Parallel Programming

Races and Critical Regions

Morgan Ericsson

Today

- Shared memory / state
- Races and critical regions
- Mutual exclusion

Exclusivity

- The operating system provides "exclusivity" in the following areas
  - Processor
  - Memory (address space)
- Other areas are shared
  - File system, for example

Exclusivity is not for free!

- Address spaces must be managed
  - It is expensive (CPU) to manage, create, and destroy address spaces
- A process runs on a processor
- Isolation is isolation
  - Collaboration between processes are more difficult with isolation

Why collaboration?

- Computation speed-up
- Information sharing
- Modularity
- Convenience

Collaboration

- Shared resources
  - Earlier, either global structures or passed references (cf. methods and functions)
  - Now, no access to shared mechanisms apart from files.
  - Files can be too slow

Interprocess communication (IPC)

- Shared memory
- Message passing
- Networking

Shared memory

- Two or more processes share a region of memory
- Can be used as ordinary memory
- It is up to the processes to keep the region consistent
- Fast, can be complex

Shared memory between processes

```
with SharedMemoryManager() as smm:
    l = smm.ShareableList(range(100))

    for i in l:
        print(i)
```

Message passing

- Processes communicate by sending messages
- Synchronous or Asynchronous
- Buffering
- Kernel or Process

IPC

- We will return to IPC and message passing later
- Processes can share memory
- Threads always[1] share memory

1. Not 100% true...

Remember

```go
func NumInside(n int) (m int) {
	for i := 0; i < n; i++ {
		x, y := rand.Float64(), rand.Float64()

		if math.Pow(x, 2)+math.Pow(y, 2) <= 1.0 {
			m += 1
		}
	}

	return m
}
```

Remember

- We used channels to return values
- Channels are a form of IPC, which we will return to
- Why channels?
  - return is difficult
  - which we will see later (futures)
- Why not a mutable variable?

m as a mutable parameter?

```
func NumInside(n int, m *int) {
	for i := 0; i < n; i++ {
		x, y := rand.Float64(), rand.Float64()

		if math.Pow(x, 2)+math.Pow(y, 2) <= 1.0 {
			*m += 1
		}
	}
}
```

Works fine!

```go
func main() {
	var m int
	NumInside(100_000, &m)
	fmt.Println(4.0 * float64(m) / 100_000)
}
```

What about goroutines?

```
func main() {
    var m int
    for i := 0; i < 4; i++ {
        go NumInside(100_000, &m)
    }
    time.Sleep(5 * time.Second)
    fmt.Println(4.0 * float64(m) / 100_000)
}
```

m as an instance variable?

- Estimating π five times...
  1. 7.28336
  2. 7.63884
  3. 6.91012
  4. 7.97132
  5. 6.87116
- What is wrong? How can we fix it?

Shared memory

- Shared memory and consistency
- Memory normally sequentially consistent
- When several processes or threads execute at once and access the shared memory, the consistency will not be automatically maintained

Sequentially consistent?

```
a = 10
a += 5
b = 5
a = 7
fmt.Println(a, b)
```

Remember CPU and ISA

- `a = 10` requires a store
- `a += 4` requires a load, add, and store

A shared queue

```go
type queue struct {
    data [50]int
    head int
    tail int
}

func (q queue) size() int {
    if q.tail >= q.head {
        return q.tail - q.head
    } else {
        return len(q.data) - (q.head - q.tail)
    }
}
```

A shared queue

```go
func (q queue) empty() bool {
    return q.size() <= 0
}

func (q queue) full() bool {
    return q.size() >= len(q.data)-1
}
```

A shared queue

```go
func (q *queue) enqueue(itm int) {
    if !q.full() {
        q.data[q.tail] = itm
        q.tail = (q.tail + 1) % len(q.data)
    }
}

func (q *queue) dequeue(itm *int, ok *bool) {
    if !q.empty() {
        *itm, *ok = q.data[q.head], true
        q.head = (q.head + 1) % len(q.data)
    } else { *ok = false }
}
```

A producer

```go
func producer(q *queue, lo int, hi int) {
    for i := lo; i <= hi; i++ {
        for q.full() {
        }
        q.enqueue(i)
    }
}
```

