```
#include <pthread.h>
#include <map>
#include <set>
#include <string.h>
#include <signal.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

//!
//! \class  two_branches_speculator
//!
//! \brief  implements a class that takes 2 conditional branches and
//!         executes them speculatively, until the supposition is
//!         proven in the pre-branch section. In order to do this the
//!         class relies on write_data and read_data functions, that
//!         check for data dependence violations; and 3 other functions
//!         (validate_supposition, speculate and get_results),
//!         that are in charge of control dependence.
//!
//! Limitations:
//!
//! * If a speculative thread is to be canceled, it cannot use functions
//! that involve system mutexes, such as printf, etc. In this case, it
//! is possible that the thread can be canceled while holding such a mutex,
//! and the application could go into deadlock. To prevent this the user
//! has to surround this "dangerous" code with:
//!     "pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);" and
//!     "pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);".
//!
//! * Sequential consistency is mostly guaranteed, save for exception
//! behaviour.
//!
class two_branches_speculator : protected _conditional_speculator
{

private:

        //enum value to determine which branch has been proven valid
        //a PB value means that so far no branch has been proven valid.
        enum _branch {PB, B1, B2} _valid_branch;

        //boolean value to determine if this is the object's first run
        //this will be used to avoid segmentation faults in the reseting
        //of the object
        bool _first_run;

        //parallel sections of the class
        _previousSection _pb;
        _speculativeBranch _b1, _b2;

        //array of canceled threads-ids
        set <pthread_t> _canceled_threads;

        //red-black tree used to keep track of data/thread readings
        //the void* is a reference to a data element in _shared_data,
        //the _branch variable indicates which branch has read, a PB
        //value means both branches have read.
        map <void*, _branch> _readers_log;

        //red-black tree containing mutexes related to each element in the
        //previous array.
        map <void*, pthread_mutex_t> _readers_log_mutexes;

        //mutex related to _readers_log as a whole, also used for _canceled_threads
        pthread_mutex_t _readers_log_mutex;

public:

        //!
        //! default constructor
        //!
        two_branches_speculator(){

                _first_run=true;

                _valid_branch=PB;

                pthread_mutex_init (&_readers_log_mutex, NULL);
        };

private:
        //!
        //! private method to reset or cancel speculative branches
```

```
//!
//! TO BE NOTED:
//! * if given PB as argument, cancels both B1 and B2
//! * this method takes all class mutexes. On invocation
//!   no mutexes should be on hold save those exclusively related to a branch that
//!   is not about to be canceled or restarted.
//!
void _reset_branch (_branch BR, bool reset_all_logs, bool cancel){

        //if given PB as argument, cancels both branches and
        //clears their copied data.


        if (BR==PB){

                pthread_mutex_lock(&_b1._mutex);
                pthread_mutex_lock(&_b2._mutex);
                pthread_mutex_lock(&_b1._copied_data_mutex);
                pthread_mutex_lock(&_b2._copied_data_mutex);

                map <void*, pthread_mutex_t>::iterator it;
                for (it=_readers_log_mutexes.begin(); it!=_readers_log_mutexes.end(); it++){
                        pthread_mutex_lock(&it->second);
                }
                pthread_mutex_lock(&_readers_log_mutex);


                pthread_mutex_lock(&_is_running_mutex);
                pthread_mutex_lock(&_valid_branch_mutex);

                if (pthread_kill (_b1._thread, 0)==0)
                        pthread_cancel (_b1._thread);
                _canceled_threads.insert(_b1._thread);


                if (pthread_kill (_b2._thread, 0)==0)
                        pthread_cancel (_b2._thread);
                _canceled_threads.insert(_b2._thread);

                map <void*, _data_copy*>::iterator k;
                if (!_b1._copied_data.empty()){
                        for (k=_b1._copied_data.begin(); k!=_b1._copied_data.end(); k++){
                                free ((void*)(k->second));
                                k->second=NULL;
                        }
                        _b1._copied_data.clear();
                }

                if (!_b2._copied_data.empty()){
                        for (k=_b2._copied_data.begin(); k!=_b2._copied_data.end(); k++){
                                free ((void*)(k->second));
                                k->second=NULL;
                        }
                        _b2._copied_data.clear();
                }

                _b1._read_data.clear();
                _b2._read_data.clear();

                _readers_log.clear();

                pthread_mutex_unlock(&_valid_branch_mutex);
                pthread_mutex_unlock(&_is_running_mutex);

                pthread_mutex_unlock(&_readers_log_mutex);

                for (it=_readers_log_mutexes.begin(); it!=_readers_log_mutexes.end(); it++){
                        pthread_mutex_unlock(&it->second);
                }

                pthread_mutex_unlock(&_b2._copied_data_mutex);
                pthread_mutex_unlock(&_b1._copied_data_mutex);



                //if (!cancel) means that the threads should be restarted

                if (!cancel){

                        pthread_attr_t attr;
                        pthread_attr_init (&attr);
                        pthread_attr_setschedpolicy(&attr, _sched_option);

                        pthread_create (&_b1._thread, &attr, _b1._instructions, _b1._args);
                        pthread_create (&_b2._thread, &attr, _b2._instructions, _b2._args);
                }
        pthread_mutex_unlock(&_b2._mutex);
        pthread_mutex_unlock(&_b1._mutex);
}
else if (BR==B1){

        pthread_mutex_lock(&_b1._mutex);
        pthread_mutex_lock(&_b1._copied_data_mutex);

        set <void*>::iterator it;

