Name: Newaz Ben Alam
ID: 2020789
Course: CSE315
Section: 03

Submitted to: Mohammad Noor Nabi

Ans A)

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

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

    int n = atoi(argv[1]);
    if (n <= 0) {
        printf("Error: Invalid argument, please provide a positive integer\n");
        return 1;
    }

    pid_t pid = fork();

    if (pid == -1) {
        printf("Error: Fork failed\n");
        return 1;
    } else if (pid == 0) { // Child process
        printf("%d", n);
        while (n > 1) {
            if (n % 2 == 0) {
                n = n / 2;
            } else {
                n = 3 * n + 1;
            }
            printf(", %d", n);
        }
        printf("\n");
    } else { // Parent process
        wait(NULL);
    }

    return 0;
}
```

Ans B)

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

#define BUF_SIZE 256

int main(void) {
   pid_t pid[3];
   int fd[6][2]; // 3 pipes, each with a read and a write end
   char buf[BUF_SIZE];
   int n;

   // Create three pipes
   for (int i = 0; i < 3; i++) {
      if (pipe(fd[i]) == -1) {
         perror("pipe");
         exit(1);
      }
   }

   // Create three child processes
   for (int i = 0; i < 3; i++) {
      pid[i] = fork();
      if (pid[i] == -1) {
         perror("fork");
         exit(1);
      } else if (pid[i] == 0) { // Child process
         close(fd[i][0]); // Close read end of pipe
         printf("Child %d ready to read from serial line\n", i);
         while (1) {
            n = read(STDIN_FILENO, buf, BUF_SIZE);
            if (n == -1) {
               perror("read");
               exit(1);
            }
            if (n == 0) {
               continue;
            }
            if (buf[0] == '\n' && buf[1] == '\n') {
               printf("Child %d terminating\n", i);
               close(fd[i][1]); // Close write end of pipe
```

```
            exit(0);
        }
        write(fd[i][1], buf, n);
    }
  } else { // Parent process
    close(fd[i][1]); // Close write end of pipe
  }
}

// Parent process reads from pipes
while (1) {
  fd_set set;
  FD_ZERO(&set);
  for (int i = 0; i < 3; i++) {
    FD_SET(fd[i][0], &set);
  }
  int max_fd = fd[2][0] + 1;
  select(max_fd, &set, NULL, NULL, NULL);
  for (int i = 0; i < 3; i++) {
    if (FD_ISSET(fd[i][0], &set)) {
      n = read(fd[i][0], buf, BUF_SIZE);
      if (n == -1) {
        perror("read");
        exit(1);
      }
      if (n == 0) {
        continue;
      }
      write(STDOUT_FILENO, buf, n);
    }
  }
  // Check if all child processes have terminated
  int status;
  for (int i = 0; i < 3; i++) {
    if (waitpid(pid[i], &status, WNOHANG) == pid[i] && WIFEXITED(status)) {
      printf("Child %d exited with status %d\n", i, WEXITSTATUS(status));
      close(fd[i][0]); // Close read end of pipe
    }
  }
  if (pid[0] == -1 && pid[1] == -1 && pid[2] == -1) {
    printf("All child processes terminated\n");
    break;
  }
}
```

```
    return 0;
}

Ans C)
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define ARRAY_SIZE 10

int array[ARRAY_SIZE] = { 10, 2, 8, 6, 3, 7, 1, 9, 5, 4 };
int sorted_array[ARRAY_SIZE];

void merge(int start, int middle, int end) {
    int left_index = start;
    int right_index = middle;
    int merged_index = start;

    while (left_index < middle && right_index < end) {
        if (array[left_index] < array[right_index]) {
            sorted_array[merged_index++] = array[left_index++];
        } else {
            sorted_array[merged_index++] = array[right_index++];
        }
    }

    while (left_index < middle) {
        sorted_array[merged_index++] = array[left_index++];
    }

    while (right_index < end) {
        sorted_array[merged_index++] = array[right_index++];
    }

    for (int i = start; i < end; i++) {
        array[i] = sorted_array[i];
    }
}

void* sort(void* arg) {
    int start = *(int*) arg;
    int end = start + ARRAY_SIZE / 2;
    for (int i = start; i < end - 1; i++) {
```

```
        for (int j = i + 1; j < end; j++) {
            if (array[i] > array[j]) {
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }
    pthread_exit(NULL);
}

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

    // create first sorting thread
    thread_args[0] = 0;
    pthread_create(&threads[0], NULL, sort, &thread_args[0]);

    // create second sorting thread
    thread_args[1] = ARRAY_SIZE / 2;
    pthread_create(&threads[1], NULL, sort, &thread_args[1]);

    // wait for sorting threads to finish
    for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }

    // merge the two sublists
    merge(0, ARRAY_SIZE / 2, ARRAY_SIZE);

    // print the sorted array
    printf("Sorted array: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

Ans D)

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

#define MAX_READERS 10

// Shared resource
int data = 0;

// Semaphores
sem_t mutex, count;

// Reader thread function
void* reader(void* arg) {
   int id = *(int*)arg;
   while (1) {
      sem_wait(&mutex);
      sem_post(&count);
      printf("Reader %d is reading data: %d\n", id, data);
      sem_wait(&count);
      sem_post(&mutex);
      // Reading is finished, do something else for a while
      usleep(rand() % 1000000);
   }
   pthread_exit(NULL);
}

// Writer thread function
void* writer(void* arg) {
   int id = *(int*)arg;
   while (1) {
      sem_wait(&mutex);
      printf("Writer %d is writing data\n", id);
      data++; // Write to the shared resource
      sem_post(&mutex);
      // Writing is finished, do something else for a while
      usleep(rand() % 1000000);
   }
   pthread_exit(NULL);
}

int main() {
```

```
    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&count, 0, MAX_READERS);

    // Create reader threads
    pthread_t reader_threads[MAX_READERS];
    int reader_ids[MAX_READERS];
    for (int i = 0; i < MAX_READERS; i++) {
        reader_ids[i] = i + 1;
        pthread_create(&reader_threads[i], NULL, reader, &reader_ids[i]);
    }

    // Create writer threads
    pthread_t writer_threads[2];
    int writer_ids[2] = {1, 2};
    for (int i = 0; i < 2; i++) {
        pthread_create(&writer_threads[i], NULL, writer, &writer_ids[i]);
    }

    // Wait for threads to finish
    for (int i = 0; i < MAX_READERS; i++) {
        pthread_join(reader_threads[i], NULL);
    }
    for (int i = 0; i < 2; i++) {
        pthread_join(writer_threads[i], NULL);
    }

    // Clean up semaphores
    sem_destroy(&mutex);
    sem_destroy(&count);

    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
```