Will this work as expected?

```go
func main() {
	myq := queue{}
	go producer(&myq, 1, 100)
	go producer(&myq, 1000, 1100)
}
```

As expected

```
len=200
[1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017
1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034
1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051
1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068
1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085
1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1 2 3 4 5 6 7
8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

But not always

```
len=172
[1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017
1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034
1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051
1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068
1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 17 1082 1083 1084 1085 1086
1087 1088 22 1090 1091 1092 1093 1094 1095 1096 27 1098 1099 1100 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
94 95 96 97 98 99 100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0]
```

How much of a problem?

- Two producers enqueueing 1 000 elements each
- Run 10 000 times and check the number of times we get fewer than 2 000 elements
- About 85-90% of the time
- So, our queue rarely works!

Terminology

- *Race Condition* - When there is concurrent access to shared data and the final outcome depends upon the order of execution.
- *Critical Section* - Section of code where shared data is accessed.
- *Entry Section* - Code that requests permission to enter its critical section.
- *Exit Section* - Code that is run after exiting the critical section

Race in producers

```
P2: inserting value 1001
P1: inserting value 26
ENQ 26 at 25
ENQ 1001 at 25
P2: inserting value 1002
P1: inserting value 27
ENQ 27 at 26
ENQ 1002 at 26
```

What is going on?

```
P1: // Insert 100 money
    tmp = bank_account
    tmp = tmp + 100
    bank_account = tmp

P2: // Withdraw 50 money
    tmp = bank_account
    tmp = tmp - 50
    bank_account = tmp
```

“Serial” cases

Assume there is 250 money in the back account

```
P1#1: 250 -> tmp
P1#2: 250 + 100 -> tmp
P1#3: 350 -> bank_account
P2#1: 350 -> tmp
P2#2: 350 - 50 -> tmp
P2#3: 300 -> bank_account
```

“Serial” cases

Assume there is 250 money in the back account

```
P2#1: 250 -> tmp
P2#2: 250 - 50 -> tmp
P2#3: 200 -> bank_account
P1#1: 200 -> tmp
P1#2: 200 + 100 -> tmp
P1#3: 300 -> bank_account
```

Can interleaved cases go wrong?

