Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського» Інститут прикладного системного аналізу
Кафедра системного проектування

ЗВІТ

про виконання лабораторної роботи №2 з дисципліни
«Паралельні обчислення»

Виконав:
Студент III курсу
Групи ДА-92
Шляхов Данило Сергійович
Варіант №23

Київ – 2022

1. **Мета роботи:** Метою роботи є набуття навичок роботи з потоками при програмуванні на мові Java або будь-якій іншій.
2. **Склад робочого місця:**

   - Обладнання: IBM-сумісний персональний комп'ютер.
   - Програмне забезпечення: операційна система Windows, Java SDK версії 1.2.2 або вище.
3. **Завдання за варіантом 23:**

Програма моделює обслуговування двох процесів з різними параметрами двома центральними процесорами і двома чергами. Кожен процес надходить в свою чергу і обслуговується своїм процесором. Якщо перший процесор вільний і черга першого потоку процесів порожня, процесор обирає процес із другої черги, проте, якщо під час обробки процесу, генерується процес першого потоку, обробка процесу переривається і він повертається в свою чергу як перший в черзі. Визначити максимальні довжини черг і кількість перерваних процесів другого потоку.

Хід роботи

Для виконання роботи була обрана мова програмування GoLang.

Процесу відповідає структура CPUProcess.
Процессору відповідає функція з такою сигнатурою:

```
type CPU func(ctx context.Context, wg sync.WaitGroup)
```

Процеси генеруються в функції ProcessGenerator. Вона генерує процеси з випадковим часом виконання та в випадкові моменти часу. Процес з шансом 50% потрапить в одну з черг.

Черзі відповідає структура FIFO з бібліотеки goconcurrentqueue.

Структура CPUProcess описана нижче. Канал ch дозволяє зрозуміти коли процесс закінчив роботу. worktime – параметр, що вказує скільки виконується процес.

Функція work – основний метод структури. На вхід він отримує контекст завдяки якому можна перервати виконання процесу. Якщо через час який вказаний в полі worktime не буде викличено функцію відміни, процес завершиться корректно. В іншому разі, виконання буде завершено.

```
type CPUProcess struct {
    ch       chan int
    workTime time.Duration
}

func (process *CPUProcess) SetChannel(ch chan int) {
    process.ch = ch
}

func (process *CPUProcess) GetWorkTime() time.Duration {
    return process.workTime
}

func (process *CPUProcess) Work(ctx context.Context) {
    defer func() {
        process.ch <- 0
    }()
    select {
    case <-time.After(process.workTime):
        fmt.Println("done")
    case <-ctx.Done():
        fmt.Println("halted operation")
    }
}
```

Функція ProcessGenerator описана нижче.

```
func ProcessGenerator(ctx context.Context, firstQueue, secondQueue *queue.FIFO, wg
*sync.WaitGroup) {
    wg.Add(1)
    defer wg.Done()
    for {
        select {
        case <-ctx.Done():
            goto outOfLoop
        case <-time.After(time.Duration(<-randomNumbers)%maxGenerateTime +
minGenerateTime):
            process := CPUProcess{
                workTime: time.Duration(<-randomNumbers)%maxWorkTime + minWorkTime,
```

```
            }

            if <-randomNumbers%2 == 0 {
                m1.Lock()
                err := firstQueue.Enqueue(process)
                if err != nil {
                    m1.Unlock()
                    continue
                }
                if firstQueue.GetLen() > int(q1MaxLength) {
                    atomic.StoreInt64(&q1MaxLength, int64(firstQueue.GetLen()))
                }
                fmt.Println("Generated process 1, size q1: ", firstQueue.GetLen())
                m1.Unlock()
            } else {
                m2.Lock()
                err := secondQueue.Enqueue(process)
                if err != nil {
                    m2.Unlock()
                    continue
                }
                if secondQueue.GetLen() > int(q2MaxLength) {
                    atomic.StoreInt64(&q2MaxLength, int64(secondQueue.GetLen()))
                }
                fmt.Println("Generated process 2, size q2: ", secondQueue.GetLen())
                m2.Unlock()
            }
        }
    }
outOfLoop:
}
```

В функції main запускається генератор випадкових чисел:

```
    //creating random numbers generator routine
    randomNumbers = make(chan int64)
    go NumberGenerator(randomNumbers)
```

Створюються черги для процесів:

```
    //creating concurrentsafe queues
    q1 := queue.NewFIFO()
    q2 := queue.NewFIFO()
```

Запускається генератор процесів:

```
    //creating process generator
    genctx := context.Background()
```

```
    genctx, gencancel := context.WithCancel(genctx)
    go ProcessGenerator(genctx, q1, q2, wg)
```

А також обидва процесори.
Розглянемо спочатку процесор 2, оскільки він простіший.

