Programming Projects on Design of Operating System

(Autumn'23)

Course Code: CSE315

Section: 02

Submitted to: Mohammad Noor Nabi

Submitted by: Kaushik Dey Joy

ID: 1821818

Date: 18.12.2023

A.

```
#include <iostream>
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void collatzSequence(int n) {
   std::cout << n << ", ";
   while (n != 1) {
      if (n % 2 == 0) {
         n = n / 2;
      } else {
         n = 3 * n + 1;
      }
      std::cout << n << ", ";
   }
   std::cout << std::endl;
}
int main(int argc, char *argv[]) {
   if (argc != 2) {
      std::cerr << "Usage: " << argv[0] << " <positive_integer>" << std::endl;
      exit(EXIT_FAILURE);
   }
```

```
    int startNumber = std::atoi(argv[1]);
     if (startNumber <= 0) {
        std::cerr << "Please provide a positive integer." << std::endl;
        exit(EXIT_FAILURE);
    }
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {         // Child process
        collatzSequence(startNumber);
    } else {            // Parent process
        wait(NULL);
    }
    return 0;
}
```

B.

```
#include <iostream>
#include <vector>
#include <thread>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
using namespace std;
```

```
const int NUM_CHILDREN = 3;

void child_process(int read_pipe, int write_pipe) {
    char buffer[256];
    int consecutive_newlines = 0;
    while (true) {
        int bytes_read = read(read_pipe, buffer, sizeof(buffer));
        if (bytes_read == -1) {
            perror("read error");
            exit(1);
        }
        for (int i = 0; i < bytes_read; ++i) {
            if (buffer[i] == '\n') {
                consecutive_newlines++;
            } else {
                consecutive_newlines = 0;
            }
            int bytes_written = write(write_pipe, &buffer[i], 1);
            if (bytes_written == -1) {
                perror("write error");
                exit(1);
            }
            if (consecutive_newlines == 2) {
                break;
            }
        }
```

```
    }
    close(read_pipe);
    close(write_pipe);
    exit(0);
}

int main() {
    int pipes[NUM_CHILDREN][2];
    for (int i = 0; i < NUM_CHILDREN; ++i) {
        if (pipe(pipes[i]) == -1) {
            perror("pipe error");
            exit(1);
        }
    }
    vector<thread> child_threads;
    for (int i = 0; i < NUM_CHILDREN; ++i) {
        child_threads.emplace_back(child_process, pipes[i][0], pipes[i][1]);
        close(pipes[i][0]);
    }
    char buffer;
    int child_terminated = 0;
    while (child_terminated < NUM_CHILDREN) {
        for (int i = 0; i < NUM_CHILDREN; ++i) {
            if (read(pipes[i][1], &buffer, 1) > 0) {
                cout << buffer;
            } else if (errno != EAGAIN) {
                close(pipes[i][1]);
```

```
            child_terminated++;
        }
    }
  }
  for (auto& thread : child_threads) {
    thread.join();
  }
  cout << "Child processes terminated." << endl;

  for (int i = 0; i < NUM_CHILDREN; ++i) {
    close(pipes[i][1]);
  }
  return 0;
}
```

C.

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>

int arr1[50] = {7, 12, 19, 3, 18, 4, 2, 6, 15, 8};
int arr2[50], arr3[50], arr4[50];
int subarr1, subarr2, total;

void *subarr1_func(void* arg) {
```

```
    sleep(1);
    std::cout << "\nFirst subarray: ";
    for (int i = 0; i < subarr1; i++) {
        std::cout << arr2[i] << " ";
    }

    for (int i = 0; i < subarr1; i++) {
        for (int j = 0; j < subarr1 - (i + 1); j++) {
            if (arr2[j] > arr2[j + 1]) {
                int temp = arr2[j];
                arr2[j] = arr2[j + 1];
                arr2[j + 1] = temp;
            }
        }
    }
    std::cout << "\nFirst Sorted array: ";
    for (int i = 0; i < subarr1; i++) {
        std::cout << arr2[i] << " ";
    }
}

void *subarr2_func(void* arg) {
    sleep(2);
    std::cout << "\nSecond subarray: ";
    for (int i = 0; i < subarr2; i++) {
        std::cout << arr3[i] << " ";
    }
```

