Concurrency

Dr. Michael Marsh
Department of Computer Science
University of Maryland

This set includes a number of games that can be incorporated into lectures or used as study aids. The purpose of these card games is to help students see the mechanics of concurrency in a simplified model, without the baggage of a full programming framework and memory model. As such, we focus less on a formal programming language and architecture, instead employing *pseudocode* and *pseudoassembly*; the former ties concepts to familiar programming methods, while the latter highlights the fact that, at the processor level, a single line of code might translate to several CPU instructions.

We divide the games into a number of sections, dealing with specific topics. Most courses covering concurrency would begin with the first section, but after that some of the sections might be covered in a different order than we have here, or omitted entirely. The section order here corresponds to the order of topics covered in the University of Maryland course CMSC 433: Programming Language Technologies and Paradigms.

Each game should take 5 to 10 minutes in a classroom setting. Where appropriate, we provide variations that might take longer for use in studying. The games are designed for 4 to 6 players.

In addition to the contents of the game set, you will need enough paper and writing instruments for all players, as well as for the shared machine state.

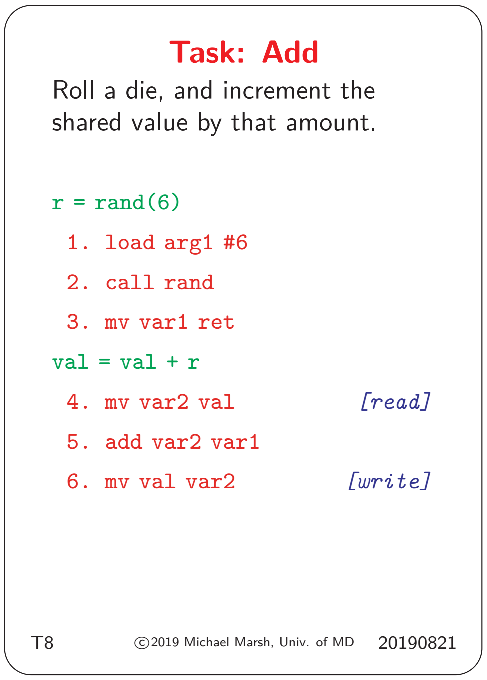
1 General Concurrency

These games cover topics like data races, synchronization, and starting and waiting for threads.

1.1 Introducing the Game Set

Each set includes a deck of cards and six 6-sided dice. The deck includes *thread* cards and *shared* cards, which are indicated on the backs of the cards, as well as by a card number in the lower-left corner. Each player should have a complete set of thread cards, though not every game will use all of the thread cards. The shared cards represent machine state, limited resources, and things like data to pass between threads or processes.

Thread cards represent *tasks* for threads to perform. Here is an example of a task:



Task: Add

Roll a die, and increment the shared value by that amount.

```
r = rand(6)
 1. load arg1 #6
 2. call rand
 3. mv var1 ret
val = val + r
 4. mv var2 val           [read]
 5. add var2 var1
 6. mv val var2          [write]
```

T8 ©2019 Michael Marsh, Univ. of MD 20190821

This shows the general layout of a task. At the top, you see that this task performs an add. Beneath that, we have a high-level description of the task. In this case, we're just going to add a random number to a shared value.

In green, we have the pseudocode for this task, which in this case is two statemnts:

1. Choose a random number.
2. Add this number to the shared value `val`.

The variable `val` will always refer to the single shared value in main memory.

In red, we have the pseudoassembly for the task, numbered consecutively from the first line of pseudocode. Each assembly instruction takes one step of the CPU, unless it is a sleep or wait. The format of an instruction is

operation arg1 . . .

For an operation like `add`, the format is

`add` *dest rval*

This replaces the variable referred to by *dest* with the sum of the original value of *dest* and *rval*.

Note that two of the operations have addition text to the right, in blue. That's because these operations involve *events* that impact shared state. These will become important in games that involve *event traces*, but for now we can ignore these.

