Coordinator based failure/recovery simulation framework

Tsatsarakis Myron

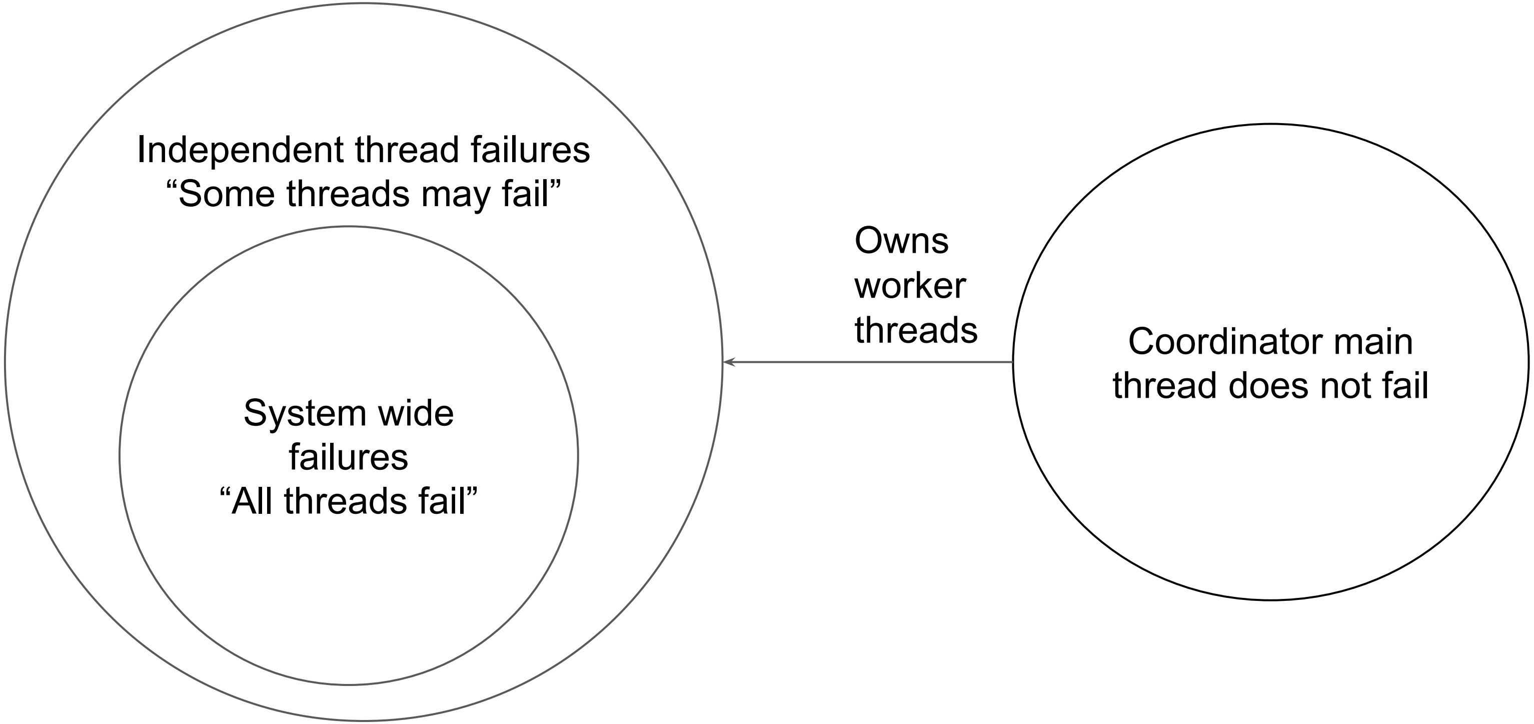
System wide failures

- Hard to simulate
  - Turn off machine?
  - Disconnect power cable?

Framework idea

- The coordinator is running on the main thread
- It’s worker threads are used to run the experiments where the crash/recovery scenarios are simulated
- The coordinator can terminate it’s worker threads to simulate a failure event
    - `pthread_cancel(thread.native_handle())`
- The coordinator can restart it’s worker threads to simulate a recovery event
    - `failedThread = std::jthread(RecoveryFunction)`

Both failure modes are encapsulated



Regarding thread resurrection

- We cannot reuse a terminated thread object
  - We can assign a new one in its place instead
- Entry point (code) of resurrected thread?
  - Depends on the system implementation
  - May restart it’s operation
  - May continue where it has left off
  - May execute a recovery operation

