum resources this thread may claim (1 instance per
resource)
    setarray(claim, NUMR, 1, 1, 1, 1, 1);
    reman_claim(claim);

    for (int i = 0; i < ITERATIONS; i++) {
        // Randomly request resources
        generate_random_request(request, NUMR);
        pr(tid, "REQ", NUMR, request);
        reman_request(request);

        sleep(rand() % 3 + 1); // Simulate work (1-3 seconds)

        // Release the resources
        setarray(release, NUMR, request[0], request[1], request[2],
request[3], request[4],
                 request[5], request[6], request[7], request[8],
request[9]);
        pr(tid, "REL", NUMR, release);
        reman_release(release);

        sleep(rand() % 2 + 1); // Simulate additional work
    }

    reman_disconnect();
```

```
    pthread_exit(NULL);
}

// Thread function for Thread 2
void *threadfunc2(void *a) {
    int tid = *((int *)a);
    int claim[NUMR];
    int request[NUMR];
    int release[NUMR];

    reman_connect(tid);

    // Declare maximum resources this thread may claim (1 instance per
resource)
    setarray(claim, NUMR, 1, 1, 1, 1, 1);
    reman_claim(claim);

    for (int i = 0; i < ITERATIONS; i++) {
        // Randomly request resources
        generate_random_request(request, NUMR);
        pr(tid, "REQ", NUMR, request);
        reman_request(request);

        sleep(rand() % 3 + 1); // Simulate work (1-3 seconds)

        // Release the resources
        setarray(release, NUMR, request[0], request[1], request[2],
request[3], request[4],
                 request[5], request[6], request[7], request[8],
request[9]);
        pr(tid, "REL", NUMR, release);
        reman_release(release);

        sleep(rand() % 2 + 1); // Simulate additional work
    }

    reman_disconnect();
    pthread_exit(NULL);
}

// Thread function for Thread 3
void *threadfunc3(void *a) {
    int tid = *((int *)a);
```

```
    int claim[NUMR];
    int request[NUMR];
    int release[NUMR];

    reman_connect(tid);

    // Declare maximum resources this thread may claim (1 instance per
resource)
    setarray(claim, NUMR, 1, 1, 1, 1, 1);
    reman_claim(claim);

    for (int i = 0; i < ITERATIONS; i++) {
        // Randomly request resources
        generate_random_request(request, NUMR);
        pr(tid, "REQ", NUMR, request);
        reman_request(request);

        sleep(rand() % 3 + 1); // Simulate work (1-3 seconds)

        // Release the resources
        setarray(release, NUMR, request[0], request[1], request[2],
request[3], request[4],
                 request[5], request[6], request[7], request[8],
request[9]);
        pr(tid, "REL", NUMR, release);
        reman_release(release);

        sleep(rand() % 2 + 1); // Simulate additional work
    }

    reman_disconnect();
    pthread_exit(NULL);
}

// Thread function for Thread 4
void *threadfunc4(void *a) {
    int tid = *((int *)a);
    int claim[NUMR];
    int request[NUMR];
    int release[NUMR];

    reman_connect(tid);
```

```
    // Declare maximum resources this thread may claim (1 instance per
resource)
    setarray(claim, NUMR, 1, 1, 1, 1, 1);
    reman_claim(claim);

    for (int i = 0; i < ITERATIONS; i++) {
        // Randomly request resources
        generate_random_request(request, NUMR);
        pr(tid, "REQ", NUMR, request);
        reman_request(request);

        sleep(rand() % 3 + 1); // Simulate work (1-3 seconds)

        // Release the resources
        setarray(release, NUMR, request[0], request[1], request[2],
request[3], request[4],
                 request[5], request[6], request[7], request[8],
request[9]);
        pr(tid, "REL", NUMR, release);
        reman_release(release);

        sleep(rand() % 2 + 1); // Simulate additional work
    }

    reman_disconnect();
    pthread_exit(NULL);
}

void *threadfunc5(void *a) {
    int tid = *((int *)a);
    int claim[NUMR];
    int request[NUMR];
    int release[NUMR];

    reman_connect(tid);

    // Declare maximum resources this thread may claim (1 instance per
resource)
    setarray(claim, NUMR, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1);
    reman_claim(claim);

    for (int i = 0; i < ITERATIONS; i++) {
        // Randomly request resources
```

```
        generate_random_request(request, NUMR);
        pr(tid, "REQ", NUMR, request);
        reman_request(request);

        sleep(rand() % 3 + 1); // Simulate work (1-3 seconds)

        // Release the resources
        setarray(release, NUMR, request[0], request[1], request[2],
request[3], request[4],
                 request[5], request[6], request[7], request[8],
request[9]);
        pr(tid, "REL", NUMR, release);
        reman_release(release);

        sleep(rand() % 2 + 1); // Simulate additional work
    }

    reman_disconnect();
    pthread_exit(NULL);
}

void *threadfunc6(void *a) {
    int tid = *((int *)a);
    int claim[NUMR];
    int request[NUMR];
    int release[NUMR];

    reman_connect(tid);