        pthread_mutex_lock(&_readers_log_mutex);

        pthread_mutex_lock(&_is_running_mutex);
        pthread_mutex_lock(&_valid_branch_mutex);

        if (pthread_kill (_b1._thread, 0)==0)
                pthread_cancel (_b1._thread);
        _canceled_threads.insert(_b1._thread);

        if (!_b1._copied_data.empty()){
                map <void*, _data_copy*>::iterator k;
                for (k=_b1._copied_data.begin(); k!=_b1._copied_data.end(); k++){
                        free ((void*)(k->second));
                        k->second=NULL;
                }
                _b1._copied_data.clear();
        }

        pthread_mutex_unlock(&_valid_branch_mutex);
        pthread_mutex_unlock(&_is_running_mutex);

        //if (reset_all_logs) means that the common logs will be deleted

        if (reset_all_logs){

                _readers_log.clear();
        }
        else {

        //if (!reset_all_logs) only the data from B1 will be deleted

                if (!_readers_log.empty()){

                        for (it=_b1._read_data.begin(); it!=_b1._read_data.end(); it++){
                                if (_readers_log[*it]==PB){
                                        _readers_log[*it]=B2;
                                }
                                else if (_readers_log[*it]==B1){
                                        _readers_log.erase(*it);
                                }
                        }

                }
        }

        pthread_mutex_unlock(&_readers_log_mutex);

        _b1._read_data.clear();

        //if (!cancel) means that the thread should be restarted

        if (!cancel){
                pthread_attr_t attr;
                pthread_attr_init (&attr);
                pthread_attr_setschedpolicy(&attr, _sched_option);

                pthread_create (&_b1._thread, &attr, _b1._instructions, _b1._args);

        }


        pthread_mutex_unlock(&_b1._copied_data_mutex);

        pthread_mutex_unlock(&_b1._mutex);
}
else {

        pthread_mutex_lock(&_b2._mutex);
        pthread_mutex_lock(&_b2._copied_data_mutex);

        set <void*>::iterator it;

        pthread_mutex_lock(&_readers_log_mutex);

        pthread_mutex_lock(&_is_running_mutex);
        pthread_mutex_lock(&_valid_branch_mutex);
```

```
                        if (pthread_kill (_b2._thread, 0)==0)
                                pthread_cancel (_b2._thread);
                        _canceled_threads.insert(_b2._thread);

                        if (!_b2._copied_data.empty()){
                                map <void*, _data_copy*>::iterator k;
                                for (k=_b2._copied_data.begin(); k!=_b2._copied_data.end(); k++){
                                        free ((void*)(k->second));
                                        k->second=NULL;
                                }
                                _b2._copied_data.clear();
                        }


                        pthread_mutex_unlock(&_valid_branch_mutex);
                        pthread_mutex_unlock(&_is_running_mutex);

                        //if (reset_all_logs) means that the common logs will be deleted

                        if (reset_all_logs){

                                _readers_log.clear();

                        }
                        else {

                        //if (!reset_all_logs) only the data from B2 will be deleted

                                if (!_readers_log.empty()){

                                        for (it=_b2._read_data.begin(); it!=_b2._read_data.end(); it++){
                                                if (_readers_log[*it]==PB){
                                                        _readers_log[*it]=B1;
                                                }
                                                else if (_readers_log[*it]==B2){
                                                        _readers_log.erase(*it);
                                                }
                                        }

                                }
                        }


                        pthread_mutex_unlock(&_readers_log_mutex);

                        _b2._read_data.clear();

                        //if (!cancel) means that the thread should be restarted

                        if (!cancel){
                                pthread_attr_t attr;
                                pthread_attr_init (&attr);
                                pthread_attr_setschedpolicy(&attr, _sched_option);


                                pthread_create (&_b2._thread, &attr, _b2._instructions, _b2._args);

                        }


                        pthread_mutex_unlock(&_b2._copied_data_mutex);

                        pthread_mutex_unlock(&_b2._mutex);

                }
        };


public:

        //!
        //! function providing access to the shared data as a whole
        //!
        void*& get_shared_data (){

                pthread_mutex_lock(&_is_running_mutex);
                bool manages_pre_branch=!_pb._isExternal;
                if (!_is_running){

                        pthread_mutex_unlock(&_is_running_mutex);
                        return (void*&)_null_data; //the object is inactive

                }

                pthread_mutex_unlock(&_is_running_mutex);
                if (pthread_self()==_b1._thread || pthread_self()==_b2._thread){
                        return _shared_data;

                }
                if (manages_pre_branch){

                        if (pthread_self()==_pb._thread){
                                return _shared_data;
                        }
                        else{
                                return (void*&)_null_data; //invalid caller
                        }
                }

                return _shared_data; //unmanaged pre-branch or branch in deferred cancel
        };

        //!
        //! function allowing to validate b1 with argument==true
        //! b2 in the other case. Is required to keep expected control-flow
        //! consistency, otherwise only the pre-branch will affect the results.
        //!
        //! TO BE NOTED:
        //! * has to be called from the pre-branch section, keeping
        //! with control dependences.