```
    for (int i = 0; i < subarr2; i++) {
        for (int j = 0; j < subarr2 - (i + 1); j++) {
            if (arr3[j] > arr3[j + 1]) {
                int temp = arr3[j];
                arr3[j] = arr3[j + 1];
                arr3[j + 1] = temp;
            }
        }
    }
    std::cout << "\nSecond Sorted array: ";
    for (int i = 0; i < subarr2; i++) {
        std::cout << arr3[i] << " ";
    }
}

void *merge_func(void* arg) {
    sleep(3);
    total = subarr1 + subarr2;
    for (int i = 0; i < subarr1; i++) {
        arr4[i] = arr2[i];
    }

    int tempsubarr1 = subarr1;
    for (int i = 0; i < subarr2; i++) {
        arr4[tempsubarr1] = arr3[i];
        tempsubarr1++;
```

```
    }
    std::cout << "\nMerged Array: ";
    for (int i = 0; i < total; i++) {
        std::cout << arr4[i] << " ";
    }

    for (int i = 0; i < total; i++) {
        for (int j = 0; j < total - i - 1; j++) {
            if (arr4[j + 1] < arr4[j]) {
                int temp = arr4[j];
                arr4[j] = arr4[j + 1];
                arr4[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n = 10;
    pthread_t t1, t2, t3;

    std::cout << "Given Array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr1[i] << " ";
    }

    int j = 0;
```

```
for (int i = 0; i < n / 2; i++) {
    arr2[j] = arr1[i];
    j++;
}
subarr1 = j;

int k = 0;
for (int i = n / 2; i < n; i++) {
    arr3[k] = arr1[i];
    k++;
}
subarr2 = k;

pthread_create(&t1, NULL, subarr1_func, NULL);
pthread_create(&t2, NULL, subarr2_func, NULL);
pthread_create(&t3, NULL, merge_func, NULL);

pthread_join(t1, NULL);
pthread_join(t2, NULL);
pthread_join(t3, NULL);

std::cout << "\nSorted Merged Array: ";
for (int i = 0; i < total; i++) {
    std::cout << arr4[i] << " ";
}
std::cout << "\n";
```

```
    return 0;
}
```

D.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <semaphore>

std::mutex mtx;
std::semaphore wrtSem(1);    // Writer semaphore with initial value 1
std::semaphore readSem(1);     // Reader semaphore with initial value 1

int readerCount = 0;     // Counter

void reader() {
  while (true) {
    wrtSem.wait();

    // Lock mutex to update count
    mtx.lock();
    readerCount++;
    if (readerCount == 1) {
      readSem.wait();
    }
```

```
    mtx.unlock();

    std::this_thread::sleep_for(std::chrono::milliseconds(100));


    mtx.lock();
    readerCount--;
    if (readerCount == 0) {
      readSem.signal();    }
    mtx.unlock();

    wrtSem.signal();

    std::this_thread::sleep_for(std::chrono::milliseconds(500));
  }
}

void writer() {
  while (true) {
    wrtSem.wait();

    mtx.lock();

    std::cout << "Writer is writing...\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
```

```
    // Unlock mutex
    mtx.unlock();

    wrtSem.signal(); // Release writer semaphore

    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  }
}

int main() {
  std::vector<thread> readers, writers;
  for (int i = 0; i < 5; ++i) {
    readers.emplace_back(reader);
  }
  for (int i = 0; i < 3; ++i) {
    writers.emplace_back(writer);
  }

  // Join all threads
  for (auto& thread : readers) {
    thread.join();
  }
  for (auto& thread : writers) {
    thread.join();
  }
  return 0;
 }
```

E.

Server.cpp:

```
#include <iostream>
#include <cstring>
#include <thread>
#include <netinet/in.h>
#include <unistd.h>

constexpr int PORT = 1234;
constexpr int BUFFER_SIZE = 1024;

void handleClient(int clientSocket) {
    char buffer[BUFFER_SIZE];
    ssize_t bytesRead;

    while ((bytesRead = recv(clientSocket, buffer, sizeof(buffer), 0)) > 0) {
        buffer[bytesRead] = '\0';
        std::cout << "Sent from the client: " << buffer << std::endl;

        if (strcmp(buffer, "exit") == 0) {
            std::cout << "Client Disconnected" << std::endl;
            send(clientSocket, "you are disconnected", strlen("you are disconnected"), 0);
            break;
        } else {
            send(clientSocket, buffer, bytesRead, 0);
```

```
        }
    }

    close(clientSocket);
}

int main() {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == -1) {
        perror("Error creating socket");
        return -1;
    }

    sockaddr_in serverAddress{};
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(PORT);
    serverAddress.sin_addr.s_addr = INADDR_ANY;

    if (bind(serverSocket, reinterpret_cast<struct sockaddr*>(&serverAddress),
sizeof(serverAddress)) == -1) {
        perror("Error binding socket");
        close(serverSocket);
        return -1;
    }

    if (listen(serverSocket, 10) == -1) {
        perror("Error listening on socket");
```

```
        close(serverSocket);
        return -1;
    }

    std::cout << "Server listening on port " << PORT << std::endl;

    while (true) {
        sockaddr_in clientAddress{};
        socklen_t clientAddressSize = sizeof(clientAddress);

        int clientSocket = accept(serverSocket, reinterpret_cast<struct
sockaddr*>(&clientAddress), &clientAddressSize);
        if (clientSocket == -1) {
            perror("Error accepting connection");
            continue;
        }

        std::cout << "New client connected " << inet_ntoa(clientAddress.sin_addr) << std::endl;

        std::thread(handleClient, clientSocket).detach();
    }

    close(serverSocket);
    return 0;
}
```