Event system for failure/recovery

- Failures are simulated as events appended to an event queue
- The event loop is run on the main thread
- Example events
  - Timer based thread failure → The coordinator terminates a thread after a time duration has passed
  - System state based thread crash → The coordinator terminates a thread based on a condition regarding the system state
- Same principles apply for recovery events

Coordinator runtime draft

Input → preparation and experiment operations, number of worker threads

State → worker threads, event queue, …

```
Run(){
    Create Threads
    Assign experiment operations to threads and execute
    While (eventQueue.IsEmpty() == false)
        Process event
    Wait on each live thread
}
```

Experiment templates/examples

- Base runtime
  - An experiment without failure/recovery
- Recovery runtime benchmark
  - Only the recovery function is measured by the user
- System wide failure/recovery
  - All threads fail, then the same number of threads are restarted, running recovery code
- Partial system failure/recovery
  - Some threads fail, then are restarted running recovery code
- State based failure/recovery
  - Eg. Some threads fail after our data structure contains X elements
  - Eg. Some threads are restarted after it contains Y elements

Implementing thread failures

Coordinator	**Worker**
Cooperative token based	
Stop = true	If stop: terminate else: continue
Cooperative signal based	
signal_send(stop)	signal_wait(stop) terminate
signal_send(stop)	signal_wait(stop) sleep(then is resurrected)
Forced	
pthead_cancel(thread)	Gets canceled

- Cooperative methods require user responsibility for synchronous handling on worker thread
- “It’s not an abrupt failure if I can queue and handle it at certain time points”

Implementing a Recoverable Thread

- Assigned a fixed id
- The fixed id does not change upon failure or restart
- **The lifetime of a recoverableThread includes failure/recovery events acted upon it.**
- Recoverability
  - Overwrite the crashed thread with a newly created one

```
struct RecoverableThread {
    std::jthread thread;
    int fixedId;
};
```

- End of experiment run condition
  - A std::latch is initialized with the input number of worker threads
  - A recoverableThread ends it’s lifetime upon reaching the latch

Coordinator runtime draft

```
Coordinator(){
     Create recoverableThreads based on input
     latch.Init(recoverableThreads.Size());
     PrepareExperiment(); //adds failure/recovery events to eventQueue
     For each thread:
          thread.RunAsync(Worker);
     While eventQueue.IsEmpty() == false:
          Process event //send faulure/recovery events to threads
     For each thread:
          thread.Join();
}
Worker(){
     InputExperiment();
     latch.ArriveAndWait();
}
```

Appending Failure/Recovery events to queue

- Typically occurs on
  - Experiment preparation → User prepares a sequence of events
  - During event handling → Handling an event triggers another event
  - Triggered by timers → More on that later…
- Failure
  - `eventQueue.append([&](){pthread_cancel(recoverableThread[2]);});`
  - `eventQueue.append([&](){if(db.IsFull())pthread_cancel(recoverableThread[7]);});`
- Recovery
  - `eventQueue.append([&](){recoverableThread[2]=std::jthread(InsertWorkload);});`
  - `eventQueue.append([&](){if(db.IsFull())recoverableThread[7]=std::jthread(Recover);});`

Timer based events

- Sometimes we want to schedule failure/recovery events to happen on certain intervals
- Timer threads asynchronously append an event after a duration

```
StartTimer(eventFunction, duration, repetitions){
std::jthread([&](){
For(i = 0 -> repetitions):
    sleep(duration);
    eventQueue.append(eventFunction);
}).detach();}

StartTimer([&](){cancel(thread[4]);restart(thread[4]);}, 10s, 3);
```

Coordinator runtime draft

```
Coordinator(){
     recoverableThreads = { … }; //based on input
     latch.Init(recoverableThreads.Size());
     PrepareExperiment();
     For each thread in recoverableThreads:
          thread.RunAsync(Worker);
     While !done:
          eventQueue.WaitForEvent();
          events = eventQueue.DequeueAll();
          For each event in events: event();
     For each thread in recoverableThreads:
          thread.Join();
}
```

```
Worker(){
     RunExperiment();
     latch.ArriveAndWait();
     std::call_once({done=true;});
}
```

Research user requirements

- Determine whether persistent concurrent data structure authors want to use this tool
- Is my experiment abstraction and usage scenarios convenient
- Access experiment codes of current work
- LCRQueue experiments by Mallis, simulate usage scenarios
- Friday

Optimization Note: Thread CPU affinity

- Coordinator thread → **sleeps** while waiting for an event
- Timer threads → **sleeps** for its intended duration
- Event handling and event appending code time cost is negligible
- Regardless
- We reserve a CPU core to exclusively pin the coordinator and timer threads
- We pin the rest of the worker threads, to the remaining CPU cores

