By Joshua Cantrell
jjc@cory.berkeley.edu

Concurrency Diagrams

Introduction:

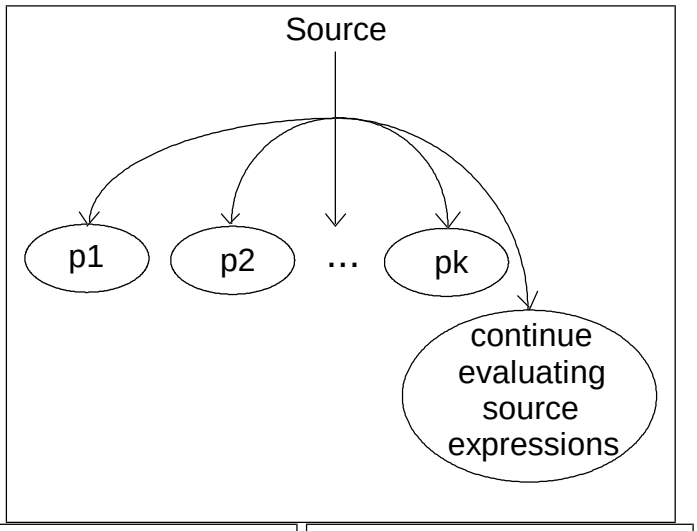
Concurrency is widely used in many new applications because of the usefulness of parallel processing. Examples of concurrency can be found in preemptive multitasking systems, multiple processor computers, and special networks of computers that work together to run a single program (the NOW project here at UC Berkeley does this). The major obstacle encountered when designing concurrent processes is synchronizing them so that they use the same data without any conflicts.

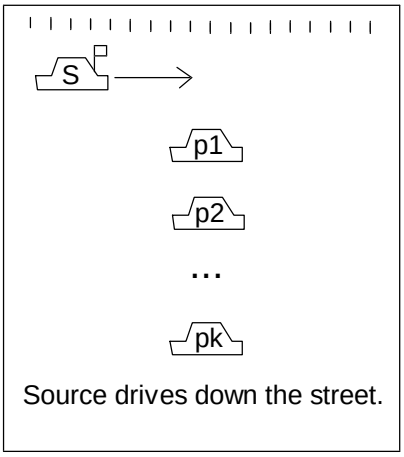
Parallel Execution:

Our book, SICP (*Structure and Interpretation of Computer Programs*), goes over a procedure called `parallel-execute` which takes as actual parameters any number of procedures without formal parameters and evaluates them simultaneously along with the rest of the expressions in the body that follow the parallel-execute procedure. To draw many procedures being evaluated simultaneously, bubbles can be drawn around the procedures being evaluated and they can be connected by an arrow flowing into each of them from the parallel-execute call. The procedure prototype is:

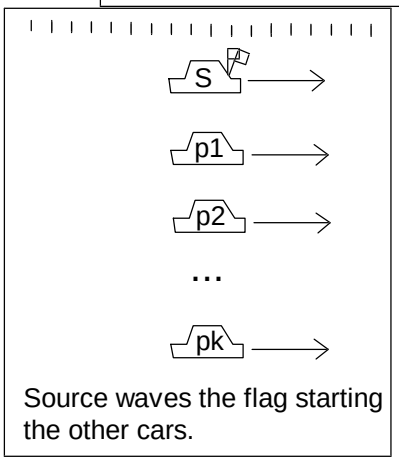
`(parallel-execute <p1> <p2> ... <pk>)`

One way to visualize how `parallel-execute` works is by thinking of the main executing body as a car driving down the road. The `parallel-execute` procedure would be a flag to the other procedures that says “go” (or maybe a gun used at a race). When the car gets to the procedure, the flag is waved and the other cars start moving and evaluating their bodies. They all stop when their bodies have finally been evaluated independent of each other.

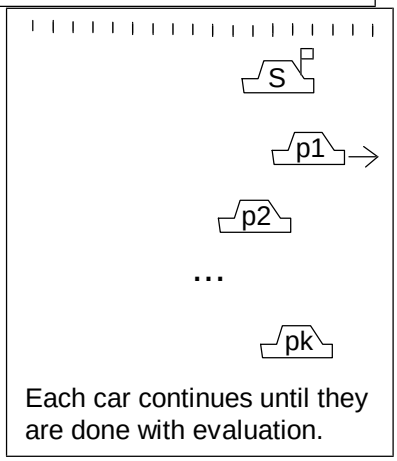




Source drives down the street.



