CSE315 Project

Ans A)

Here's a C program that uses the fork() system call to generate the Collatz sequence for a given positive integer passed as a command line argument:

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s n\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    if (n <= 0) {
        printf("n must be a positive integer\n");
        return 1;
    }

    pid_t pid = fork();

    if (pid < 0) {
        printf("Fork failed\n");
        return 1;
    } else if (pid == 0) { // child process
        while (n != 1) {
            printf("%d, ", n);
            if (n % 2 == 0) {
                n = n / 2;
            } else {
                n = 3 * n + 1;
            }
        }
        printf("%d\n", n);
```

```
        exit(0);
    } else { // parent process
        wait(NULL);
    }

    return 0;
}
```

Ans B)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define NUM_CHILDREN 3
#define BUFFER_SIZE 100

int main() {
    pid_t pid[NUM_CHILDREN];
    int fd[NUM_CHILDREN][2]; //file descriptor
    char buffer[NUM_CHILDREN][BUFFER_SIZE];
    int n;

    // Create pipes
    for (int i = 0; i < NUM_CHILDREN; i++) {
        if (pipe(fd[i]) == -1) {
            printf("Error: pipe failed\n");
            exit(1);
        }
    }

    // Create child processes
    for (int i = 0; i < NUM_CHILDREN; i++) {
        pid[i] = fork();

        if (pid[i] < 0) {
            printf("Error: fork failed\n");
            exit(1);
        } else if (pid[i] == 0) {
```

```
        // Child process
        close(fd[i][0]); // Close unused read end of pipe

        // Read from serial line (keyboard)
        while ((n = read(STDIN_FILENO, buffer[i], BUFFER_SIZE)) > 0) {
            // Write to pipe
            write(fd[i][1], buffer[i], n);

            // Terminate if two newline characters received consecutively
            if (buffer[i][n-1] == '\n' && buffer[i][n-2] == '\n') {
                close(fd[i][1]); // Close write end of pipe
                exit(0);
            }
        }

        // Terminate if read error
        close(fd[i][1]); // Close write end of pipe
        exit(1);
    } else {
        // Parent process
        close(fd[i][1]); // Close unused write end of pipe
    }
}

// Read from pipes and output to console
while (1) {
    fd_set rfds;
    int maxfd = -1;
    FD_ZERO(&rfds);

    // Add file descriptors for all read ends of pipes to set
```

```
for (int i = 0; i < NUM_CHILDREN; i++) {
    if (fd[i][0] > maxfd) {
        maxfd = fd[i][0];
    }
    FD_SET(fd[i][0], &rfds);
}

// Wait for data on any pipe
if (select(maxfd + 1, &rfds, NULL, NULL, NULL) == -1) {
    printf("Error: select failed\n");
    exit(1);
}

// Read data from all pipes that have data
for (int i = 0; i < NUM_CHILDREN; i++) {
    if (FD_ISSET(fd[i][0], &rfds)) {
        n = read(fd[i][0], buffer[i], BUFFER_SIZE);
        if (n == -1) {
            printf("Error: read failed\n");
            exit(1);
        } else if (n == 0) {
            // Child process has terminated
            waitpid(pid[i], NULL, 0);
            close(fd[i][0]); // Close read end of pipe
        } else {
            // Output data to console
            write(STDOUT_FILENO, buffer[i], n);
        }
```

Ans C)

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

#define SIZE 10

int arr[SIZE] = {4, 2, 1, 6, 9, 8, 7, 5, 3, 0};
int temp_arr[SIZE];

typedef struct {
    int start;
    int end;
} thread_args;

void *sort(void *arg) {
    thread_args *t_args = (thread_args *) arg;
    int start = t_args->start;
    int end = t_args->end;

    // sort the sublist using insertion sort
    for (int i = start + 1; i <= end; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= start && arr[j] > key) {
            arr[j+1] = arr[j];
            j--;
        }

        arr[j+1] = key;
```

```
    }

    pthread_exit(NULL);
}

void merge(int start, int mid, int end) {
    int i = start, j = mid+1, k = start;

    while (i <= mid && j <= end) {
        if (arr[i] < arr[j]) {
            temp_arr[k] = arr[i];
            i++;
        } else {
            temp_arr[k] = arr[j];
            j++;
        }
        k++;
    }

    while (i <= mid) {
        temp_arr[k] = arr[i];
        i++;
        k++;
    }

    while (j <= end) {
        temp_arr[k] = arr[j];
        j++;
        k++;
    }
```

```
    for (int i = start; i <= end; i++) {
        arr[i] = temp_arr[i];
    }
}

int main() {                                                    3 threads
    thread_args t_args[2];
    pthread_t tid[3];

    int mid = SIZE / 2;

    // sort the two sublists using two separate threads
    t_args[0].start = 0;
    t_args[0].end = mid-1;
    pthread_create(&tid[0], NULL, sort, &t_args[0]);

    t_args[1].start = mid;
    t_args[1].end = SIZE-1;
    pthread_create(&tid[1], NULL, sort, &t_args[1]);