    // Declare maximum resources this thread may claim (1 instance per
resource)
    setarray(claim, NUMR, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1);
    reman_claim(claim);

    for (int i = 0; i < ITERATIONS; i++) {
        // Randomly request resources
        generate_random_request(request, NUMR);
        pr(tid, "REQ", NUMR, request);
        reman_request(request);

        sleep(rand() % 3 + 1); // Simulate work (1-3 seconds)

        // Release the resources
```

```
        setarray(release, NUMR, request[0], request[1], request[2],
request[3], request[4],
                  request[5], request[6], request[7], request[8],
request[9]);
        pr(tid, "REL", NUMR, release);
        reman_release(release);

        sleep(rand() % 2 + 1); // Simulate additional work
    }

    reman_disconnect();
    pthread_exit(NULL);
}

void *threadfunc7(void *a) {
    int tid = *((int *)a);
    int claim[NUMR];
    int request[NUMR];
    int release[NUMR];

    reman_connect(tid);

    // Declare maximum resources this thread may claim (1 instance per
resource)
    setarray(claim, NUMR, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1);
    reman_claim(claim);

    for (int i = 0; i < ITERATIONS; i++) {
        // Randomly request resources
        generate_random_request(request, NUMR);
        pr(tid, "REQ", NUMR, request);
        reman_request(request);

        sleep(rand() % 3 + 1); // Simulate work (1-3 seconds)

        // Release the resources
        setarray(release, NUMR, request[0], request[1], request[2],
request[3], request[4],
                  request[5], request[6], request[7], request[8],
request[9]);
        pr(tid, "REL", NUMR, release);
        reman_release(release);
```

```
        sleep(rand() % 2 + 1); // Simulate additional work
    }

    reman_disconnect();
    pthread_exit(NULL);
}

void *threadfunc8(void *a) {
    int tid = *((int *)a);
    int claim[NUMR];
    int request[NUMR];
    int release[NUMR];

    reman_connect(tid);

    // Declare maximum resources this thread may claim (1 instance per
resource)
    setarray(claim, NUMR, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1);
    reman_claim(claim);

    for (int i = 0; i < ITERATIONS; i++) {
        // Randomly request resources
        generate_random_request(request, NUMR);
        pr(tid, "REQ", NUMR, request);
        reman_request(request);

        sleep(rand() % 3 + 1); // Simulate work (1-3 seconds)

        // Release the resources
        setarray(release, NUMR, request[0], request[1], request[2],
request[3], request[4],
                 request[5], request[6], request[7], request[8],
request[9]);
        pr(tid, "REL", NUMR, release);
        reman_release(release);

        sleep(rand() % 2 + 1); // Simulate additional work
    }

    reman_disconnect();
    pthread_exit(NULL);
}
```

```c
int main(int argc, char **argv) {
    pthread_t threads[NUMT];
    int tids[NUMT];
    int ret;
    double total_detection_time = 0.0; // To calculate average detection
time
    struct timespec start, end;

    if (argc != 2) {
        printf("Usage: %s <avoid_flag>\n", argv[0]);
        return 1;
    }

    int avoid = atoi(argv[1]);
    reman_init(NUMT, NUMR, avoid);

    srand(time(NULL)); // Seed random number generator

    // Create threads
    for (int i = 0; i < NUMT; i++) {
        tids[i] = i;
        if (i == 0) pthread_create(&threads[i], NULL, threadfunc1,
&tids[i]);
        if (i == 1) pthread_create(&threads[i], NULL, threadfunc2,
&tids[i]);
        if (i == 2) pthread_create(&threads[i], NULL, threadfunc3,
&tids[i]);
        if (i == 3) pthread_create(&threads[i], NULL, threadfunc4,
&tids[i]);
        if (i == 4) pthread_create(&threads[i], NULL, threadfunc5,
&tids[i]);
        if (i == 5) pthread_create(&threads[i], NULL, threadfunc6,
&tids[i]);
        if (i == 6) pthread_create(&threads[i], NULL, threadfunc7,
&tids[i]);
        if (i == 7) pthread_create(&threads[i], NULL, threadfunc8,
&tids[i]);

    }

    // Monitor for deadlocks
    for (int i = 0; i < ITERATIONS * 3; i++) {
        sleep(4);
```

```
        // Start timing
        clock_gettime(CLOCK_MONOTONIC, &start);

        ret = reman_detect();

        // End timing
        clock_gettime(CLOCK_MONOTONIC, &end);

        // Calculate elapsed time
        double elapsed_time = (end.tv_sec - start.tv_sec) + (end.tv_nsec
- start.tv_nsec) / 1e9;
        total_detection_time += elapsed_time;

        printf("Deadlock detection time: %.9f seconds\n", elapsed_time);

        if (ret > 0) {
            printf("Deadlock detected! Number of deadlocked threads:
%d\n", ret);
            reman_print("System State at Deadlock");
        }
    }

    double average_detection_time = total_detection_time / (ITERATIONS *
3);
    printf("Average deadlock detection time: %.9f seconds\n",
average_detection_time);

    // Wait for threads to complete
    for (int i = 0; i < NUMT; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```