1 Readings

The readings for today were:

- *Eventually Consistent Transactions*, by Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen and Mooly Sagiv, available at
`http://research.microsoft.com/pubs/155638/msr-tr-2011-117.pdf`
- *Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects*, by Maurice Herlihy and Eric Kosniken, available at
`http://www.cl.cam.ac.uk/~ejk39/papers/boosting-ppopp08.pdf`
- *Coarse-Grained Transactions*, by Eric Koskinen, Matthew Parkinson and Maurice Herlihy, available at
`http://www.cl.cam.ac.uk/~ejk39/papers/cgt.pdf`
Most of the discussion was based on this last paper.

2 Skew heaps

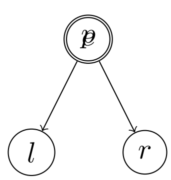
2.1 Definitions

Let us start with an example: parallelizing insertions in skew heaps.

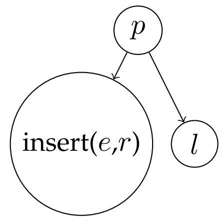
A skew heap (`http://en.wikipedia.org/wiki/Skew_heap`) is a heap data structure implemented as a binary tree. It is defined recursively as:

- A heap with only one element is a skew heap;
- The result of skew merging two skew heaps is also a skew heap.

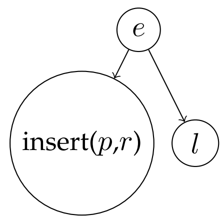
Let us just describe how to insert one element e into a skew heap h with root p, whose subtrees are l on the left and r on the right; the general merge of two heaps can be found on Wikipedia:



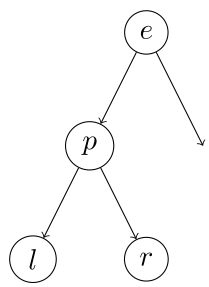
- inserting e into an empty heap just results in the heap with just that element;
- if $e > p$, the resulting heap has root p, its left subtree is the result of recursively inserting e in r, and its right subtree is l. Note how we swapped r and l in the process;



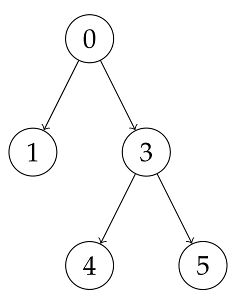
- if $e < p$, the resulting heap has e as a root, its left subtree is the result of inserting p in r and its right subtree is l.



Remark: What we said in class does not seem to match the Wikipedia article. For wikipedia, the resulting heap has root e, with the original heap h as its left subtree and an empty right subtree.



Now suppose we have the following heap and two concurrent threads, A adding 2 to it, and B adding 6 to it. Several solutions are possible.

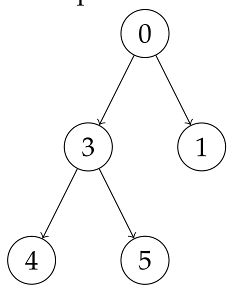
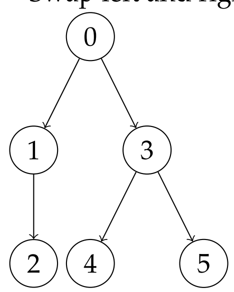
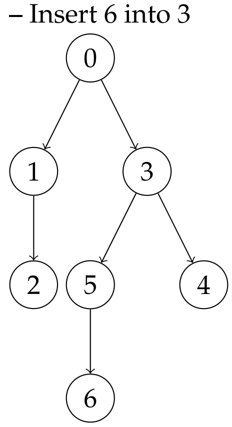


2.2 Global lock

The first solution is to use a global lock on the heap. This is not so nice because it is slow: you have to wait for one insertion to complete before the next one can start.

2.3 Local locks

The second solution is to use a fine grain locking procedure: use one lock per node, and acquire and release the locks as we traverse the tree. Here is an example of an execution using such a procedure:

Thread A	Thread B
– Acquire lock for node 0	
– Swap children of 0	
	
– Release lock for node 0	
– Acquire lock for node 1	– Acquire lock for node 0
– Insert 2 into 1	– Swap left and right children of 0
	
	– Insert 6 into 3
	

This performs relatively well, but it is hard to write and get right.

2.4 Transactions

We could also use transactions, but in the previous examples, here is what would happen:

Thread A	Thread B
– Read 0	– Read 0
– Write $0.\{L, R\}$	– Write $0.\{L, R\}$
$\vdots$	$\vdots$

We will get a conflict when committing, and one of the threads will have to be rolled back. It hinders concurrency, even though it is efficient for the programmer.

Here is one more example: imagine trying to insert a few single-digit numbers and 6000 into a huge heap. The insert of a single-digit number has a short log, and will always try to commit first. The insert of 6000, on the other hand, will always get rolled back. If trying to parallelize it with several insertions of single-digit numbers, it will only actually happen at the very end.