        //!
        int validate_supposition (bool validation){

                pthread_mutex_lock(&_is_running_mutex);

                if (!_is_running){
                        pthread_mutex_unlock(&_is_running_mutex);
                        return -1; //the object is inactive
                }

                bool manages_pre_branch=!_pb._isExternal;

                pthread_t thrd_id=pthread_self();

                pthread_mutex_unlock(&_is_running_mutex);


                if (manages_pre_branch){

                        if (thrd_id!=_pb._thread){

                                return -1; //invalid caller
                        }
                }
                else if (thrd_id==_b1._thread || thrd_id==_b2._thread){

                        return -1; //invalid caller

                }

                map<void*, pthread_mutex_t>::iterator it;
                if (validation){//validates B1, cancels B2

                        pthread_mutex_lock(&_valid_branch_mutex);
                        _valid_branch=B1;
                        pthread_mutex_unlock(&_valid_branch_mutex);

                        _reset_branch(B2, false, true);

                }
                else {//validates B2, cancels B1

                        pthread_mutex_lock(&_valid_branch_mutex);
                        _valid_branch=B2;
                        pthread_mutex_unlock(&_valid_branch_mutex);

                        _reset_branch(B1, false, true);
                }
                return 0;

        };

        //!
        //! function to be called from the speculative branches
        //! and pre-branch, in order to read the shared data
        //! while keeping the expected sequential data consistency
        //!

        void* read_data (void* data_to_be_read, unsigned int size){

                pthread_mutex_lock(&_is_running_mutex);
```

```
if (!_is_running){
        pthread_mutex_unlock(&_is_running_mutex);
        return data_to_be_read; //the object is inactive, it's calling data is returned instead
                                     //of NULL, to prevent segmentation faults.
}
bool manages_pre_branch=!_pb._isExternal;
pthread_mutex_unlock(&_is_running_mutex);

void* retval;

if (manages_pre_branch){

        if (pthread_self()==_pb._thread){
                return data_to_be_read; //the pre-branch does a standard read
        }
}

//the branches need to make a copy or read an already existing copy, and log the reading.

pthread_mutex_lock(&_b1._mutex);
pthread_mutex_lock(&_b2._mutex);

pthread_t thrd_id=pthread_self();

if (thrd_id==_b1._thread){
        pthread_mutex_unlock(&_b2._mutex);
        pthread_mutex_lock(&_b1._copied_data_mutex);
        map <void*, _data_copy*>::iterator it;
        if (!_b1._copied_data.empty()){
                it=_b1._copied_data.find((void*)data_to_be_read);
                if (it!=_b1._copied_data.end()){
                        //if there is a copy, it should be returned instead of the
                        //value that the pre-branch has.
                        retval=(void*)malloc (it->second->_size);
                        memcpy (retval, &it->second->_data, it->second->_size);
                        pthread_mutex_unlock(&_b1._copied_data_mutex);
                        pthread_mutex_unlock(&_b1._mutex);
                        return retval;
                }
        }
        pthread_mutex_unlock(&_b1._copied_data_mutex);

        //if there is no copy, then the data is read and logged before returning


        if (_b1._read_data.find((void*)data_to_be_read)!=_b1._read_data.end()){ //If the data has already been read, it dc
                        pthread_mutex_unlock(&_b1._mutex);
                        return data_to_be_read;
        }

        bool data_mutex_on_hold=false;

        _b1._read_data.insert((void*)data_to_be_read);//the read is logged in the thread

        if (!_readers_log_mutexes.empty()){
                if (_readers_log_mutexes.find((void*)data_to_be_read)!=_readers_log_mutexes.end()){
                        pthread_mutex_lock(&_readers_log_mutexes.find((void*)data_to_be_read)->second);
                        data_mutex_on_hold=true;
                        if (_readers_log.find((void*)data_to_be_read)!=_readers_log.end()){
                                if (_readers_log.find((void*)data_to_be_read)->second==B2)
                                        _readers_log[_readers_log.find((void*)data_to_be_read)->first]=PB;
                                pthread_mutex_unlock(&_readers_log_mutexes.find((void*)data_to_be_read)->second);
                                pthread_mutex_unlock(&_b1._mutex);
                                return data_to_be_read;
                        }
                }
        }

        //if it's the first reading of the data, then it should be inserted in the log as a new entry
        if (!data_mutex_on_hold){
                pthread_mutex_t new_mutex;
                pthread_mutex_init (&new_mutex, NULL);
                _readers_log_mutexes.insert(pair <void*, pthread_mutex_t>((void*)data_to_be_read, new_mutex));
                pthread_mutex_lock(&_readers_log_mutexes.find((void*)data_to_be_read)->second);
        }
        pthread_mutex_lock(&_readers_log_mutex);
        _readers_log.insert(map <void*, _branch>::value_type((void*)data_to_be_read, B1));
        pthread_mutex_unlock(&_readers_log_mutex);
        pthread_mutex_unlock(&_readers_log_mutexes.find((void*)data_to_be_read)->second);
        pthread_mutex_unlock(&_b1._mutex);
        return data_to_be_read;

}
else if (thrd_id==_b2._thread){
        pthread_mutex_unlock(&_b1._mutex);
        pthread_mutex_lock(&_b2._copied_data_mutex);
        map <void*,_data_copy*>::iterator it;
```

```
                if (!_b2._copied_data.empty()){
                        it=_b2._copied_data.find((void*)data_to_be_read);
                        if (it!=_b2._copied_data.end()){
                                //if there is a copy, it should be returned instead of the
                                //value that the pre-branch has.