The dice serve two purposes: they add randomness, and they record the status of the current task. Some tasks will have a prompt to roll a die to generate a random number. We will also use die rolls to select threads to run, as a *random scheduler*. The random scheduler will also use a die roll to determine the number of operations that a thread will run before yielding to the scheduler. Each thread has a *thread number* card, which the player should keep in front of them. If the random scheduler die roll selects a thread number that is not present, simply roll it again.

The other use of dice is to record where in a task a thread currently is. This is a convienient way to keep track of where you are when your thread is interrupted for another thread. The easiest way to do this is to put the number of the next operation to run facing up and sitting on the appropriate task card. A few of the tasks have more than six operations, so some other mechanism will be necessary for these, such as placement of the die on the card.

1.2 Unsynchronized Access to a Shared Variable

This game requires each thread to use the following tasks:

T15 Sleep

T13 Increment

Threads will also have a thread number card (T1–T6), for scheduling.

Each thread will also need some way to record its view of the shared value, as well as some way to record the current shared value in main memory. Paper and pencil work well for this.

The shared value begins at 0. Each thread begins ready to execute the first instruction of the Sleep task. Scheduling is performed randomly:

1. One player rolls a die to select the next thread to run.
2. The selected thread rolls a die to determine the number of instructions to run before the next round of scheduling.
3. The running thread then executes up to this number of instructions, stopping when it completes both tasks or reaches the step `call sleep`.

Since each thread should increment the shared value by 1, when all threads have completed, the shared value should be equal to the number of players. Is this the result you obtained, or did you end up with a smaller number?

1.3 Synchronized Access to a Shared Variable

This game requires each thread to use the following tasks:

T15	Sleep
T17	Lock A
T13	Increment
T19	Unlock A

Threads will also have a thread number card (T1–T6), for scheduling. You will also use a single Lock/Semaphore A (S1) card from the set of shared resource cards.

This is very similar to the previous game, with the addition of the locking cards. A thread that is sleeping or waiting to acquire the lock will yield back to the scheduler immediately.

When completed, you should always obtain the number of players as the shared value. Did this game take longer to complete than the previous one?

1.4 Deadlock

This game builds on the previous by adding a second lock. Half of the players (rounded up) have the following list of tasks:

T15	Sleep
T17	Lock A
T18	Lock B
T13	Increment
T20	Unlock B
T19	Unlock A

The other half (rounded down) have:

T15	Sleep
T18	Lock B
T17	Lock A
T13	Increment
T19	Unlock A
T20	Unlock B

Is every thread able to complete? If you have enough time, try playing through this game multiple times.

2 Task Pools

I think the existing cards support this

2.1 Fork/Join

2.2 Work Queues

something with work-stealing

one player is the scheduler

deals random tasks to threads' deques

tasks just used to count computation times – no sleeping/blocking/waiting

steal from other threads if deque empty

3 Streams

this seems like it would require a different type of card

I think we need:

- numbers (these can be incorporated into other games)
- generator operations
- intermediate operations
- terminal operations

The operations would have arrow indicating whether something must come before or after them, like the queue items.

Since this would not be threaded, we can use the thread numbers for any modular operations.

4 Messaging

akka and MPI

what new cards do we need for this?

stream cards might work for this, as well as for map-reduce

5 Nonblocking Algorithms

this will need some new cards

6 Acknowledgements

An early prototype for this set was playtested in my CMSC 433 section at the end of the Spring 2019 semester. The following students provided invaluable feedback and suggestions:

- Ben Carlisle
- Michael Dressner
- Brent Hazman
- Sean Malone
- Marco Mocchetti
- Kevin Pagkatipunan
- Tristan Perry
- Trulee Riley
- Fernando Rodriguez
- Jeff Sweeny
- William Sentosa Tio
- Minh Truong

I would also like to thank the teaching staff of CMSC 433, who also provided great feedback:

- Dr. Anwar Mamat, with whom I co-taught the course
- Rimon Melamed
- Mehdi Moaddeb
- Alex Proctor
- Nao Rho
- Julian Vanecek