3 Coarse-Grained Transaction

We now focus on Maurice Herlihy and Eric Koskinen's work, and especially their POPL'10 paper *Coarse-Grained Transactions*.

The basic idea is to have a transaction mechanism that is convenient to programmers but does not have the big conflicts that normal transactions generate.

Example 1.
```
global SkipList Set;
T1 : atomic {
  Set.add(5);
  Set.remove(3);
}
T2 : atomic {
  if(Set.contains(6)){
    Set.remove(7);
  }
}
```
In this example using a fine-grained synchronization technique would produce a conflict, because adding 5 and removing 7 will read or write the same memory locations. However at a coarse-grained level, these operations commute with each other as soon as their arguments are distinct.

We will consider two execution semantics:

- Pessimistic Execution Semantics (section 3.1 of the paper) prevent conflicts by checking if there will be a conflict before doing anything. They are called pessimistic because they act like there will always be a conflict.
- Optimistic Execution Semantics (section 3.2 of the paper) detect conflicts afterwards: they copy the initial state, and roll back if there is a conflict when committing. They are called optimistic because they act hoping that there will not be a conflict.

3.1 Syntax

The language is given by:

$$\begin{array}{rcl} c & ::= & c; c \mid e := e \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \\ s & ::= & s; s \mid c \mid \texttt{beg};\ t;\ \texttt{end} \\ t & ::= & t; t \mid c \mid o.m \end{array}$$

$\texttt{beg};\ t;\ \texttt{end}$ is the syntax for a transaction. $o.m$ represents a method call m on a shared object o. Note that the syntax purposely makes nested transactions impossible.

The semantics is a labeled transition system, with transitions

$$T, \sigma_{\mathsf{sh}} \xrightarrow{\alpha}_A T', \sigma'_{\mathsf{sh}}$$

where T is a set of transactions t, σ_{sh} is a global state, i.e., a finite map from object ids to objects. $\alpha \in \bot \cup (\mathbb{N} \times (o.m \cup \{\texttt{beg}, \texttt{end}, \texttt{cmt}\}))$ is a label. `beg` is short for begin, and `cmt` is short for commit. A just means atomic.

Each transaction is a tuple $\langle \tau, p \rangle$, where τ is a transaction identifier and p is the program text for the transaction.

The abridged atomic semantics is given by:

$$\frac{}{(\langle \tau, c; c' \rangle \cup T), \sigma_{\mathsf{sh}} \xrightarrow{\bot}_A (\langle \tau, c' \rangle \cup T), [\![c]\!]\sigma_{\mathsf{sh}}}$$

$$\frac{\alpha = (\tau, \texttt{beg}) \cdot \alpha' \cdot (\tau, \texttt{end}) \cdot (\tau, \texttt{cmt}) \quad \tau \text{ fresh} \quad (\langle \tau, t \rangle \cup T), \sigma_{\mathsf{sh}} \xrightarrow{\alpha'}{}^{*} (\langle \tau, \texttt{skip} \rangle \cup T), \sigma'_{\mathsf{sh}}}{(\langle \bot, \texttt{beg};\, t;\ \texttt{end} \rangle \cup T), \sigma_{\mathsf{sh}} \xrightarrow{\alpha}_A (\langle \bot, \texttt{skip} \rangle \cup T), \sigma'_{\mathsf{sh}}}$$

3.2 Pessimistic semantics (section 3.1)

A transaction is now a tuple $\langle \tau, s, M, \sigma_\tau \rangle$, where M is a sequence of object methods, and σ_τ is a local state. All the arrows are indexed with a P, for pessimistic.

$$\frac{}{(\langle \tau, s \in \{:=, \texttt{if}, \ldots\}; s', M, \sigma_\tau \rangle \cup T), \sigma_{\mathsf{sh}} \xrightarrow{\bot}_P (\langle \tau, s', M, [\![s]\!]\sigma_\tau \rangle \cup T), \sigma_{\mathsf{sh}}}$$

$$\frac{\tau \text{ fresh}}{(\langle \bot, \texttt{beg}; s, M, \sigma_\tau \rangle \cup T), \sigma_{\mathsf{sh}} \xrightarrow{\tau, \mathsf{beg}}_P (\langle \tau, s, [\,], \sigma_\tau \rangle \cup T), \sigma_{\mathsf{sh}}}$$

$$\frac{\{o.m\} \lhd \mathsf{meths}(T)}{(\langle \tau, x := o.m; s, M, \sigma_\tau \rangle \cup T), \sigma_{\mathsf{sh}} \xrightarrow{\tau, o.m}_P (\langle \tau, s, M :: (\text{“}x := o.m; s\text{”}, \sigma_\tau, \text{“}o.m\text{”}), \sigma_\tau[x \mapsto rv([\![o]\!]\sigma_{\mathsf{sh}}.m)] \rangle \cup T), \sigma_{\mathsf{sh}}[o \mapsto [\![o]\!]\sigma_{\mathsf{sh}}.m]}$$

The difficult part is $o.m$ being invoked has some global effect: each invokation of o returns a new object and a return value (accessed through rv). Under transactional semantics, there is a guarantee to the programmer that the actual execution will be equivalent to some serial execution. To get this we need to know that it was ok to call $o.m$ at that point. This is the reason behind the introduction of the mover concept. We write $\{o.m\} \lhd \mathsf{meths}(T)$ for “$o.m$ is a left mover of T”.