Source waves the flag starting the other cars.



Each car continues until they are done with evaluation.

By Joshua Cantrell
jjc@cory.berkeley.edu

Problems With Concurrent Evaluation:

For procedures that aren't functional[1], concurrent evaluation could cause some strange side-effects. The most notable are those that involve mutators and accessing memory. If multiple concurrently evaluated procedures use and mutate shared data, there is a possibility of changing the data being used in the middle of a computation. An example is given below:

```
(let ((a 50))
  (let ((proc1 (lambda () (set! a 25)))
        (proc2 (lambda () (display (+ a a)))))
    (parallel-execute proc1
                      proc2)))
```

What are all the numbers that might be displayed? There are three possibilities, 50, 75, and 100. How can we figure these out through inspection? First we need to locate all of the mutators in the given procedures. The mutators are the only expressions that can create the different outcomes. Next we must locate all elements related to the various mutators. These make up the different places that can be changed during evaluation. If multiple mutators appear in a procedure, we must take into account that all of the expressions in a procedure work in serial, so a *set!* found later in the procedure always occurs after the previous *set!*.

For an example, I'll make a slightly more complex example:

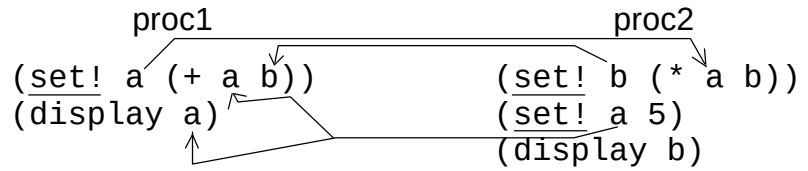
```
(let ((a 4)
      (b 3))
  (let ((proc1 (lambda () (set! a (+ a b))
                          (display a))
        (proc2 (lambda () (set! b (* a b))
                          (set! a 5)
                          (display b))))
    (parallel-execute proc1
                      proc2)))
```

proc1	proc2
(set! a (+ a b))	(set! b (* a b))
(display a)	(set! a 5)
	(display b)

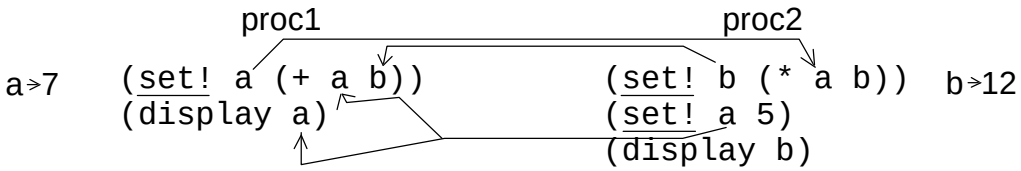
Step1:
Layout all the sets of expressions being evaluated in parallel, and find the mutators in those sets.

[1] Functional procedures are those that always return the same value given the same actual parameters and result in no mutation.

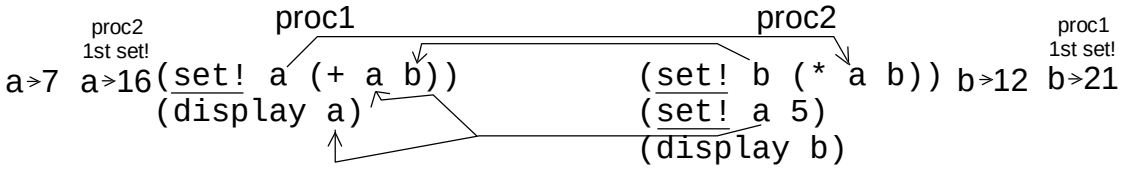
By Joshua Cantrell
jjc@cory.berkeley.edu



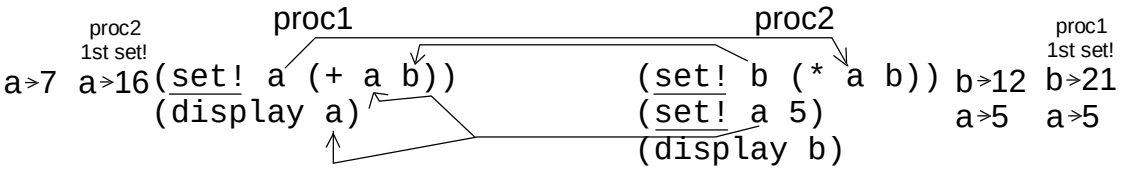
Step2:
Find the symbols or data related to the mutators that can be changed in the middle of evaluation.



