E Fundamentals… with Donuts

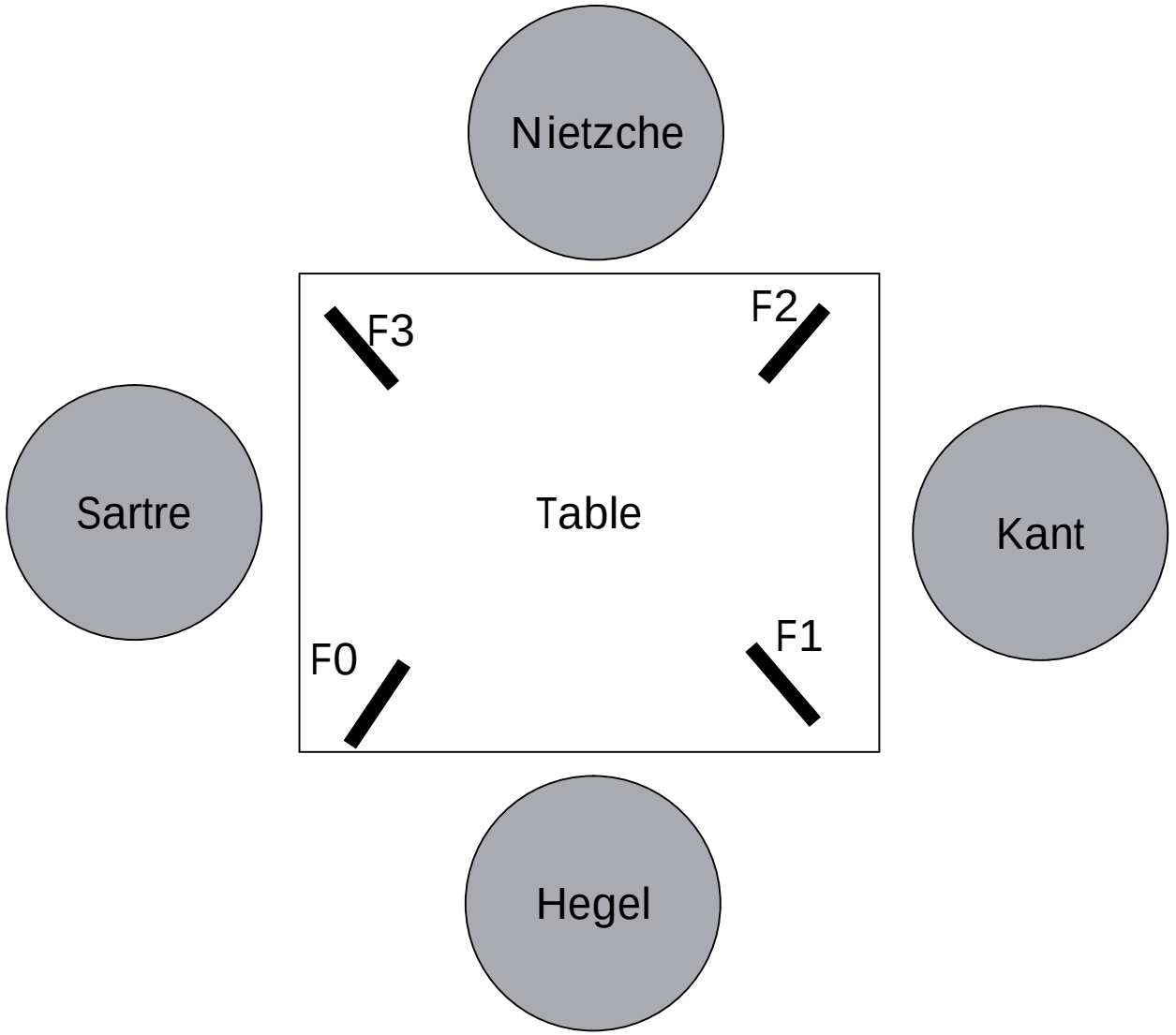
Marc Stiegler
Visiting Scholar, HP

Introduction

- Questions?
- Installing E?
- Dining Philosophers
- A Total Trust Version
- A Reliable Version
- A Satanic Version

Tell Me If I Go
Too Fast Or
Too Slow!