Створюються два канали. Канал procchan призначений для отримання процесорів з другої черги. Waitchan призначений для отримання сигналу про закінчення виконання процесу.

```
        procchan := make(chan CPUProcess)
        waitchan := make(chan int)
```

Після цього запускається нескінченний цикл.

Далі запускається горутина, що буде очікувати появу процесу в другій черзі. Горутину можна зупинити за допомогою контексту.

```
            deqctx := context.Background()
            deqctx, deqcancel := context.WithCancel(deqctx)
            go func(ctx context.Context) {
                process, err := q2.DequeueOrWaitForNextElementContext(ctx)
                if err != nil {
                    return
                }
                procchan <- process.(CPUProcess)
            }(deqctx)
```

Селект дозволяє обрати один з двох випадків:

1. Процессор отримує процес. В такому випадку починається його виконання.
2. В іншому випадку приходить сигнал про завершення роботи процесору. Зупиняється очікування процесу з черги та завершується робота процесору.

```
            select {
            case process = <-procchan:
                fmt.Println("cpu2 got process. ", "worktime: ",
process.GetWorkTime())
                process.SetChannel(waitchan)
                go process.Work(prctx)
            case <-ctx.Done():
                fmt.Println("stopped cpu2")
                deqcancel()
                prcancel()
                return
```

```
            }
```

Наступний селект чекає на:

1. Завершення роботи процесу. В такому випадку виводиться повідомлення та процесор заходить на новий цикл очікування процесу.
2. Завершення роботи процесору. В такому випадку відміняється робота процесу і процесор завершує свою роботу.

```
3.              select {
4.              case <-waitchan:
5.                  fmt.Println("cpu2 finished processing")
6.              case <-ctx.Done():
7.                  prcancel()
8.                  <-waitchan
9.                  fmt.Println("stopped cpu2")
10.                 return
11.             }
```

Процесор отримує сигнал про завершення своєї роботи з main.

Розглянемо процесор 1.

На відміну від процесору 2, процесор 1 очікує одночасно на процес з першої та другої черги.
Якщо приходить процесор з першої черги, алгоритм схожий з другим процесором. Очікування на процес з другої черги припиняється. Отриманий процес виконується або до кінця, або до завершення роботи процесору.

```
select {
            case process = <-proc1chan:
                deq2cancel()
                fmt.Println("cpu1 got process from q1", "worktime: ",
process.GetWorkTime())
                process.SetChannel(waitchan)
                go process.Work(prctx)

                select {
                case <-waitchan:
                    fmt.Println("cpu1 finished processing")
                case <-ctx.Done():
                    prcancel()
                    <-waitchan
```

```
                    fmt.Println("stopped cpu1")
                    return
                }
```

Якщо приходить процес з другої черги, очікування на процес з першої черги не припиняється. Починається виконання процесу:

1. До завершення виконання;
2. До появи процесу з першої черги.
3. До завершення роботи процесору.

В другому випадку процесор поверне процес в другу чергу і почне виконувати перший процес згідно алгоритму наведеного вище.

```
case process = <-proc2chan:
                fmt.Println("cpu1 got process from q2", "worktime: ",
process.GetWorkTime())
                process.SetChannel(waitchan)
                go process.Work(prctx)
                var tmp CPUProcess
                select {
                case <-waitchan:
                    fmt.Println("cpu1 finished processing")
                case tmp = <-proc1chan:
                    prcancel()
                    <-waitchan
                    prctx, prcancel = context.WithCancel(prctx)
                    atomic.AddInt64(&interruptedProcesses, 1)
                    m2.Lock()
                    q2.Enqueue(process)
                    if q2.GetLen() > int(q2MaxLength) {
                        atomic.StoreInt64(&q2MaxLength, int64(q2.GetLen()))
                    }
                    fmt.Println("size q2: ", q2.GetLen())
                    m2.Unlock()
                    process = tmp
                    fmt.Println("cpu1 return process to q2")
                    fmt.Println("cpu1 got process from q1", "worktime: ",
process.GetWorkTime())
                    process.SetChannel(waitchan)
                    go process.Work(prctx)

                    select {
                    case <-waitchan:
                        fmt.Println("cpu1 finished processing")
                    case <-ctx.Done():
                        prcancel()
```

```
                <-waitchan
                fmt.Println("stopped cpu1")
                return
            }
        case <-ctx.Done():
            prcancel()
            deq1cancel()
            deq2cancel()
            <-waitchan
            fmt.Println("stopped cpu1")
            return
        }
```

Результати виконання (кожного разу різні, оскільки процеси створюються випадковим чином):

First queue max length: 4

Second queue max length: 2

Second queue processes interrupted: 7

Посилання на GitHub репозиторій: Лаба2

Висновок

Під час виконання роботи була розроблена програма для вирішення задачі на мові GoLang. Були реалізовані абстракції для процесу, процесору та черги процесів. Для переривання виконання goroutines були використані контексти.