Step3:
Start calculating what the possible values are for each procedure by finding all the possible values that could be calculated at the first 'set!'s.

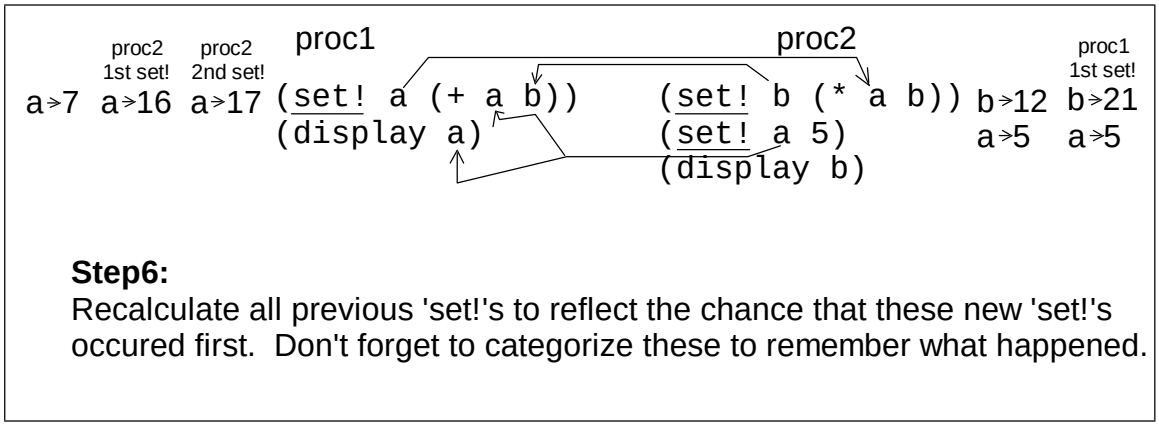


Step4:
Recalculate the possible values of the 'set!'s by including the values the other 'set!' could produce. Remember to make a note of what 'set!'s have passed to give you those values.

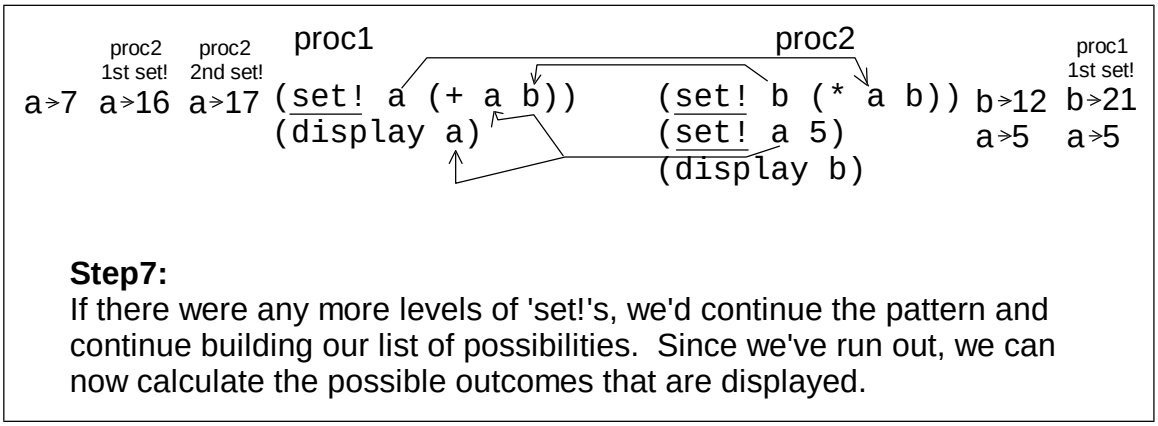


Step5:
Now move on to the next 'set!'s in each expression and write down all of their possibilites. Don't forget to also categorize them under their correct previous 'set!'s.