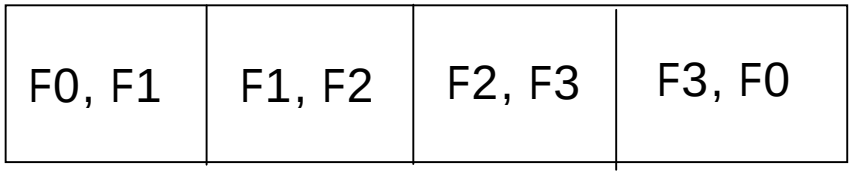
Dining Philosophers



Algorithm

- Request 2 forks at the same time
- Create ForksRequests Queue (FlexList)
- Each time a fork is released:
  - Can the foremost request be serviced?
  - Iff yes
    - Service foremost request
    - Check all other requests for serviceability

Forks Request Queue



Threat Model (Make Explicit!)

- Starve a philosopher
- Satan may subvert other philosophers
- Disregard low-level (out of model) DOS attacks (see DonutLab for solution)
- Assume TTP restaurant owners (us)

Trusted Philosophers

```
def makePhilosopher(name, fork1, fork2, table, println) {
    def philosopher {
        to think() :void {
            println <- (`$name thinking`)
            philosopher <- eat()
        }
        to eat() :void {
            when (table <- pickupForks (fork1, fork2)) ->
                                                done([heldFork1, heldFork2]) {
                when (heldFork1 <- scoopWith(heldFork2)) -> done(eating) {
                    println <- (`$name ate: $eating`)
                    heldFork1 <- putDown()
                    heldFork2 <- putDown()
                    philosopher <- think()
                } catch prob {throw(prob)}
            } catch prob {println <- (`$name starving: $prob`)}
    }   }
    philosopher <- think()
    return philosopher
}
```

Table Setup

```
def table {
    to pickupForks(fork1, fork2) :vow {
        def [forksVow, solver] := Ref.promise()
        def forksRequest := makeForksRequest([fork1, fork2], solver)
        forksRequestsMgr.processRequest(forksRequest)
        return forksVow
    }
}
def philosopherNames := [“Kant", “Sartre", “Hegel", “Nietsche"]
for i => name in philosopherNames {
    forks.push(makeFork(i))
}
bind  availableForks := forks.asSet().diverge()
def philosophers := [].asSet().diverge()
for i => name in philosopherNames {
    philosophers.addElement(
        makePhilosopher <- (name, forks[i],
                            forks[(i+1)%%forks.size()], table, println))
}
interp.blockAtTop()
```

ForksRequestsMgr Part 1 (Utilities)


```
bind forksRequestsMgr := {
    def forksRequests := [].diverge()
    def forksMatch(pair1, pair2) {
        return pair1.contains(pair2[0]) && pair1.contains(pair2[1])
    }
    def optFindPairIndex(pair) {
        for i => each in forksRequests {
            if (forksMatch(pair, each.getForks())) {return i}
        }
        return null
    }
    def forksAreAvailable(forksRequest) {
        def pair := forksRequest.getForks()
        return availableForks.contains(pair[0]) &&
          availableForks.contains(pair[1])
    }
    def removeRequest(forkPair) {
        def i := optFindPairIndex(forkPair)
        if (i != null) {forksRequests.removeRun(i, i+1)}
    }
```

ForksRequestsMgr Part 2

```
def forksRequestsMgr {
    to processRequest(forksRequest) {
        forksRequests.push(forksRequest)
        forksRequestsMgr.tryFulfillment()
    }
    to tryFulfillment() {
        if (forksRequests.size() > 0 &&
                    forksAreAvailable(forksRequests[0])) {
            for request in forksRequests.snapshot() {
                if (forksAreAvailable(request)) {
                    def pair := each.getForks()
                    request.getResolver().resolve(
                        [makeHeldFork(pair[0]),
                         makeHeldFork(pair[1])])
                    removeRequest(pair)
} } } } } }
```

Forks

```
def forks := [].diverge()
def availableForks
def forksRequestsMgr

def makeFork(position) {
    def fork {
        to __printOn(out) {out.println(`fork$position`)}
    }
    return fork
}

def makeForksRequest(forkPair, fulfillmentResolver) {
    def forksRequest {
        to getForks() {return forkPair}
        to getResolver() {return fulfillmentResolver}
    }
    return forksRequest
}
```

HeldFork

```
def makeHeldFork(fork) {
    availableForks.remove(fork)
    def heldFork {
        to getFork() {return fork}
        to putDown() {
            availableForks.addElement(fork)
            forksRequestsMgr.tryFulfillment()
        }
        to scoopWith(heldFork2) {
            println(`held forks in use: $heldFork and $heldFork2`)
            return ("full portion")
        }
        to __printOn(writer) {
            writer.print("held ")
            fork.__printOn(writer)
        }
    }
    return heldFork
}
```

