Lock-Free Fire Dispatch Sytem

SENG490 – Directed Study

Ethan McNamara
Professor: Sean Chester
Fall 2022

INTRODUCTION

- 4th year Software Engineering student
- Previous Dr. Chester student
- Worked on fire dispatch system during co-op work term

Agenda

01 FIRE & EMERGENCY DISPATCH

02 LOCK-FREE PROGRAMMING

03 PROGRAM IMPLEMENTATION

04 LESSONS LEARNED

01
FIRE & EMERGENCY DISPATCH

WHAT IS A DISPATCH SYSTEM?

- Interface between 911 call centre and fire station
- Human operator required to assign trucks to emergency situations
- Dispatch system notifies station

MANUAL DISPATCHING

911 OPERATOR RECEIVES INFO

Caller provides event location & requirements

INFO IS PASSED TO DISPATCHER

Fire, police, EMS are dispatched seperately

DISPATCHER ASSIGNS CREWS TO EVENT

Crews are notified through radios/pagers connected to dispatch system

DISPATCH SYSTEM REQUIREMENTS

PERFORMANCE

Events are happening
in real-time

FAULT-TOLERANT

System failure may have
lethal consequences

EXTERNAL COMMUNICATION

Messages need to be
sent to radios & pagers

RESPONSIVENESS

Multiple operators may
use system concurrently

DISPATCH SIMULATOR

- Available vehicles & stations remain constant
- Timestamped event data provided at runtime
- Completely automated dispatching
- Results outputted to log

WHY USE MULTITHREADING?

INCREASED PERFORMANCE

Multiple events can be dispatched concurrently

IMPROVED RELIABILITY

Individual thread failure will not cause system failure

02

LOCK-FREE PROGRAMMING

MULTITHREADED PROGRAMMING

LOCK-FREE PROGRAMMING TOOLS

Compare and Swap/Exchange (CAS)

Atomic compare & swap if equal operation

- `std::atomic::compare_exchange_[weak/strong]`

```
atomic function CAS(int* modified, int old, int new):
    if *modified ≠ old
        return false
    *modified ← new
    return true
```

CAS LOOP
CODE EXAMPLE

CAS function usage:

1. Read data into local variable
2. Modify local variable
3. Compare local with stored and swap if equal

```
void increment(int* p):
    repeat:
        value := *p
    until CAS(p, value, value + 1)
```

LIMITATIONS OF CAS

DATA SIZE LIMITS

CAS operates on single- or double-word data

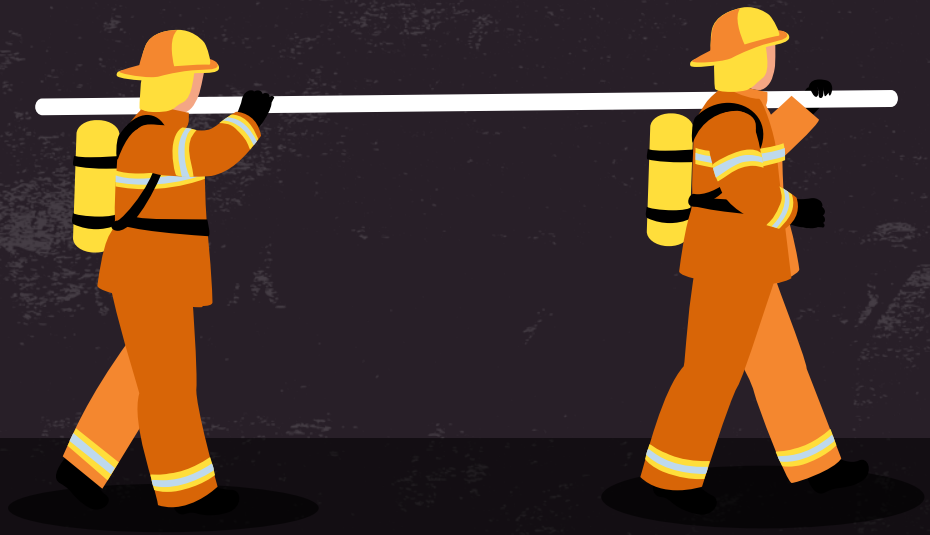
A-B-A PROBLEM

Data is reallocated in address of previously deallocated data

LINEARIZABLE HISTORY

- Manner of organizing multithreaded interactions with shared data
- Each interaction occurs at a discrete instant
- All interactions ordered sequentially
- This interaction is known as the linearization point

POINT OF LINEARIZATION



- Interaction with shared data occurs at single point
- Location in code may differ based on branching
- Requirement for proving correctness of lock-free algorithms

TREIBER STACK
LOCK-FREE EXAMPLE

```
class stack <node*> top


void stack.push(node* n):
    repeat:
        o := top
        n->next := o
    until CAS(&top, o, n)
```

```
node* stack.pop():
    repeat:
        o := top
        if o = null: return null
        n := o->next
    until CAS(&top, o, n)
    return o
```

03

PROGRAM IMPLEMENTATION

PROGRAM ARCHITECTURE

OBJECT-ORIENTED DESIGN

Vehicles, events, fire stations, etc. are individual classes

EVENT QUEUE

Events moved from pending queue to active queue at start time

GLOBAL BIT ARRAY

One bit per vehicle represents its availability

EVENTS

- Events occur at a discrete time
- Each event has different vehicular and crew requirements
- Travel from station to event is considered for vehicles