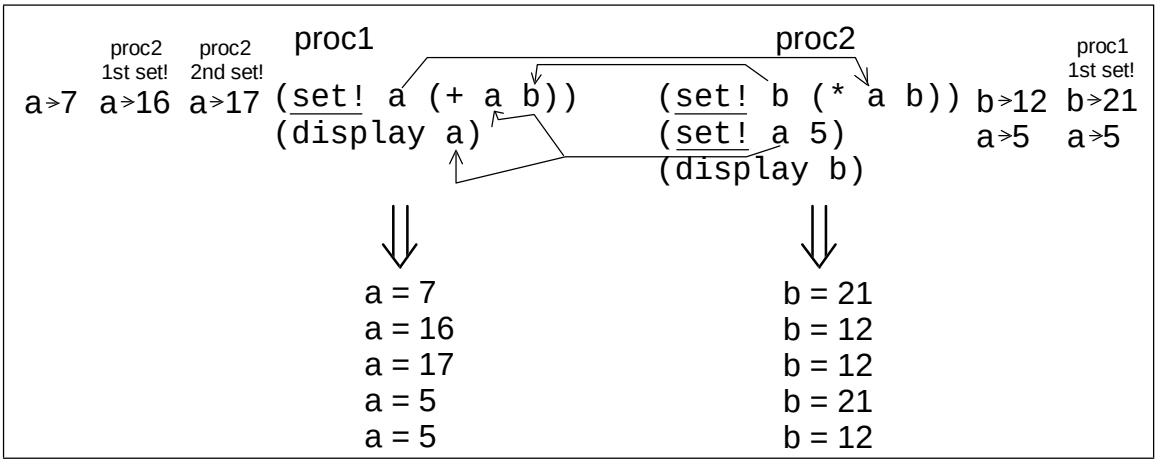
By Joshua Cantrell
jjc@cory.berkeley.edu



Step6:
Recalculate all previous 'set!'s to reflect the chance that these new 'set!'s occured first. Don't forget to categorize these to remember what happened.



Step7:
If there were any more levels of 'set!'s, we'd continue the pattern and continue building our list of possibilities. Since we've run out, we can now calculate the possible outcomes that are displayed.

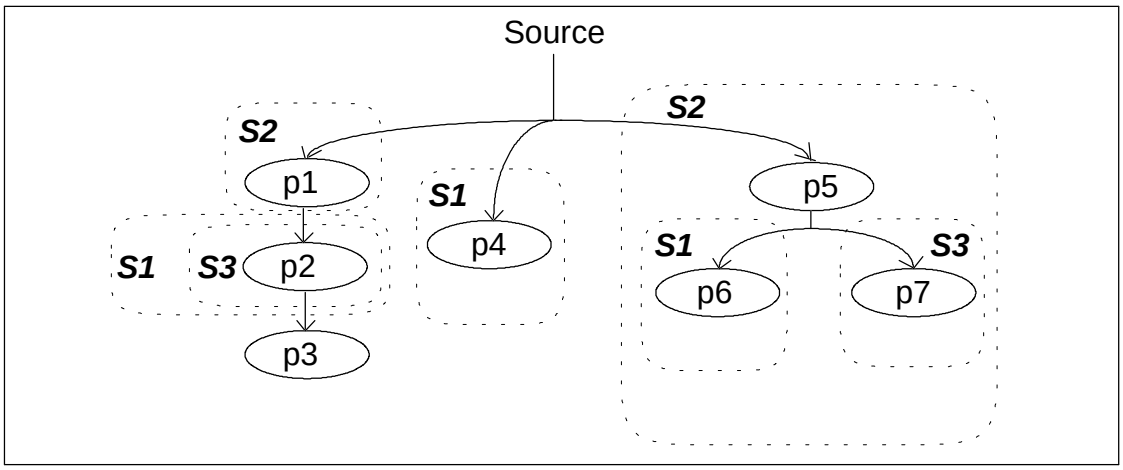


By Joshua Cantrell
jjc@cory.berkeley.edu

Serializers:

If we were to give each procedure to another computer or microprocessor to be operated on, we can see our process could finish faster than if we only evaluated them one at a time. The problem is that they might all share the same data and we can't have them blindly reading and changing the data simultaneously, or we'll get random results all dependent on who did what first (or possibly last). This need to protect data from being tampered with indiscriminately causes the creation of serializers which keep certain procedures from being executed simultaneously.