Reliability Enhancements

Let the Philosophers Think!

```
def makePhilosopher(name, fork1, fork2, table, println) {
    def philosopher {
        to think() :void {
            println(`$name thinking`)
            for i in 1..1000 { def a := 1 }
            philosopher <- think() eat()
        }
        to eat() :void {
            when (table <- pickupForks (fork1, fork2)) ->
                                                done([heldFork1, heldFork2]) {
                when (heldFork1 <- scoopWith(heldFork2)) -> done(eating) {
                    println <- (`$name ate: $eating`)
                } catch prob {}
                heldFork1 <- putDown()
                heldFork2 <- putDown()
                philosopher <- eat() think()
            } catch prob {println <- (`$name starving: $prob`)}
    }   }
    philosopher <- think()
    philosopher <- eat()
    return philosopher }
```

The Difference Between Deadlock and Datalock

Kant: Did I Ask For My Forks Earlier? Did I? (I'll Just Ask Again)


```
def forksRequestsMgr {
    to processRequest(forksRequest) {
        if (optFindPairIndex(forksRequest.getForks()) == null) {
            forksRequests.push(forksRequest)
            forksRequestsMgr.tryFulfillment()
         }
    }
    to tryFulfillment() {
        if (forksRequests.size() > 0 &&
                        forksAreAvailable(forksRequests[0])) {
            for request in forksRequests.snapshot() {
                if (forksAreAvailable(request)) {
                    def pair := each.getForks()
                    request.getResolver().resolve(
                        [makeHeldFork(pair[0]),
                         makeHeldFork(pair[1])])
                    removeRequest(pair)
} } } } } }
```

Hegel: Single Fork Impairment

```
def table {
    to pickupForks(fork1, fork2) :vow {
        if (fork1 == fork2) {return null}
        def [forksVow, solver] := Ref.promise()
        def forksRequest := makeForksRequest([fork1, fork2], solver)
        forksRequestsMgr.processRequest(forksRequest)
        return forksVow
    }
}
```

```
to scoopWith(heldFork2) {
    require (fork != heldFork2.getFork())
    println(`held forks in use: $heldFork and $heldFork2`)
    return ("full portion")
}
```

Nietsche: Strange Forks


```
interface Fork {}
interface HeldFork {}

def fork implements Fork {

def heldFork implements HeldFork {


 def table {
     to pickupForks(fork1 :Fork, fork2 :Fork) :vow {


to scoopWith(heldFork2 :HeldFork) {
```

Sarte: Could You Put Down the Forks, Sarte? Put ‘em down!

- Old “makeHeldFork” renamed makeBaseFork
- New “HeldFork” is revocable forwarder to BaseFork
- PutDown() revokes (baseFork := null)
- Timer calls PutDown()

Timed HeldFork

```
def makeHeldFork(fork ) {
    var baseFork := makeBaseFork(fork)
    def heldFork implements HeldFork {
        to putDown() {
            baseFork.putDown()
            baseFork := null
        }
        match [verb, args] {
            try {
                E.call(baseFork, verb, args)
            } catch prob {println(
                `fork failed: $prob`)}
        }
    }
    timer.whenAlarm(timer.now()+2000,
        def drop(){heldFork.putDown()})
    return heldFork
}
```

Security Enhancements

Satan

- Move Philosophers to separate vats, separate jvms, separate machines
- What can he do?

```
def makeServer := <elang:interp.makeServerAuthor>(
    <unsafe>, introducer)
for i => name in philosopherNames {
    def [makePhilosopher, vatEnv, vat] :=
        makeServer(philosopherSource)
    philosophers.addElement(
        makePhilosopher <- (name, forks[i],
                    forks[(i+1)%%forks.size()], table, println))
}
```

Satanic Code Review

Conspiring Philosophers: Confinement, covert channels, separate machines

```
def makePhilosopher(name, fork1, fork2, table, println) {
    def philosopher {
        to think() :void {
            println(`$name thinking`)
            for i in 1..1000 { def a := 1 }
            philosopher <- think()
        }
        to eat() :void {
            when (table <- pickupForks (fork1, fork2)) ->
                                                done([heldFork1, heldFork2]) {
                when (heldFork1 <- scoopWith(heldFork2)) -> done(eating) {
                    println <- (`$name ate: $eating`)
                } catch prob {}
                heldFork1 <- putDown()
                heldFork2 <- putDown()
                philosopher <- eat()
            } catch prob {println <- (`$name starving: $prob`)}
    }   }
    philosopher <- think()
    philosopher <- eat()
    return philosopher }
```

A Reliable Restaurant is Already Secure!

- Security and Reliability: Two sides of the same coin?