```
P1#1: 250 -> tmp
P2#1: 250 -> tmp
P2#2: 250 - 50 -> tmp
P2#3: 200 -> bank_account
P1#2: 250 + 100 -> tmp
P1#3: 350 -> bank_account
```

Important

Depending on the order, we can get 200, 300, and 350.

Back to the queue

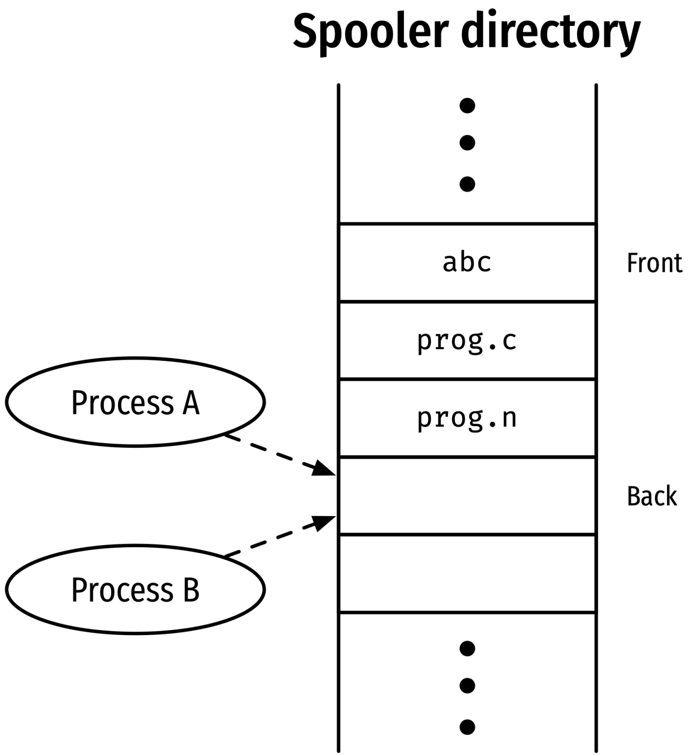
```go
func (q *queue) enqueue(itm int) {
	if !q.full() {
		q.data[q.tail] = itm
		q.tail = (q.tail + 1) % len(q.data)
	}
}
```

Critical region

» Process A and B might both end up writing to slot 7

» Consider 1 to N processors

» Race condition

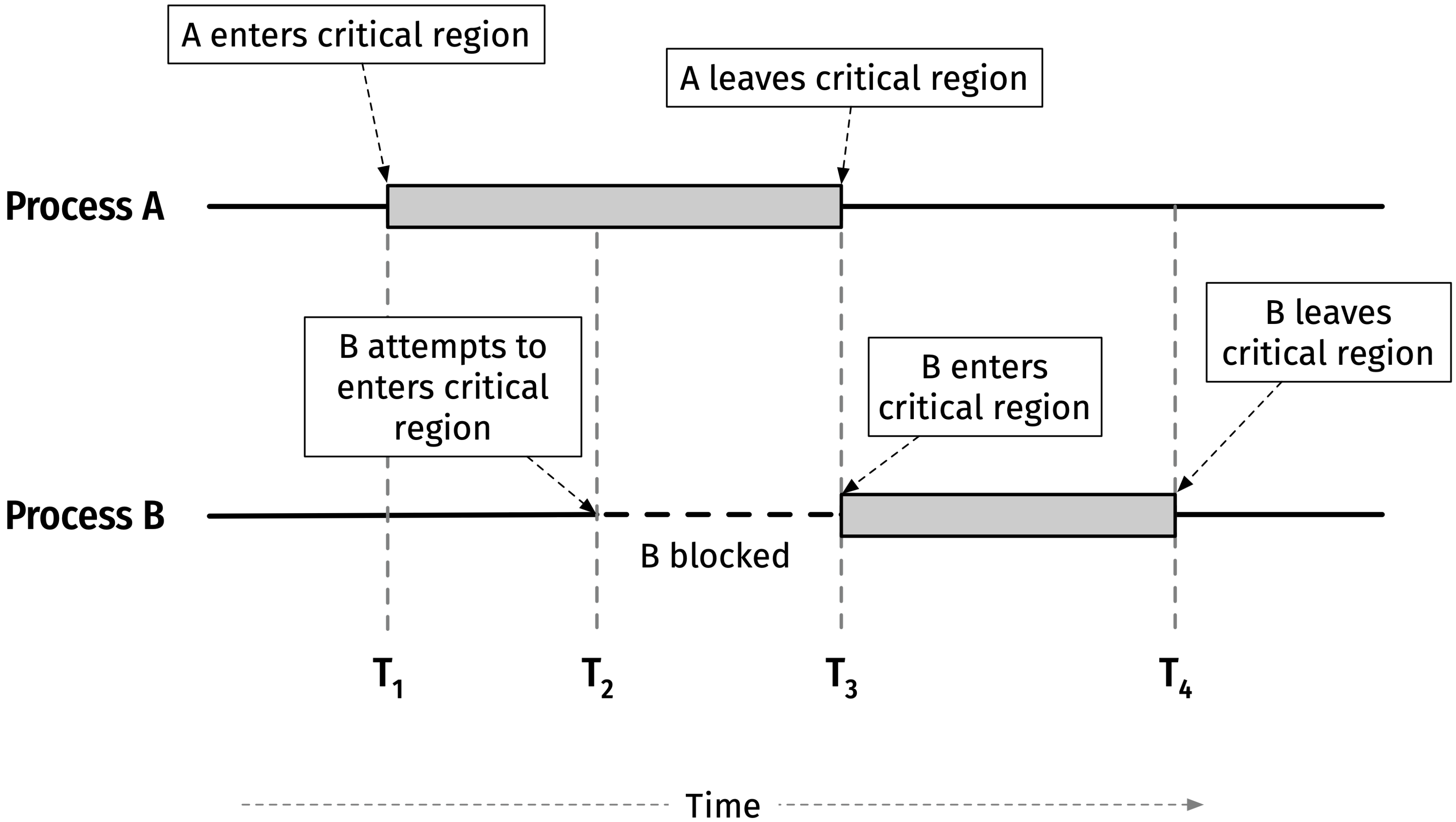


Critical region

The three following must hold to solve the critical region problem:

1. Mutual Exclusion
2. Progress
3. Bounded wait

Mutual Exclusion



Solutions

- Lock variables
  - Use a variable, lock, to decide if the shared resource is available or not
  - Does not work!
- Strict alteration
- Peterson’s solution
- Generalisations

Strict alteration

Process 0