    // wait for both sorting threads to finish
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    // merge the two sorted sublists using a third thread
    pthread_create(&tid[2], NULL, merge, mid);

    // wait for the merging thread to finish
    pthread_join(tid[2], NULL);

    // output the sorted array
```

```
    printf("Sorted array: ");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Ans D)

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

#define MAX_READERS 10

int read_count = 0;
sem_t mutex, write;

void *writer(void *arg) {
    int writer_id = *(int *) arg;

    sem_wait(&write);
    printf("Writer %d is writing\n", writer_id);
    sem_post(&write);

    printf("Writer %d has finished writing\n", writer_id);
    pthread_exit(NULL);
}

void *reader(void *arg) {
    int reader_id = *(int *) arg;

    sem_wait(&mutex);
    read_count++;
    if (read_count == 1) {
        sem_wait(&write);
    }
    sem_post(&mutex);
```

```
    printf("Reader %d is reading\n", reader_id);

    sem_wait(&mutex);
    read_count--;
    if (read_count == 0) {
        sem_post(&write);
    }
    sem_post(&mutex);

    printf("Reader %d has finished reading\n", reader_id);
    pthread_exit(NULL);
}

int main() {
    int i, num_readers = 0, num_writers = 0;
    pthread_t readers[MAX_READERS], writers[MAX_READERS];

    printf("Enter the number of readers: ");
    scanf("%d", &num_readers);

    printf("Enter the number of writers: ");
    scanf("%d", &num_writers);

    sem_init(&mutex, 0, 1);
    sem_init(&write, 0, 1);

    int reader_ids[num_readers], writer_ids[num_writers];

    for (i = 0; i < num_readers; i++) {
```

```
        reader_ids[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
    }

    for (i = 0; i < num_writers; i++) {
        writer_ids[i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
    }

    for (i = 0; i < num_readers; i++) {
        pthread_join(readers[i], NULL);
    }

    for (i = 0; i < num_writers; i++) {
        pthread_join(writers[i], NULL);
    }

    sem_destroy(&mutex);
    sem_destroy(&write);

    return 0;
}
```

Ans E)

```
import java.net.*;
import java.io.*;

public class DateServer {

   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);
         while (true) {
            Socket client = sock.accept();
            // Start a new thread to handle the client request
            Thread t = new Thread(new ClientHandler(client));
            t.start();
         }
      } catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}

class ClientHandler implements Runnable {
   private Socket client;

   public ClientHandler(Socket socket) {
      this.client = socket;
   }

   @Override
   public void run() {
      try {
```

```
            PrintWriter pout = new PrintWriter(client.getOutputStream(), true);
            pout.println(new java.util.Date().toString());
            client.close();
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

ANS F)

```
/* buffer.h */

typedef int buffer_item;
#define BUFFER_SIZE 5

/* buffer.c */

#include <semaphore.h>
#include "buffer.h"

buffer_item buffer[BUFFER_SIZE];
int buffer_index;
sem_t empty;
sem_t full;
sem_t mutex;

void init_buffer() {
   buffer_index = 0;
   sem_init(&empty, 0, BUFFER_SIZE);
   sem_init(&full, 0, 0);
   sem_init(&mutex, 0, 1);
}

int insert_item(buffer_item item) {
   sem_wait(&empty); /* decrement empty count */
   sem_wait(&mutex); /* enter critical section */
   buffer[buffer_index] = item;
   buffer_index = (buffer_index + 1) % BUFFER_SIZE;
   sem_post(&mutex); /* leave critical section */
   sem_post(&full); /* increment full count */
```

```
    return 0;
}

int remove_item(buffer_item *item) {
    sem_wait(&full); /* decrement full count */
    sem_wait(&mutex); /* enter critical section */
    *item = buffer[(buffer_index - 1 + BUFFER_SIZE) % BUFFER_SIZE];
    buffer_index = (buffer_index - 1 + BUFFER_SIZE) % BUFFER_SIZE;
    sem_post(&mutex); /* leave critical section */
    sem_post(&empty); /* increment empty count */
    return 0;
}



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

void *producer(void *param) {
    buffer_item item;
    while (1) {
        /* generate a random item */
        item = rand();
        if (insert_item(item) == -1) {
            fprintf(stderr, "Producer error\n");
        }
        sleep(rand() % 5);
    }
}
```

```
void *consumer(void *param) {
    buffer_item item;
    while (1) {
        if (remove_item(&item) == -1) {
            fprintf(stderr, "Consumer error\n");
        } else {
            /* consume the item */
        }
        sleep(rand() % 5);
    }
}

int main(int argc, char *argv[]) {
    int sleep_time = atoi(argv[1]);
    int num_producers = atoi(argv[2]);
    int num_consumers = atoi(argv[3]);

    /* initialize buffer and semaphores */
    init_buffer();

    /* create producer threads */
    for (int i = 0; i < num_producers; i++) {
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, producer, NULL);
    }

    /* create consumer threads */
    for (int i = 0; i < num_consumers; i++) {
        pthread_t tid;
```

```
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, consumer, NULL);
}

/* sleep
```