SICP gives us the constructor `make-serializer` to create serializers which can be used on procedures to ensure they aren't executed simultaneously. The serializers made by `make-serializer` are evaluated on procedures making them serialized by the particular serializer. Procedures serialized by the same serializer cannot be evaluated simultaneously, but those serialized by different serializers can be evaluated simultaneously. Whenever we serialize a procedure, we should draw a box around it and label it with a serializer. Note that a procedure can be serialized by multiple serializers.



The above picture can be made with the following expressions using SICP's concurrency procedures (given that all the procedures are defined and have no parameters):

```
(define S1 (make-serializer))
(define S2 (make-serializer))
(define S3 (make-serializer))
(parallel-execute
    (lambda () ((s2 p1))
               ((s1 (s3 p2)))
               (p3))
    (s1 p4)
    (s2 (lambda () (p5)
                   (parallel-execute
                       (s1 p6)
                       (s3 p7)) )) )
```

Remembering that anything not sharing the same serializer can be computed simultaneously, we can see that procedures 5, 6, and 7 cannot be evaluated at the same time as 1. We can also see that procedure 2, 4, and 6 cannot be evaluated simultaneously. Note that procedure 3 can be evaluated simultaneously with any of the procedures.

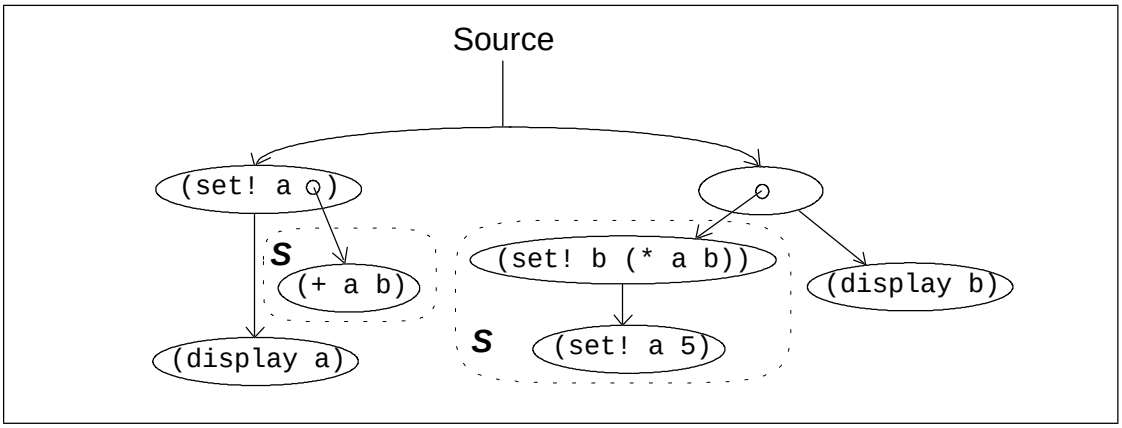
By Joshua Cantrell
jjc@cory.berkeley.edu

Serializers and Evaluation

In the step-by-step case shown above, there were no serialized procedures, so what if it did contain serialized procedures? How would that change the number of possibilities? Let's rewrite the expression and serialize portions of it:

