Tasks

TODO Memory batcher implementation

[2025-07-08 Tue] file:~/Documents/code/findex/crate/memories/src/memory_batcher.rs

Two designs are possible:

- a client-server architecture in which a server awaits for a given number of incoming messages;
- a decentralized architecture in which the first client to fill the buffer has to process it.

The pros in favor of the client-server architecture is that clients are trivial: they simply send their request and await for an answer. Also, the sensitive logic is centralized which makes verifying it easier. The cons is that a concurrent task has to be spawned in order for client calls to be answered.

The pros in favor of the decentralized architecture is that no additional concurrent task is required. The cons is that logic is distributed and thus hard to verify.

A middle-ground can be taken in adopting a version of the decentralized architecture in which all concurrency-related code is centralized in a few function which all others call. The following operations implement the logic related to the client operations. Strikingly, `batch_read` and `guarded_write` are almost identical.

Memory batcher implementation

Note that in order for this implementation to work, channels must not block the sender, otherwise the process calling `manage_buffer` would block trying to send data to itself.

```
struct MemoryBatcher(Arc<Mutex<FixedSizeBuffer>>);

impl MemoryBatcher {
    async fn unsuscribe(&self) -> Result<(), Error> {
        let res = { self.buffer.lock().unwrap().shrink_capacity() };
        if let Some(ops) = res {
            self.manage(ops).await?;
        }
        Ok(())
    }
```

```
    async fn apply(&self, op: MemoryOperation) -> Result<MemoryResult, Error> {
        let (snd, rcv) = channel();
        let res = { self.buffer.lock().unwrap().push((snd, op)) };
        if let Some(ops) = res {
            self.manage(ops).await?;
        }
        rcv.recv().await
    }

    async fn batch_read(&self, addresses: Vec<Address>) -> Result<Vec<Word>, Error> {
        let res = self.apply(MemoryOperation::Read(addresses)).await?;

        if let MemoryResult::Read(words) = res {
            Ok(words)
        } else {
            Err(Error::WrongResultType)
        }
    }

    async fn guarded_write(
        &self,
        guard: (Address, Option<Word>),
        bindings: Vec<(Address, Word)>,
    ) -> Result<Vec<Word>, Error> {
        let res = self.apply(MemoryOperation::Write(guard, bindings)).await?;

        if let MemoryResult::Write(word) = res {
            Ok(word)
        } else {
            Err(Error::WrongResultType)
        }
    }
}
```

The `manage` operation is implemented (almost) identically to the `manage_buffer` from `tbz/batcher`.

Buffer implementation

The buffer operations can trivially be implemented:

```
struct Buffer<T> {
    capacity: usize,
    data: Vec<T>,
}
```

```rust
impl<T> Buffer<T> {
    fn new(capacity: usize) -> Self {
        let data = Vec::with_capacity(capacity);
        Self { capacity, data }
    }
}

impl<T: Clone> Buffer<T> {
    /* Correctness invariant:
     *
     * 0 <= self.data.len() < self.capacity
     *
     * or
     *
     * 0 = self.data.len() = self.capacity
     *
     *  */

    fn shink_capacity(&mut self) -> Result<Option<Vec<T>>, BufferError> {
        if self.capacity == 0 {
            return Err(BufferError::Underflow);
        }

        self.capacity -= 1;

        if 0 < self.capacity && self.capacity == self.data.len() {
            Ok(Some(self.data.drain(0..self.capacity).cloned().collect()))
        } else {
            Ok(None)
        }
    }

    fn push(&mut self, t: T) -> Option<Vec<T>> {
        if self.capacity == 0 {
            return Err(BufferError::Overflow);
        }

        self.data.push(t);

        if self.capacity == self.data.len() {
            Ok(Some(self.data.drain(0..self.capacity).cloned().collect()))
        } else {
            Ok(None)
        }
    }
```

```
}
```

Since all the logic is centralized, it is possible to locally verify the correctness invariant. Since all data is returned from the operation, the lock does not need to be hold through the asynchronous call to `manage` anymore.