Client.cpp:

```
#include <iostream>
#include <cstring>
#include <netinet/in.h>
#include <unistd.h>

constexpr int PORT = 1234;
constexpr int BUFFER_SIZE = 1024;

int main() {
   int clientSocket = socket(AF_INET, SOCK_STREAM, 0);
   if (clientSocket == -1) {
      perror("Error creating socket");
      return -1;
   }

   sockaddr_in serverAddress{};
   serverAddress.sin_family = AF_INET;
   serverAddress.sin_port = htons(PORT);
   serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1");

   if (connect(clientSocket, reinterpret_cast<struct sockaddr*>(&serverAddress),
sizeof(serverAddress)) == -1) {
      perror("Error connecting to server");
      close(clientSocket);
      return -1;
```

```
    }

    std::cout << "Connected to server" << std::endl;

    char buffer[BUFFER_SIZE];
    std::string line;

    while (true) {
        std::cout << "Enter message (type 'exit' to quit): ";
        std::getline(std::cin, line);

        if (line == "exit") {
            send(clientSocket, line.c_str(), line.size(), 0);
            break;
        }

        send(clientSocket, line.c_str(), line.size(), 0);

        ssize_t bytesRead = recv(clientSocket, buffer, sizeof(buffer), 0);
        if (bytesRead > 0) {
            buffer[bytesRead] = '\0';  // Ensure null-termination
            std::cout << "Server replied: " << buffer << std::endl;
        }
    }
    close(clientSocket);
    return 0;
}
```

F.

```
#include<iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <vector>
#include <chrono>
#include <cstdlib>

#define BUFFER_SIZE 5
typedef int buffer_item;

std::mutex mtx;
std::condition_variable full, empty;
std::vector<buffer_item> buffer;
int counter = 0;

void initializeData() {
   counter = 0;
}

int insert_item(buffer_item item) {
   if (counter < BUFFER_SIZE) {
      buffer.push_back(item);
      ++counter;
      return 0;
```

```
    } else {
        return -1;
    }
}

int remove_item(buffer_item *item) {
    if (counter > 0) {
        *item = buffer.back();
        buffer.pop_back();
        --counter;
        return 0;
    } else {
        return -1;
    }
}

void producer() {
    buffer_item item;
    while (true) {
        int rNum = rand() / 100000000;
        std::this_thread::sleep_for(std::chrono::milliseconds(rNum));

        item = rand() % 100;
        std::unique_lock<std::mutex> lock(mtx);
        empty.wait(lock, [] { return counter < BUFFER_SIZE; });

        if (insert_item(item)) {
```

```
            std::cerr << "Producer report error condition\n";
        } else {
            std::cout << "Producer produced: " << item << std::endl;
        }

        lock.unlock();
        full.notify_one();
    }
}

void consumer() {
    buffer_item item;
    while (true) {
        int rNum = rand() / 1000000000;
        std::this_thread::sleep_for(std::chrono::milliseconds(rNum));

        std::unique_lock<std::mutex> lock(mtx);
        full.wait(lock, [] { return counter > 0; });

        if (remove_item(&item)) {
            std::cerr << "Consumer report error condition\n";
        } else {
            std::cout << "Consumer consumed: " << item << std::endl;
        }

        lock.unlock();
        empty.notify_one();
```

```
    }
}

int main(int argc, char *argv[]) {
    // Argument validation...

    int sleeptime = std::atoi(argv[1]);
    int numProd = std::atoi(argv[2]);
    int numCons = std::atoi(argv[3]);

    initializeData();

    std::vector<std::thread> producer_threads, consumer_threads;

    for (int i = 0; i < numProd; ++i) {
        producer_threads.emplace_back(producer);
    }

    for (int i = 0; i < numCons; ++i) {
        consumer_threads.emplace_back(consumer);
    }

    std::this_thread::sleep_for(std::chrono::seconds(sleeptime));
    std::cout << "Exiting the program" << std::endl;

    return 0;
}
```