                                retval=(void*)malloc (it->second->_size);
                                memcpy (retval, &it->second->_data, it->second->_size);
                                pthread_mutex_unlock(&_b2._copied_data_mutex);
                                pthread_mutex_unlock(&_b2._mutex);
                                return retval;
                        }
                }
                pthread_mutex_unlock(&_b2._copied_data_mutex);

                //if there is no copy, then the data is read and logged before returning

                bool data_mutex_on_hold=false;

                if (_b2._read_data.find((void*)data_to_be_read)!=_b2._read_data.end()){ //If the data has already been read, it do
                                pthread_mutex_unlock(&_b2._mutex);
                                return data_to_be_read;
                }

                _b2._read_data.insert((void*)data_to_be_read);//the read is logged in the thread

                if (!_readers_log_mutexes.empty()){
                        if (_readers_log_mutexes.find((void*)data_to_be_read)!=_readers_log_mutexes.end()){
                                pthread_mutex_lock(&_readers_log_mutexes.find((void*)data_to_be_read)->second);
                                data_mutex_on_hold=true;
                                if (_readers_log.find((void*)data_to_be_read)!=_readers_log.end()){
                                        if (_readers_log.find((void*)data_to_be_read)->second==B1)
                                                _readers_log[_readers_log.find((void*)data_to_be_read)->first]=PB;
                                        pthread_mutex_unlock(&_readers_log_mutexes.find((void*)data_to_be_read)->second);
                                        pthread_mutex_unlock(&_b2._mutex);
                                        return data_to_be_read;
                                }
                        }
                }

                //if it's the first reading of the data, then it should be inserted in the log as a new entry
                if (!data_mutex_on_hold){
                        pthread_mutex_t new_mutex;
                        pthread_mutex_init (&new_mutex, NULL);
                        _readers_log_mutexes.insert(pair <void*, pthread_mutex_t>((void*)data_to_be_read, new_mutex));
                        pthread_mutex_lock(&_readers_log_mutexes.find((void*)data_to_be_read)->second);
                }

                pthread_mutex_lock(&_readers_log_mutex);
                _readers_log.insert(map <void*, _branch>::value_type((void*)data_to_be_read, B2));
                pthread_mutex_unlock(&_readers_log_mutex);
                pthread_mutex_unlock(&_readers_log_mutexes.find((void*)data_to_be_read)->second);

                pthread_mutex_unlock(&_b2._mutex);
                return data_to_be_read;

        }
        else {
                pthread_mutex_unlock(&_b2._mutex);
                pthread_mutex_unlock(&_b1._mutex);
        }

        if (!manages_pre_branch){
                return data_to_be_read;//un-managed pre-branch
        }
        return data_to_be_read; //invalid caller or branch in deferred cancel.
                                     //it's calling data is returned instead of NULL, to prevent segmentation faults.
};

//!
//! function to be called from the speculative branches
//! and pre-branch, in order to write the shared data
//! while keeping the expected sequential data consistency
//!
int write_data(void*& data_to_be_written_upon, void* data_to_write, unsigned int size){

        pthread_mutex_lock(&_is_running_mutex);
        if (!_is_running){
                pthread_mutex_unlock(&_is_running_mutex);
                return -1; //the object is inactive.
        }
        bool manages_pre_branch=!_pb._isExternal;
        pthread_mutex_unlock(&_is_running_mutex);

        bool is_pre_branch=false;
        if (manages_pre_branch){
                if (pthread_self()==_pb._thread){
```

```
                is_pre_branch=true;
        }
}

pthread_mutex_lock(&_valid_branch_mutex);
_branch aux_valid_branch=_valid_branch;
pthread_mutex_unlock(&_valid_branch_mutex);


if ((!manages_pre_branch) && pthread_self()!=_b1._thread && pthread_self()!=_b2._thread){
        is_pre_branch=true;
        pthread_mutex_lock(&_readers_log_mutex);
        if (_canceled_threads.find(pthread_self())!=_canceled_threads.end()){
                        is_pre_branch=false;
        }
        pthread_mutex_unlock(&_readers_log_mutex);

}
pthread_mutex_lock (&_b1._mutex);
pthread_mutex_lock (&_b2._mutex);

pthread_t thrd_id=pthread_self();

if (is_pre_branch){

        //for the pre-branch, the data is written and then the branches are restarted if they have
        //read an invalid previous value, i.e. a delinquent load. This means that a true dependency
        //violation has ocurred (RAW).

        map<void*, _branch>::iterator it;

        if (_readers_log_mutexes.find((void*)data_to_be_written_upon)!=_readers_log_mutexes.end()){
                pthread_mutex_lock(&_readers_log_mutexes.find((void*)data_to_be_written_upon)->second);
        }
        else{
                pthread_mutex_t new_mutex;
                pthread_mutex_init (&new_mutex, NULL);
                _readers_log_mutexes.insert(pair <void*, pthread_mutex_t>((void*)data_to_be_written_upon, new_mutex));
                pthread_mutex_lock(&_readers_log_mutexes.find((void*)data_to_be_written_upon)->second);
        }

        memcpy ((void*)data_to_be_written_upon, (void*)&data_to_write, size);

        pthread_mutex_unlock (&_b2._mutex);
        pthread_mutex_unlock (&_b1._mutex);


        it=_readers_log.find((void*)data_to_be_written_upon);

        if (!_readers_log.empty() && it!=_readers_log.end()){  //a true dependency violation has ocurred (RAW)

                _branch delinquent_branches=it->second;

                pthread_mutex_unlock(&_readers_log_mutexes.find((void*)data_to_be_written_upon)->second);

                //dependency violation resetting will be filtered only to the valid branch
                //since the cancelation does not delete the logged readings of a canceled
                //branch (i.e. it is a lazy cancelation).

                // aux_valid_branch==PB means that no branch has been validated, yet.
                if (aux_valid_branch==PB){

                        if (delinquent_branches==PB){
                                _reset_branch (PB, true, false); //the common logs can be reseted
                        }
                        else if (delinquent_branches==B1){
                                _reset_branch (B1, false, false);//the common logs should not be reseted
                        }
                        else{
                                _reset_branch (B2, false, false);//the common logs should not be reseted
                        }

                }
                else if (aux_valid_branch==B1 && delinquent_branches!=B2){
                //the common logs should not be reseted, only B1 should be restarted.