Alternative idea: Threads as Processes

- Linux processes or MPI processes
- Need to be able to receive signals asynchronously
  - send(stop) → when received, terminate

- POSIX timers generate signals. Signals are process-wide
- Man page: POSIX.1 also requires that threads share a range of other attributes (i.e., these attributes are process-wide rather than per-thread):
  - signal dispositions
- Man page: The signal disposition is a per-process attribute: in a multithreaded application, the disposition of a particular signal is the same for all threads
- ChatGPT: In POSIX threads (pthreads), you cannot guarantee which specific thread will handle a signal when it is received. When a signal is delivered to a process, the operating system scheduler decides which thread in the process will handle the signal. This is typically the main thread, but it can be any thread in the process.

LCRQueue Experiment structure

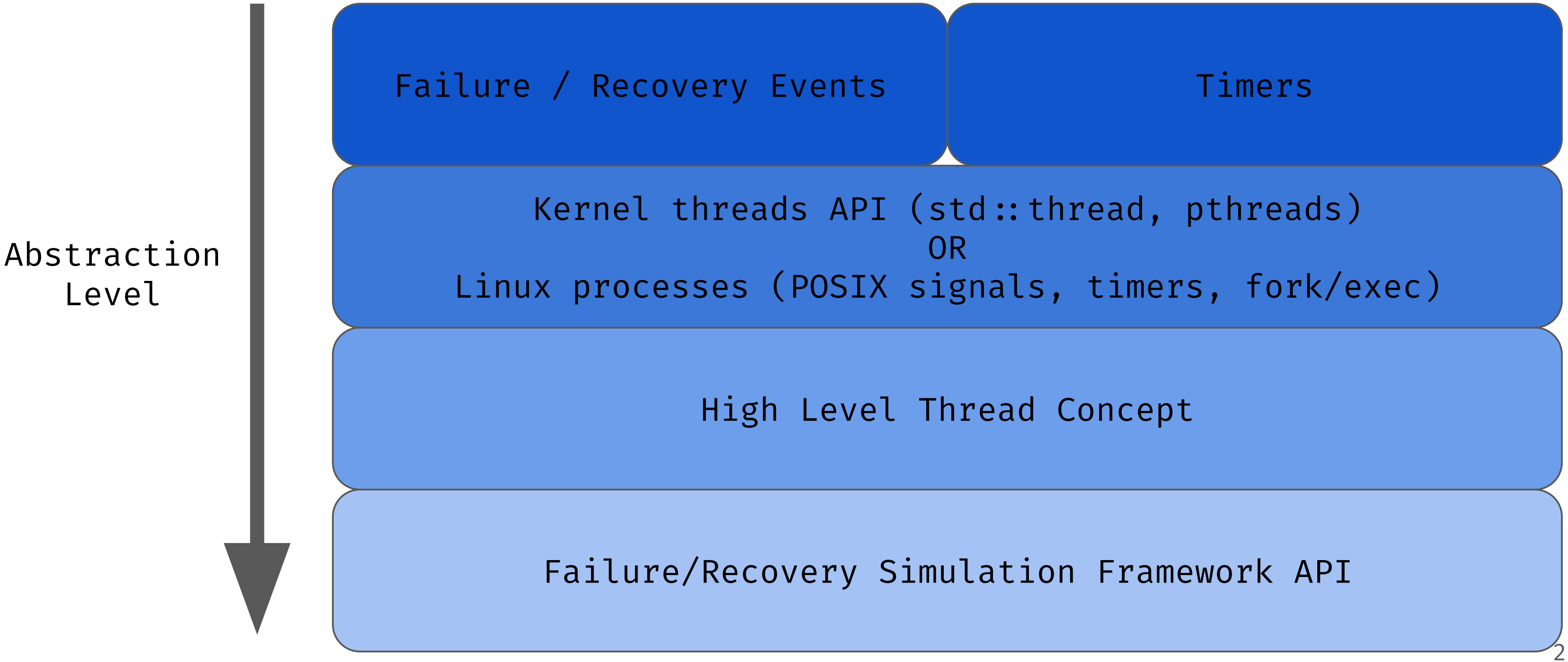
- Perform operations, then system wide failure
- Single threaded Recovery ← This is timed
  - Optional perform operations, then system wide failure
- Reset experiment state

