Producer/Consumer Monitor

#define N_BUF 5

class ProducerConsumer: public Gladiator {
public: // Constructor
ProducerConsumer();

public: // Member functions
void store_data( int value );
int retrieve_data();

private: // Data members
int buffer[ N_BUF ]; Array in which values are stored
int next_full; Index of next stored value
int next_empty; Index of next available space
int n_full; Number of values currently stored
Condition full_buffers; Block processes waiting for data
Condition empty_buffers; Block processes waiting for space
}; // ProducerConsumer

ProducerConsumer::ProducerConsumer():
next_full( 0 ), next_empty( 0 ), n_full( 0 ){
}

void ProducerConsumer::store_data( int value ){
    enter();
    if( n_full == N_BUF ){
        wait( empty_buffers );
    }
    buffer[ next_empty ] = value;
    n_full += 1;
    next_empty = ( next_empty + 1 ) % N_BUF;
    signal( full_buffers );
    leave();
}

int ProducerConsumer::retrieve_data(){
    int value;
    enter();
    if( n_full == 0 ){
        wait( full_buffers );
    }
    value = buffer[ next_full ];
    n_full -= 1;
    next_full = ( next_full + 1 ) % N_BUF;
    signal( empty_buffers );
    leave();
    return value;
}
Mailbox Monitor

#define MAX_MESSAGES 5

class Mailbox: public Gladiator {
    public: // Constructor
        Mailbox();
    
    public: // Member functions
        void send( int message ); // Send a message to another process
        int receive(); // Receive a message from another process
    
    private: // Data members
        int messages[ MAX_MESSAGES ]; // Space to store the messages
        int n_messages; // Number of messages stored
        Condition a_message; // Processes waiting for messages to arrive
        Condition some_space; // Processes waiting for empty space
}; // Mailbox

Mailbox::Mailbox(): Gladiator(), n_messages( 0 ){
}

void Mailbox::send( int message ){
    enter();

    if( n_messages == MAX_MESSAGES ){
        wait( some_space );
    }

    messages[ n_messages ] = message; // Update the counter, store the new message,
    n_messages += 1; // and signal in case anyone was waiting for messages.
    signal( a_message );

    leave();
}

int Mailbox::receive(){
    int message;

    enter();

    if( n_messages == 0 ){
        wait( a_message ); // If there aren't any messages, wait for one to arrive.
    }

    n_messages -= 1; // Retrieve the latest message, update the counter, and signal in case anyone was waiting for empty space.
    message = messages[ n_messages ];
    signal( some_space );

    leave();
    return message;
}
Dining Philosopher’s Monitor

class DiningPhilosophers: public Gladiator {
public: // Constructor
    DiningPhilosophers();

public: // Member functions
    void pick_up_forks( int who ); // Philosopher calls these to pick up and put down forks.
    void put_down_forks( int who );

private: // Data members
    bool fork[ 5 ]; // Status of each fork
    Condition forks_released; // Blocks processes waiting for forks
}; // DiningPhilosophers

DiningPhilosophers::DiningPhilosophers(): Gladiator(){
    int i;
    for( i = 0; i < 5; i += 1 ){
        fork[ i ] = true; // Initially, all forks are available.
    }
}

void DiningPhilosophers::pick_up_forks( int who ){
    enter();
    int other( ( who + 1 ) % 5 ); // who is the philosopher (and the left fork), and other is the right fork.
    while( !( fork[ who ] && fork[ other ] ) ){ // So long as both forks are not available at the same time, wait for someone to put forks down.
        wait( forks_released );
    }
    fork[ who ] = false; // We will pick up our forks, mark them as unavailable.
    fork[ other ] = false;
    leave();
}

void DiningPhilosophers::put_down_forks( int who ){
    enter();
    int other( ( who + 1 ) % 5 ); // who is the philosopher (and the left fork), and other is the right fork.
    fork[ who ] = true; // Mark the forks as available once again.
    fork[ other ] = true;
    while( queue( forks_released ) ){ // Release all processes that were waiting for forks so they can all check again; one of them may now be able to proceed.
        signal( forks_released );
    }
    leave();
}
Readers/Writers Monitor

class ReadersWriters: public Gladiator {

public: // Constructor
    ReadersWriters();

public: // Member functions
    void begin_reading(); // Processes call these when they want to start reading or writing, and when they're done reading or writing.
    void done_reading();
    void begin_writing();
    void done_writing();

private: // Data members
    int r_active; // Number of active readers
    int r_waiting; // Number of readers waiting to enter
    int w_active; // Number of active writers
    int w_waiting; // Number of writers waiting to enter
    Condition ok_to_read; // Readers waiting to enter
    Condition ok_to_write; // Writers waiting to enter
}; // ReadersWriters

ReadersWriters::ReadersWriters(): Gladiator(), r_active(0), r_waiting(0), w_active(0), w_waiting(0) {
}

void ReadersWriters::begin_reading() {
    enter();

    if (w_active > 0 || w_waiting > 0) {
        r_waiting += 1;
        wait(ok_to_read);
    }
    r_active += 1;

    leave();
}

void ReadersWriters::done_reading() {
    enter();

    r_active -= 1;
    if (r_active == 0) {
        signal(ok_to_write);
    }

    leave();
}
void ReadersWriters::begin_writing(){
    enter();
    if( r_active > 0 || w_active > 0 ){
        w_waiting += 1;
        wait( ok_to_write );
        w_waiting -= 1;
    }
    w_active += 1;
    leave();
}

void ReadersWriters::done_writing(){
    enter();
    w_active -= 1;
    if( r_waiting > 0 ){
        while( r_waiting > 0 ){
            signal( ok_to_read );
            r_waiting -= 1;
        }
    } else {
        signal( ok_to_write );
    }
    leave();
}
Write Your Own Monitors

You can get the source code for these monitors by typing the following command. I recommend you create a new directory and change to it before entering the command.

```
get kar 440-sync-examples
```

You will get the files

- `DiningPhilosophers.C`
- `Mailbox.C`
- `ProducerConsumer.C`
- `ReadersWriters.C`

which contain the monitors and client processes that exercise them.

You will also get some read-only files:

- `Monitor.h` and `Monitor.C` contain the same code that was presented in class.
- `Semaphore.h` is the same interface that was presented in class, and `Semaphore.C` implements this interface using the system’s native synchronization features.
- `Thread.h` and `Thread.C` provide a simplified interface to the system’s thread library.
- `README` contains instructions on how to compile and run the programs, and how to write monitors of your own.

Finally, read the comments in `header.mak` for information on how to change the unblocking ordering of the semaphores, a useful strategy for testing synchronization programs.