                        _reset_branch (B1, false, false);
                }

                else if (aux_valid_branch==B2 && delinquent_branches!=B1){
                //the common logs should not be reseted, only B2 should be restarted.
                        _reset_branch (B2, false, false);
                }
                else {
                        //if the delinquent branch is invalid and has already been canceled,
                        //then only the log entry for the related data should be deleted.
                        _readers_log.erase(it->first);
                        }
                }
                else{
                        pthread_mutex_unlock(&_readers_log_mutexes.find((void*)data_to_be_written_upon)->second);
                }
                return 0;
        }
        else if ((thrd_id==_b1._thread) && (aux_valid_branch!=B2)){ //for the branches, the data is written in the copy.
                pthread_mutex_unlock (&_b2._mutex);

                map <void*, _data_copy*>::iterator it;
                bool copy_found=false;
                pthread_mutex_lock(&_b1._copied_data_mutex);
                if (!_b1._copied_data.empty()){
                        it=_b1._copied_data.find((void*)data_to_be_written_upon);
                        if (it!=_b1._copied_data.end()){
                                copy_found=true;
                        }
                }
                //if a copy is found, it is written over; else, a new copy is made.
                if (copy_found){
                        _b1._copied_data[it->first]->_data=(void*)data_to_write;
                }
                else {
                        _data_copy* copy;
                        copy= (struct _data_copy*) malloc (sizeof(struct _data_copy));
                        copy->_size=size;
                        copy->_data= (void*) malloc (size);
                        memcpy (copy->_data, (void*)&data_to_write, size);
                        _b1._copied_data.insert(pair<void*, _data_copy*>((void*)data_to_be_written_upon, copy));
                        free(copy->_data);
                }
                pthread_mutex_unlock(&_b1._copied_data_mutex);
                pthread_mutex_unlock(&_b1._mutex);
                return 0;
        }
        else if ((thrd_id==_b2._thread) && (aux_valid_branch!=B1)){
                pthread_mutex_unlock (&_b1._mutex);

                map <void*, _data_copy*>::iterator it;
                bool copy_found=false;
                pthread_mutex_lock(&_b2._copied_data_mutex);
                if (!_b2._copied_data.empty()){
                        it=_b2._copied_data.find((void*)data_to_be_written_upon);
                        if (it!=_b2._copied_data.end()){
                                copy_found=true;
                        }
                }
                //if a copy is found, it is written over; else, a new copy is made.
                if (copy_found){
                        _b2._copied_data[it->first]->_data=(void*)data_to_write;
                }
                else {

                        _data_copy* copy;
                        copy= (struct _data_copy*) malloc (sizeof(struct _data_copy));
                        copy->_size=size;
                        copy->_data= (void*) malloc (size);
                        memcpy (copy->_data, (void*)&data_to_write, size);
                        _b2._copied_data.insert(pair<void*, _data_copy*>((void*)data_to_be_written_upon, copy));
                        free(copy->_data);

                }
                pthread_mutex_unlock(&_b2._copied_data_mutex);
                pthread_mutex_unlock(&_b2._mutex);
                return 0;
        }
        else{
                pthread_mutex_unlock (&_b2._mutex);
                pthread_mutex_unlock (&_b1._mutex);
        }
        return -1;//invalid caller.

};

//!
//! speculate: a complete function that takes the instructions and arguments
//! of the branches and pre-branch; and starts their speculatively parallel
//! execution, while maintaining the sequential consistency. This function
//! returns in the &shared_data, the results of the computation.
//!
//! TO BE NOTED:
//!     * In this version of the function, the object is in control of the pre-
//!     branch. This means that on invocation, the caller blocks until the pre-
//!     branch and validated branch complete their execution.
//!
```

```
int speculate (void*& shared_data, void* (fpb)(void*), void* const_args_pb, void* (fb1)(void*), void* const_args_b1, void* (fb2)(v
void* const_args_b2, int sched_policy){

        pthread_mutex_lock(&_is_running_mutex);
        if (_is_running){
                pthread_mutex_unlock(&_is_running_mutex);
                return -1;//the object is already running, hence a new speculation cannot run.
        }
        _is_running=true;
        bool had_pre_branch=!_pb._isExternal;
        _pb._isExternal=false;
        bool is_first_run=_first_run;
        pthread_mutex_unlock(&_is_running_mutex);


        if (!is_first_run){ //the object has to be reseted.