Definition 1. $o.m \lhd p.n$ if and only if

$$\{\sigma'' \mid \exists \sigma'. \sigma \xrightarrow{p.n} \sigma' \wedge \sigma' \xrightarrow{o.m} \sigma''\} \subseteq \{\sigma'' \mid \exists \sigma'. \sigma \xrightarrow{o.m} \sigma' \wedge \sigma' \xrightarrow{p.n} \sigma''\}$$

i.e., if the set of states obtained by running $p.n$ followed by $o.m$ is included in the set of states obtained by running $o.m$ followed by $p.n$ (left commutativity).

One problem of this semantics is that we can get deadlocks with two transactions sending messages that are not inverse of each other. A solution is running an $o.m$ that has an inverse that can be rolled back, and then make some progress from there.

3.3 Optimistic semantics (section 3.2)

A transaction is now a tuple $\langle \tau, s, \sigma_\tau, \overleftarrow{\sigma_\tau}, \ell_\tau \rangle$, where ℓ_τ is a log, a list of messages that have been sent already. All the arrows are indexed with an O, for optimistic.

$$\frac{\tau \text{ fresh}}{(\langle \bot, \mathsf{beg}; s, \sigma_\tau, \overleftarrow{\sigma_\tau}, [\,] \rangle \cup T), \sigma_{\mathsf{sh}}, \ell_{\mathsf{sh}} \xrightarrow{(\tau,\mathsf{beg})}_O (\langle \tau, s, \mathsf{snap}(\sigma_\tau, \sigma_{\mathsf{sh}}), \sigma_\tau[\mathsf{stmt} \mapsto \text{“}\mathsf{beg}; s\text{”}], [\,] \rangle \cup T), \sigma_{\mathsf{sh}}, \ell_{\mathsf{sh}}}$$

Message call:

$$\frac{}{(\langle \tau, x := o.m; s, \sigma_\tau, \overleftarrow{\sigma_\tau}, \ell_\tau \rangle \cup T), \sigma_{\mathsf{sh}}, \ell_{\mathsf{sh}} \xrightarrow{(\tau,o.m)}_O (\langle \tau, s, \sigma_\tau[o \mapsto (\llbracket o \rrbracket \sigma_{\mathsf{sh}}).m, x \mapsto rv((\llbracket o \rrbracket \sigma_{\mathsf{sh}}).m)], \overleftarrow{\sigma_\tau}, \ell_\tau :: (\text{“}o.m\text{”}) \rangle \cup T), \sigma_{\mathsf{sh}}, \ell_{\mathsf{sh}}}$$

$$\frac{\forall (\tau^{\mathsf{cmt}}, \ell_{\tau'}) \in \ell_{\mathsf{sh}}.\tau^{\mathsf{cmt}} > \tau \Rightarrow \ell_\tau \rhd \ell_{\tau'}}{(\langle \tau, \mathsf{end}; s, \sigma_\tau, \overleftarrow{\sigma_\tau}, \ell_\tau \rangle \cup T), \sigma_{\mathsf{sh}}, \ell_{\mathsf{sh}} \xrightarrow{(\tau,\mathsf{cmt})}_O (\langle \tau, s, \mathsf{zap}(\sigma_\tau), \mathsf{zap}(\overleftarrow{\sigma_\tau}), [\,] \rangle \cup T), \mathsf{merge}(\sigma_{\mathsf{sh}}, \ell_\tau), \ell_{\mathsf{sh}} :: (\mathsf{fresh}(\tau^{\mathsf{cmt}}), \ell_\tau)}$$

where the merge operation goes through the log where it finds operations that it applies in order to the global state. In the last rule, the condition $\forall (\tau^{\mathsf{cmt}}, \ell_{\tau'}) \in \ell_{\mathsf{sh}}.\tau^{\mathsf{cmt}} > \tau \Rightarrow \ell_\tau \rhd \ell_{\tau'}$ makes sure the end operation is sensible.

3.4 Conclusion

Section 5 of the paper develops the theory of trace semantics of those programs, develops serial execution and shows both semantics are equivalent to some serialized execution.