VEHICLE STATUS ENUM

- Status options:
  - Available
  - Responding
  - Returning
- Vehicle can be dispatched to event while in Available or Returning

GLOBAL BIT ARRAY

- 64-bit variable to accommodate single-word CAS
- Bit N represents availability of Vehicle N
- Modification of bit array is program's linearization point

GLOBAL BIT LIST MODIFICATION

RETURNS MODIFIED COPY OF globalBitArray

```
uint64_t BitArray::modifyBitArray(int vehicleID, bool writeTrue)
{
    uint64_t position = 1 << vehicleID;
    uint64_t copyGlobalBitArray  = globalBitArray;
    if( writeTrue )
    {
        // Write 1 to vehicle position in array
        copyGlobalBitArray |= position;
    }
    else
    {
        // Write 0 to vehicle position in array
        copyGlobalBitArray &= ~position;
    }
    return copyGlobalBitArray;
}
```

OBTAINING MODIFIED BIT LIST

modifiedBitArray **IS INITIALIZED AS COPY OF** globalBitArray

```
for (const auto & vehicle : vehicleList)
{
    if ( vehicle->getCurVehicleStatus() == VehicleStatus::Available
    || vehicle->getCurVehicleStatus() == VehicleStatus::Returning )
    {
         modifiedBitArray.setGlobalBitArray(
         modifiedBitArray.modifyBitArray(vehicle->getVehicleID(), true) );
    }
    else
    {
        return false;
    }
}
```

ATOMIC MODIFICATION

USING CAS

```
uint64_t expected = unmodifiedBitArray.getGlobalBitArray();
uint64_t desired = modifiedBitArray.getGlobalBitArray();
if( bitArray->globalBitArray.compare_exchange_weak(expected, desired) )
{
    for (const auto & vehicle : vehicleList)
    {
        vehicle->setCurVehicleStatus(VehicleStatus::Responding);
    }
    return true;
}
return false;
```

04

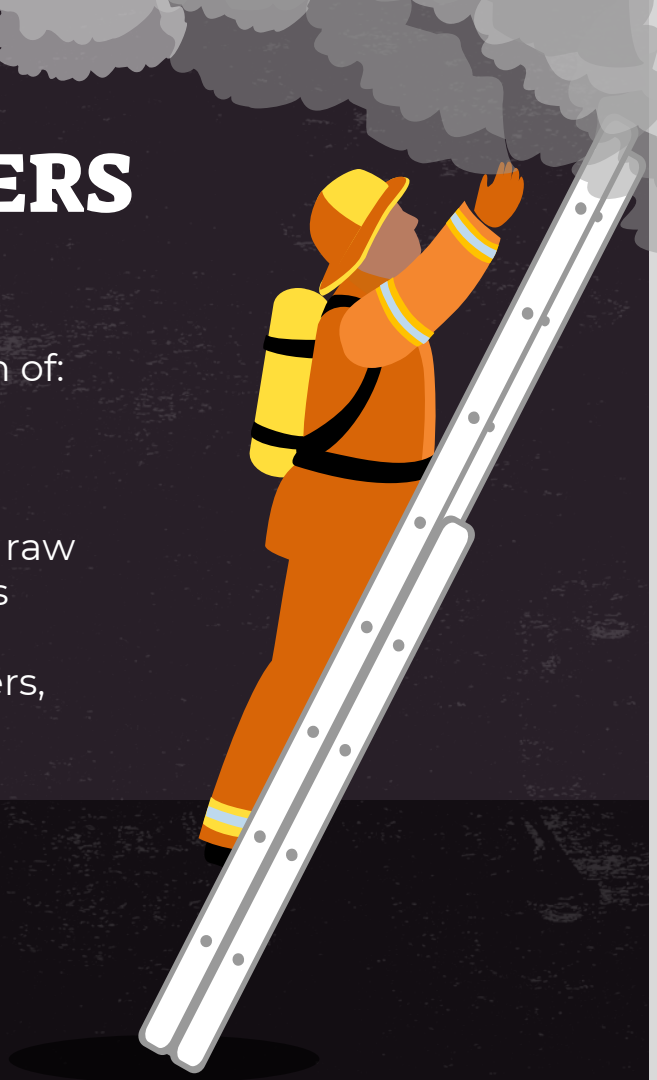
LESSONS LEARNED

DISCOVERING THE LIMITATIONS OF CAS

- Original plan was to modify statuses directly
- No atomic means of modifying disjoint memory locations exists
- CAS imposed limit on number of vehicles

C++ SMART POINTERS

- Global vector of **`unique_ptrs`** for each of:
  - Vehicles
  - Fire Stations
  - Events
- Copies of each vector are made using raw pointers to the **`unique_ptr`** addresses
- Mitigates A-B-A problem
- Initial design did not use smart pointers, refactoring was complex



THANKS!

Do you have any questions?

github.com/ethan-mcnamara
linkedin.com/in/ethan-mcnamara

Project repository:
github.com/ethan-mcnamara/lock-free-programming