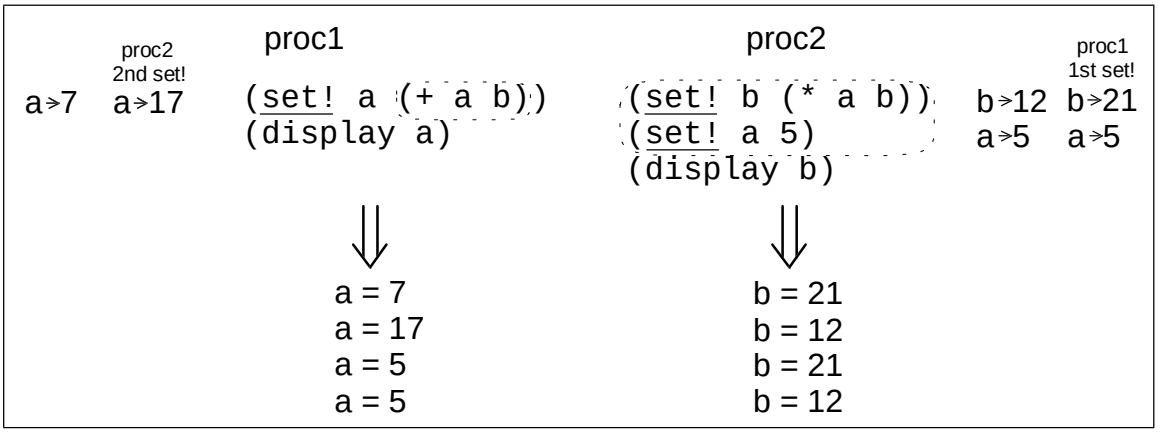
```
(let ((a 4)
      (b 3)
      (s (make-serializer)))
  (let ((proc1 (lambda () (set! a ((s (lambda () (+ a b)))))
                          (display a))
       (proc2 (lambda () ((s (lambda () (set! b (* a b))
                                        (set! a 5))))
                          (display b))))
    (parallel-execute proc1
                      proc2)))
```

What might this look like if we drew a diagram?



The only difference between the solution for this expression and the previous step-by-step solution is that parts of the procedures have been serialized. It should be noted that the serialized expressions cannot be evaluated simultaneously, so the number of possible results goes down.

I leave the solution of this problem for you to do in your spare time. The end result should look something like:

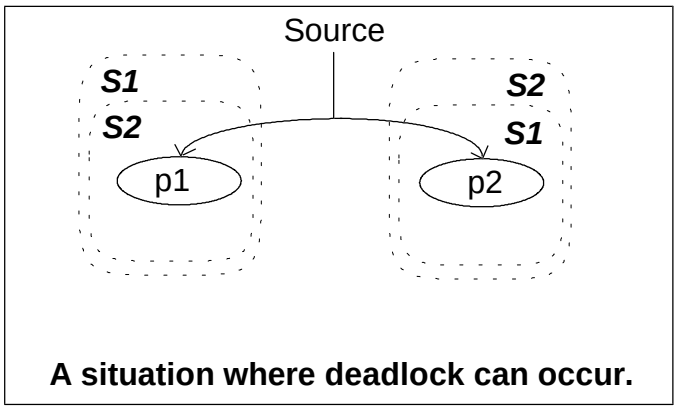


By Joshua Cantrell
jjc@cory.berkeley.edu

Deadlock:

There is one problem that can occur with serialization. Since it takes time to serialize something, it is possible that one procedure locks up one serializer and then finds its second serializer already locked. Let's say in this case, that a second serializer had just locked up that second serializer and its second serializer was the other procedure's first, so it locks up. Now we have a problem called *deadlock*.

We are saved from deadlock by various methods of avoiding it or backing out of the sticky situation. The book describes how procedures can be given priority over each other, and the procedure with the lower priority would back out so that the higher priority procedure could continue. Another way to avoid this predicament would be to serialize any possible places of deadlock. Since the technique used to avoid deadlock depends on the situation, you should be ready to choose from a variety of solutions rather than always using a single solution.



A situation where deadlock can occur.

The Essence of a Serializer:

At its lowest level, a serializer is something that checks a flag in memory to see if evaluation can continue. If the flag is set so that evaluation is stalled, it waits until the flag says evaluation can continue, and then sets the flag. Setting the flag stops other processes that use the same serializer from evaluating until it finishes and resets the flag. In some programming circles, this flag is called a *semaphore*. The most important feature of checking and setting the flag is to insure that a process cannot check the flag if it going to be set by another process. This means that these two operations (checking and setting) must happen together without any interruption of concurrent processes. The operating system, computer architecture, or interpreter normally has some mechanism to do both without interruption, so when making your own serializer, you should look for the provided semaphore support.