                if (had_pre_branch){
                        pthread_mutex_lock(&_pb._mutex);
                        if (pthread_kill (_pb._thread, 0)==0){
                                pthread_cancel (_pb._thread);
                                _canceled_threads.insert(_pb._thread);
                        }
                        pthread_mutex_unlock(&_pb._mutex);
                }

                pthread_mutex_lock(&_b1._mutex);
                if (pthread_kill (_b1._thread, 0)==0){
                        pthread_cancel (_b1._thread);
                        _canceled_threads.insert(_b1._thread);
                }
                pthread_mutex_unlock(&_b1._mutex);

                pthread_mutex_lock(&_b2._mutex);
                if (pthread_kill (_b2._thread, 0)==0){
                        pthread_cancel (_b2._thread);
                        _canceled_threads.insert(_b2._thread);
                }
                pthread_mutex_unlock(&_b2._mutex);


                pthread_mutex_lock(&_valid_branch_mutex);
                _valid_branch=PB;
                pthread_mutex_unlock(&_valid_branch_mutex);

                map <void*, _data_copy*>::iterator k;
                pthread_mutex_lock(&_b1._copied_data_mutex);
                if (!_b1._copied_data.empty()){
                        for (k=_b1._copied_data.begin(); k!=_b1._copied_data.end(); k++){
                                free ((void*)(k->second));
                                k->second=NULL;
                        }
                        _b1._copied_data.clear();
                }
                pthread_mutex_unlock(&_b1._copied_data_mutex);

                pthread_mutex_lock(&_b2._copied_data_mutex);
                if (!_b2._copied_data.empty()){
                        for (k=_b2._copied_data.begin(); k!=_b2._copied_data.end(); k++){
                                free ((k->second));
                                k->second=NULL;
                        }
                        _b2._copied_data.clear();
                }
                pthread_mutex_unlock(&_b2._copied_data_mutex);

                if (!_canceled_threads.empty()){
                        set<pthread_t>::iterator it;
                        for (it=_canceled_threads.begin(); it!=_canceled_threads.end(); it++){
                                int a;
                                do{
                                         a= pthread_kill (*it, 0);
                                } while (a==0);
                        }


                        map<void*, pthread_mutex_t>::iterator it2;
                        for (it2=_readers_log_mutexes.begin(); it2!=_readers_log_mutexes.end(); it2++){
                                pthread_mutex_lock(&it2->second);
                        }

                        pthread_mutex_lock(&_readers_log_mutex);
                        _readers_log.clear();
                        _canceled_threads.clear();
                        for (it2=_readers_log_mutexes.begin(); it2!=_readers_log_mutexes.end(); it2++){
                                pthread_mutex_destroy(&it2->second);
                        }
                        _readers_log_mutexes.clear();
                        pthread_mutex_unlock(&_readers_log_mutex);
                }
                else {
                        if (!_readers_log.empty()){
                                map<void*, pthread_mutex_t>::iterator it2;
                                for (it2=_readers_log_mutexes.begin(); it2!=_readers_log_mutexes.end(); it2++){
                                        pthread_mutex_lock(&it2->second);
                                }

                                pthread_mutex_lock(&_readers_log_mutex);
                                _readers_log.clear();
                                for (it2=_readers_log_mutexes.begin(); it2!=_readers_log_mutexes.end(); it2++){
                                        pthread_mutex_destroy(&it2->second);
                                }
                                _readers_log_mutexes.clear();
                                pthread_mutex_unlock(&_readers_log_mutex);
                        }
                }
        }

        _shared_data=shared_data;
        _b1._instructions=fb1;
        _b2._instructions=fb2;
        _b1._args=const_args_b1;
        _b2._args=const_args_b2;

        pthread_attr_t attr;
        pthread_attr_init (&attr);

        if (sched_policy== SCHED_FIFO||sched_policy==SCHED_RR||sched_policy== SCHED_OTHER){
                _sched_option=sched_policy;
        }
        else {
                _sched_option=SCHED_RR;
        }

        pthread_attr_setschedpolicy(&attr, _sched_option);


        pthread_mutex_lock(&_readers_log_mutex);
        pthread_mutex_lock(&_b2._mutex);
        pthread_mutex_lock(&_b1._mutex);
        pthread_mutex_lock(&_pb._mutex);
        int a=pthread_create (&_pb._thread, &attr, fpb, const_args_pb);
        pthread_mutex_unlock(&_pb._mutex);

        if (a==0){
                a=pthread_create (&_b1._thread, &attr, fb1, const_args_b1);
                pthread_mutex_unlock(&_b1._mutex);
                if (a==0){
                        a=pthread_create (&_b2._thread, &attr, fb2, const_args_b2);
                        pthread_mutex_unlock(&_b2._mutex);
                        pthread_mutex_unlock(&_readers_log_mutex);

                        if (a==0){
                                pthread_mutex_lock(&_is_running_mutex);
                                _first_run=false;
                                pthread_mutex_unlock(&_is_running_mutex);

                                do{
                                        a=pthread_join (_pb._thread, NULL);

                                } while (a==0);

                                if (_valid_branch!=PB){

                                        if (_valid_branch==B1){

                                                do{

                                                        a=pthread_join (_b1._thread, NULL);

                                                } while (a==0);


                                                pthread_mutex_lock(&_readers_log_mutex);
                                                pthread_mutex_lock(&_b1._copied_data_mutex);

                                                if (!_b1._copied_data.empty()){

                                                //The data that b1 changed should be copied in the &shared_data
                                                        map <void*, _data_copy*>::iterator i;
                                                        for (i=_b1._copied_data.begin(); i!=_b1._copied_data.end(); i++){
                                                                memcpy((void*&)const_cast<void*&>(i->first), (void*)&(i->second->_
                                                                free ((void*)(i->second));
                                                                i->second=NULL;
                                                        }
```

```
                                                        _b1._copied_data.clear();
                                                }
                                                pthread_mutex_unlock(&_b1._copied_data_mutex);

                                                _readers_log.clear();

                                                pthread_mutex_unlock(&_readers_log_mutex);

                                                pthread_mutex_lock(&_is_running_mutex);
                                                _is_running=false;
                                                pthread_mutex_unlock(&_is_running_mutex);

                                                return 0;
                                        }
                                        else{
                                                do{

                                                        a=pthread_join (_b2._thread, NULL);

                                                } while (a==0);

                                                pthread_mutex_lock(&_readers_log_mutex);
                                                pthread_mutex_lock(&_b2._copied_data_mutex);

                                                if (!_b2._copied_data.empty()){

                                                //The data that b2 changed should be copied in the &shared_data

                                                        map <void*, _data_copy*>::iterator i;
                                                        for (i=_b2._copied_data.begin(); i!=_b2._copied_data.end(); i++){
                                                                memcpy((void*&)const_cast<void*&>(i->first), (void*)&(i->second->_
                                                                free ((void*)(i->second));
                                                                i->second=NULL;
                                                        }
                                                        _b2._copied_data.clear();
                                                }

                                                pthread_mutex_unlock(&_b2._copied_data_mutex);
                                                _readers_log.clear();

                                                pthread_mutex_unlock(&_readers_log_mutex);

                                                pthread_mutex_lock(&_is_running_mutex);
                                                _is_running=false;
                                                pthread_mutex_unlock(&_is_running_mutex);

                                                return 0;
                                        }
                                }
                                else{

                                        _reset_branch(PB, true, true);//B1 and B2 are canceled.