LCRQueue experimentation requirements

- System wide failure based on system state
  - Fail after a number of completed operations, repeat 5 times
- System wide failure based on timer
  - Optional system state serves as an upper bound fail safe
  - Fail after 3 secs, recover after 3 secs, repeat 5 times
  - Fail after 3 inserts, recover immediately, repeat 5 times
  - Fail after 2 hours, recover after 5 mins, repeat after 5 times. Inform me after 1.000.000 inserts or after occupied RAM > 10GB
- System wide failure by keyboard interrupt
  - Useful for debugging and demonstration
  - “I simulate a power-failure when I press ‘Space’”
- Independent thread failures are viable, if the Recovery code takes into account other threads operating in the queue at the same time

Importance of a Coordinator based framework

- Experimental Expressiveness to the user
- Time related context where failure/recovery events are applied
  - User expresses failure/recovery scenarios in relation to the time flow of the experiment
- Centralized management of threads/processes by framework
  - How else can I manage thread objects given as input?
  - How can I allow the user to use any thread library he wishes?
- High Level Threads modeling
  - Threads using a thread API → management of threads
  - Threads as processes → management of processes
  - Threads as containers → WIP
  - Shell Script based experimentation → ???


Abstraction
Level
Failure / Recovery Events
Timers
Kernel threads API (std::thread, pthreads)
OR
Linux processes (POSIX signals, timers, fork/exec)
High Level Thread Concept
Failure/Recovery Simulation Framework API

Design Ideas

CPU Cache warming and benchmarking

- “Warm” cache can produce “good” times in benchmarks
- On independent thread failures caches are allowed to retain their “warmth level”, since other threads continue to operate on the system state
- On system wide failures, it is not realistic to retain the cache “warmth level” after a failure, since caches are cleared on real power-failures
- Something to keep in mind

Coordinator Based system

- Threads as std::threads, with STL concepts
- Workers execute experiment code
- Coordinator schedules events/timers
  - Coordinator thread / Separate thread for event handling
- Coordinator does bookkeeping of the workers (waits on his child threads)
- Cache is warm after system recovery
  - During system failure simulation: Run “cache cooling” code

Coordinator Based system

- Threads as pthreads, with POSIX Signals/Timers concepts
  - Workers execute experiment code
  - Coordinator sends signals/timers
    - Coordinator thread / Separate thread for signal handling
    - Signal handling by specific thread using pthread_kill()
    - (Signal disposition is process wide)
  - Coordinator does bookkeeping of the workers (waits on his child threads)
  - Cache is warm after system recovery
    - During system failure simulation: Run “cache cooling” code

Coordinator based system

- Threads as Linux processes, with POSIX Signals/Timers concepts
  - Worker threads cannot operate on shared address space (user cannot share global variables on his experiments)
  - Workers execute experiment code
  - Coordinator sends/handles signals/timers
  - Coordinator does bookkeeping of the workers (waits on his child processes)
  - Usage of fork()/exec() idiom
    - fork() duplicates parent process
    - exec() overwrites the state of the current process
- Pthread level signal handling issue is irrelevant since each process handles its received signals
- Cache warming is irrelevant since each process has its own virtual address space (wip convince myself about that)

Script(?) based system

- For System Wide failures only
- Coordinator as separate Linux process
- Worker Threads as std::threads of another Linux process
    - Workers execute experiment code
    - Coordinator sends signals/timers
    - Worker process handles signals
    - Coordinator has limited knowledge and control over worker threads
    - A script launches the Coordinator process
    - Coordinator can terminate and relaunch the Worker process using fork()/exec()
    - Usage of fork()/exec() idiom
        - fork() duplicates parent process
        - exec() overwrites the state of the current process
- Pthread level signal handling issue is irrelevant since we do not handle individual thread failures
- Cache warming is irrelevant since each process has its own virtual address space (wip convince myself about that)

STL and POSIX libs

- Preference of STL concepts over POSIX pthreads / signals / timers concepts
- STL features
  - Wide compiler support (GCC, Clang, MSVC) over multiple OS (Linux, Windows, Mac)
  - C++ as well as C support
- POSIX features
  - Supported in old versions of GCC

Thread failure modeling

- Thread canceling
  - Coordinator forces a thread to terminate
- Request for code execution through signals
  - Coordinator issues code execution request though a signal
    - Request for sleep, busy work loop
  - Worker stops when it reaches the signal handling part
- Cooperative termination
  - Coordinator issues stop request
  - Worker stops when it reaches the request handling part
  - User must inject request handling code in experiments
- Cooperative request for code execution
  - Coordinator issues code execution request
    - Request for sleep, busy work loop
  - Worker stops when it reaches the request handling part
  - User must inject request handling code in experiments