```
for {
    for turn != 0 {}
    critical_region()
    turn = 1
    noncritical_region()
}
```

Process 1

```
for {
    for turn != 1 {}
    critical_region()
    turn = 0
    noncritical_region()
}
```

Producer with strict alteration

```go
func producer(id int, turn *int, q *queue, lo int, hi int)
    for i := lo; i <= hi; i++ {
        for *turn != id {}
        q.enqueue(i)
        *turn = (*turn + 1) % 2
    }
}
```

Producer with strict alteration

```go
func main() {
	var turn int
	myq := queue{}

	go producer(0, &turn, &myq, 1001, 2000)
	go producer(1, &turn, &myq, 5001, 6000)
}
```

Strict Alteration

- Provides mutual exclusion!
- Does not uphold all requirements
- A process not interested in the CR can block one interested!
- Violates progress

Why?

```go
func main() {
    var turn int
    myq := queue{}

    go producer(0, &turn, &myq, 1001, 1100)
    go producer(1, &turn, &myq, 5001, 6000)
}
```

Peterson's Solution

```go
type peterson struct {
    interest [2]bool
    turn int
}

func (p *peterson) enter_region(proc int) {
    other := 1 - proc
    p.interest[proc] = true
    p.turn = other
    for p.turn == other && p.interest[other] {
        runtime.Gosched()
    }
}

func (p *peterson) leave_region(proc int) {
    p.interest[proc] = false
}
```

Peterson's Solution

```
func producer(id int, p *peterson, q *queue, lo int, hi int) {
    for i := lo; i <= hi; i++ {
        p.enter_region(id)
        q.enqueue(i)
        p.leave_region(id)
    }
}
```

Peterson's Solution

- Provides mutual exclusion
- Provides progress
- Provides bounded wait
- Solution to the CR problem!

Semaphores

- Integer variable and two operations
  - Down (P)
  - Up (V)
- Down requests a lock
- Up releases a lock

Semaphores

```python
from threading import Semaphore

s = Semaphore(5)
s.acquire() # Down
# CR
s.release() # Up
```

Semaphores

- Counting semaphore
  - integer value can range over an unrestricted domain
- Binary semaphore
  - integer value can range only between 0 and 1; can be simpler to implement
- Also known as mutex locks
  - `sync.Mutex` in Go

Producer with mutex

```
type queue struct {
    data [3000]int
    head int
    tail int
    mu sync.Mutex
}
```

Producer with mutex

```go
func producer(id int, q *queue, lo, hi, slp int) {
    for i := lo; i <= hi; i++ {
        q.mu.Lock()
        q.enqueue(i)
        q.mu.Unlock()
    }
}
```

Recursive locks

- Simply, do not!
- Mutex does not allow you to lock resursively, i.e., lock the same lock again
  - This will deadlock (which we will talk about more later)

Example

```
func dummy(mu *sync.Mutex, cnt int) {
	if cnt == 0 { return }
	mu.Lock()
	dummy(&mu, cnt-1)
	mu.Unlock()
}
```

Back to the queue

```go
func (q *queue) enqueue(itm int) {
    q.mu.Lock()
    defer q.mu.Unlock()
    if !q.full() {
        q.data[q.tail] = itm
        q.tail = (q.tail + 1) % len(q.data)
    }
}
```

Readers/writers

- It can sometimes be useful to seperate readers and writers
  - Multiple readers is generally safe
- `sync.RWMutex` allows for read or write locks
  - Several can lock to read
  - One can lock to write

Waitgroups

- A waitgroup can be used to wait for a number of goroutines to finish
  - Remmeber, `sleep` is a bad idea
- `sync.WaitGroup`
  - `Add` to add something to wait for
  - `Done` to signal that something is done

Example

```
var wg sync.WaitGroup
for i := 0;i<10;i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        doSmth()
    }()
}
wg.Wait()
```

Deadlocks?

- We will get there...

Next time

- Channels and concurrency patterns