                                        pthread_mutex_lock(&_is_running_mutex);
                                        _is_running=false;
                                        pthread_mutex_unlock(&_is_running_mutex);

                                        return -1;//No branch was validated.
                                }
                        }
                }
                else {
                        pthread_mutex_unlock(&_b2._mutex);
                        _reset_branch (B1, true, true); //If B2 was not created, B1 is canceled.
                }
        }
        else{ //Could not create B1-
                pthread_mutex_unlock(&_b1._mutex);
                pthread_mutex_unlock(&_b2._mutex);
        }

        pthread_mutex_lock(&_is_running_mutex);
        _is_running=false;
        pthread_mutex_unlock(&_is_running_mutex);

        return -1; //Some thread in the class could not be created.

};

//!
//! speculate: a function that takes the instructions and arguments
//! of the branches , and starts their speculatively parallel execution,
//! while maintaining the sequential consistency with an unmanaged pre-branch.
//!
//! TO BE NOTED:
//!    * In this version of the function, the object is not in control of the pre-
//!    branch. This means that on invocation, the caller only blocks for the creation
//!    of the branches, and can resume it's execution as a possible pre-branch, until
//!    calling get_results(), when the caller blocks until the valid branch returns.
//!
int speculate (void*& shared_data, void* (fb1)(void*), void* const_args_b1, void* (fb2)(void*), void* const_args_b2, int sched_pol

        pthread_mutex_lock(&_is_running_mutex);
        if (_is_running){
                pthread_mutex_unlock(&_is_running_mutex);
                return -1; //The object is inactive.
        }
        _is_running=true;
        bool had_pre_branch=!_pb._isExternal;
        _pb._isExternal=true;
        bool is_first_run=_first_run;
        pthread_mutex_unlock(&_is_running_mutex);

        if (!is_first_run){ //If this is not the first run, then the object should be reseted.
                if (had_pre_branch){
                        pthread_mutex_lock(&_pb._mutex);
                        if (pthread_kill (_pb._thread, 0)==0){
                                pthread_cancel (_pb._thread);
                                _canceled_threads.insert(_pb._thread);
                        }
                        pthread_mutex_unlock(&_pb._mutex);
                }

                pthread_mutex_lock(&_b1._mutex);
                if (pthread_kill (_b1._thread, 0)==0){
                        pthread_cancel (_b1._thread);
                        _canceled_threads.insert(_b1._thread);
                }
                pthread_mutex_unlock(&_b1._mutex);

                pthread_mutex_lock(&_b2._mutex);
                if (pthread_kill (_b2._thread, 0)==0){
                        pthread_cancel (_b2._thread);
                        _canceled_threads.insert(_b2._thread);
                }
                pthread_mutex_unlock(&_b2._mutex);


                pthread_mutex_lock(&_valid_branch_mutex);
                _valid_branch=PB;
                pthread_mutex_unlock(&_valid_branch_mutex);

                map <void*, _data_copy*>::iterator k;
                pthread_mutex_lock(&_b1._copied_data_mutex);
                if (!_b1._copied_data.empty()){
                        for (k=_b1._copied_data.begin(); k!=_b1._copied_data.end(); k++){
                                free ((void*)(k->second));
                                k->second=NULL;
                        }
                        _b1._copied_data.clear();
                }
                pthread_mutex_unlock(&_b1._copied_data_mutex);

                pthread_mutex_lock(&_b2._copied_data_mutex);
                if (!_b2._copied_data.empty()){
                        for (k=_b2._copied_data.begin(); k!=_b2._copied_data.end(); k++){
                                free ((void*)(k->second));
                                k->second=NULL;
                        }
                        _b2._copied_data.clear();
                }
                pthread_mutex_unlock(&_b2._copied_data_mutex);

                if (!_canceled_threads.empty()){
                        set<pthread_t>::iterator it;
                        for (it=_canceled_threads.begin(); it!=_canceled_threads.end(); it++){
                                int a;
                                do{
                                         a= pthread_kill (*it, 0);
                                } while (a==0);
                        }

                        map<void*, pthread_mutex_t>::iterator it2;
                        for (it2=_readers_log_mutexes.begin(); it2!=_readers_log_mutexes.end(); it2++){
                                pthread_mutex_lock(&it2->second);
                        }

                        pthread_mutex_lock(&_readers_log_mutex);
                        _readers_log.clear();
                        _canceled_threads.clear();
                        for (it2=_readers_log_mutexes.begin(); it2!=_readers_log_mutexes.end(); it2++){
                                pthread_mutex_destroy(&it2->second);
                        }
                        _readers_log_mutexes.clear();
```

```
			pthread_mutex_unlock(&_readers_log_mutex);
		}
		else {
			if (!_readers_log.empty()){
				map<void*, pthread_mutex_t>::iterator it2;
				for (it2=_readers_log_mutexes.begin(); it2!=_readers_log_mutexes.end(); it2++){
					pthread_mutex_lock(&it2->second);
				}

				pthread_mutex_lock(&_readers_log_mutex);
				_readers_log.clear();
				for (it2=_readers_log_mutexes.begin(); it2!=_readers_log_mutexes.end(); it2++){
					pthread_mutex_destroy(&it2->second);
				}
				_readers_log_mutexes.clear();
				pthread_mutex_unlock(&_readers_log_mutex);
			}
		}
	}

	_shared_data=shared_data;
	_b1._instructions=fb1;
	_b2._instructions=fb2;
	_b1._args=const_args_b1;
	_b2._args=const_args_b2;

	pthread_attr_t attr;
	pthread_attr_init (&attr);

	if (sched_policy== SCHED_FIFO||sched_policy==SCHED_RR||sched_policy== SCHED_OTHER){
		_sched_option=sched_policy;
	}
	else {
		_sched_option=SCHED_RR;
	}

	pthread_attr_setschedpolicy(&attr, _sched_option);


	pthread_mutex_lock(&_readers_log_mutex);
	pthread_mutex_lock(&_b2._mutex);
	pthread_mutex_lock(&_b1._mutex);
	int a=pthread_create (&_b1._thread, &attr, fb1, const_args_b1);
	pthread_mutex_unlock(&_b1._mutex);
	if (a!=0){
		pthread_mutex_unlock(&_b2._mutex);
		pthread_mutex_unlock(&_readers_log_mutex);
		pthread_mutex_lock(&_is_running_mutex);
		_is_running=false;
		pthread_mutex_unlock(&_is_running_mutex);
		return -1; //B1 could not be created.