API Usage
Timer Example

System Failure/Recovery API Use Simplified

```
Void PrepareExperiment() {
        //set up experiment state
        Q.insert(1);
        //etc … //split workload to threads
}
Void RunExperiment(int threadId) {
        //thread code
        Q.LookUp(x); //etc …
}
Int main() {
        Coord.SetThreadsNumber(8);
        Coordinator.ConstructFailureRecoveryScenario(
        FailureFunc, RecoveryFunc, 5, 2, 3);
        //fail thread 2 after 5 secs, recovery after 2, repeat 3 times
        Coord.SetPrepare(PrepareExperiment);
        Coord.SetRunAllThreads(RunExperiment);
        …
        Coord.Prepare();
        Coord.RunAllThreads();

        //parse results
        Return 0;
}
```

```
Coordinator Coord{};
Queue Q{};
Void RecoveryCode(int id) {
        //Performs recovery
        RunExperiment(id);
}
Void FailureFunc() {
        Coord.TerminateAllThreads();
}
Void RecoveryFunc() {
        Coord.RestartAllThreads(RecoveryCode);
}
```

Thread Failure/Recovery API Use Simplified

```
Void PrepareExperiment() {
        //set up experiment state
        Q.insert(1);
        //etc … //split workload to threads
}
Void RunExperiment(int threadId) {
        //thread code
        Q.LookUp(x); //etc …
}
Int main() {
        Coord.SetThreadsNumber(8);
        Coordinator.ConstructFailureRecoveryScenario(
        FailureFunc, RecoveryFunc, 5, 2, 3);
        //fail thread 2 after 5 secs, recovery after 2, repeat 3 times
        Coord.SetPrepare(PrepareExperiment);
        Coord.SetRunAllThreads(RunExperiment);
        …
        Coord.Prepare();
        Coord.RunAllThreads();

        //parse results
        Return 0;
}
```

```
Coordinator Coord{};
Queue Q{};
Void RecoveryCode(int id) {
        //Performs recovery
        RunExperiment(id);
}
Void FailureFunc() {
        Coord.TerminateThread(2);
}
Void RecoveryFunc() {
        Coord.RestartThread(2,RecoveryCode);
}
```

Thread Failure/Recovery API Use

```
Void PrepareExperiment() {
        //set up experiment state
        Q.insert(1);
        //etc … //split workload to threads
}
Void RunExperiment(int threadId) {
        //thread code
        Q.LookUp(x); //etc …
}
Int main() {
        Coord.SetThreadsNumber(8);
        Coordinator.SetTimer("failureTimer",
        [](){FailureFunc();}, 5, 3).StartOnRun(true);
        //fail every 7 secs, repeat 3 times
        Coordinator.SetTimer("recoveryTimer",
        [](int id){RecoveryFunc(id);}, 2, 3).StartOnRun(false);
        //recover every 2 secs, repeat 3 times
        Coord.SetPrepare([](){PrepareExperiment();});
        Coord.SetRun(1, [](int id){RunExperiment(id);});
        Coord.SetRun(2, [](int id){RunExperiment(id);});
        …
        Coord.Prepare();
        Coord.RunAllThreads();

        //parse results
        Return 0;
}
```

```
Coordinator Coord{};
Queue Q{};
Void RecoveryCode(int id) {
        //Performs recovery
        RunExperiment(id);
}
Void FailureFunc() {
        Coord.TerminateThread(2);
        Coordinator.GetTimer("recoveryTimer").Start();
}
Void RecoveryFunc() {
        Coord.RestartThread(2,[](int id){RecoveryCode(id);});
}
```

User Requirements

- Uniform distribution of failures
  - Time based description of failures
- Predicate based description
- Keyboard Triggered failures
- Check “user requirements” doc

Experiment Descriptions

- Use a JSON file to describe the experiment parameters
- User code “ThreadRun”, “Recovery” functions are provided as function pointer parameters
- Check “config.json” “main.cpp” files

WIP

- Plot cache warming phenomena
  - Experiment: Recover a 1 million uint64 array from persistence to DRAM
  - Recovery performed by new thread vs process, compare runtime
  - Code authored → Debugging → Recovery Works
- Construct own timing framework
  - Convenient timing utilities from my Msc work
- Construct own plotting framework
  - Using python: seaborn library
  - Library installed → hello world done → library conceptualization phase
- Signal disposition to threads code
  - In a process address-space, I can send a signal from a master to a thread of choice and run handling code