	}
	a=pthread_create (&_b2._thread, &attr, fb2, const_args_b2);
	if (a!=0){
		pthread_mutex_unlock(&_b2._mutex);
		pthread_mutex_unlock(&_readers_log_mutex);
		_reset_branch (B1, true, true);

		pthread_mutex_lock(&_is_running_mutex);
		_is_running=false;
		pthread_mutex_unlock(&_is_running_mutex);
		return -1;//B2 could not be created.
	}
	pthread_mutex_unlock(&_b2._mutex);
	pthread_mutex_unlock(&_readers_log_mutex);

	pthread_mutex_lock(&_is_running_mutex);
	_first_run=false;
	pthread_mutex_unlock(&_is_running_mutex);

	return 0;
};

//!
//! function to get the results of the speculation once the un-managed pre-
//! branch has ended it's execution.
//!
//! TO BE NOTED:
//!    * If the pre-branch does not validate the supposition, then none of
//!    the results of the branches will be accepted.
//!
int get_results(){
	pthread_mutex_lock(&_is_running_mutex);
	if (!_is_running || !_pb._isExternal){
		pthread_mutex_unlock(&_is_running_mutex);
		return -1;//The object is inactive.
	}
```

```
                pthread_mutex_unlock(&_is_running_mutex);

                pthread_mutex_lock(&_valid_branch_mutex);
                _branch aux_valid_branch=_valid_branch;
                pthread_mutex_unlock(&_valid_branch_mutex);

                int a;

                if (aux_valid_branch==PB){
                        _reset_branch(PB, true, true);

                        pthread_mutex_lock(&_is_running_mutex);
                        _is_running=false;
                        pthread_mutex_unlock(&_is_running_mutex);
                        return -1; //No branch was validated.
                }
                else if (aux_valid_branch==B1){
                        do {

                                a=pthread_join (_b1._thread, NULL);

                        } while (a==0);

                        pthread_mutex_lock(&_readers_log_mutex);
                        pthread_mutex_lock(&_b1._copied_data_mutex);
                        if (!_b1._copied_data.empty()){
                                //The data that b1 changed should be copied in the &shared_data
                                map <void*, _data_copy*>::iterator i;
                                for (i=_b1._copied_data.begin(); i!=_b1._copied_data.end(); i++){
                                        memcpy((void*&)const_cast<void*&>(i->first), (void*)&(i->second->_data), i->second->_size);
                                        free ((void*)(i->second));
                                        i->second=NULL;
                                }
                                _b1._copied_data.clear();
                        }
                        pthread_mutex_unlock(&_b1._copied_data_mutex);
                        _readers_log.clear();
                        pthread_mutex_unlock(&_readers_log_mutex);

                        pthread_mutex_lock(&_is_running_mutex);
                        _is_running=false;
                        pthread_mutex_unlock(&_is_running_mutex);
                        return 0;
                }
                else {
                        do {

                                a=pthread_join (_b2._thread, NULL);

                        } while (a==0);

                        pthread_mutex_lock(&_readers_log_mutex);
                        pthread_mutex_lock(&_b2._copied_data_mutex);
                        if (!_b2._copied_data.empty()){

                                //The data that b2 changed should be copied in the &shared_data
                                map <void*, _data_copy*>::iterator i;

                                for (i=_b2._copied_data.begin(); i!=_b2._copied_data.end(); i++){
                                        memcpy((void*&)const_cast<void*&>(i->first), (void*)&(i->second->_data), i->second->_size);
                                        free ((void*)(i->second));
                                        (*i).second=NULL;
                                }
                                _b2._copied_data.clear();
                        }
                        pthread_mutex_unlock(&_b2._copied_data_mutex);
                        _readers_log.clear();
                        pthread_mutex_unlock(&_readers_log_mutex);

                        pthread_mutex_lock(&_is_running_mutex);
                        _is_running=false;
                        pthread_mutex_unlock(&_is_running_mutex);
                        return 0;
                }

        };


};
```