BTP
Illustration of Using BPJ Library as Behavioral Programming in Java by Examples

Pintu Kumar 120101050

Katravath Manoj 120101031

Car Manufacturing

- Six tasks:
  - Chassis
  - Tires
  - Seats
  - Engine(assume this includes everything under the hood and the steering wheel)
  - The top cover
  - Painting
- Constraints
  - Tires, seats or engine can not be added until the chassis is placed on the belt
  - The car top can not be added until tires, seats and the engine are put in.
  - The car can not be painted until the top is put on.

Car Manufacturing contd..

- A stop on the conveyer belt in the car company has four technicians assigned to it- Abe, Bob, Charlie and Dave.
- Skills' distribution:

Skills	Technician
Adding Tires	Abe
Painting	Abe
Put the chassis on the belt	Bob
Attach the seats	Charlie
Add the engine	Dave
Add the top	Dave

Car Manufacturing contd..

- Write a program for Abe, Bob, Charlie and Dave to be able to work on the car without violating task order outlined.
- Events : All have single instances
  - Chassis
  - Engine
  - Painting
  - Seats
  - TheTopCover
  - Tires

Car Manufacturing contd..

- Bthreads & Priority

BThread	Priority
ApplyChasis	1.0
ApplyEngine	2.0
ApplyPainting	2.1
applySeats	2.2
ApplyTheTopCover	2.3
ApplyTires	3.3

Car Manufacturing contd..

- Scenarios
- ********* Starting 6 scenarios **********
- Application Parameters: nTasks=6
- Event #1: Chassis(ID=0) requested by ApplyChassis
- Bob applying chassis
- Event #2: Tires(ID=5) requested by ApplyTires
- Abe applying tires
- Event #3: Seats(ID=3) requested by ApplySeats
- Charlie applying seats
- Event #4: Engine(ID=1) requested by ApplyEngine
- Dave applying engine
- Event #5: TheTopCover(ID=4) requested by ApplyTheTopCover
- Dave applying Car Top
- Event #6: Painting(ID=2) requested by ApplyPainting
- Abe applying painting

Car Manufacturing contd..

- Problem : To maintain the order of the tasks given in the constraints
- Solution: Could have been solved by using lock and mutexes in other paradigms, but since, BPJ automatically solves the problem of concurrency and synchronization by the following function:
- *bp.bSync(requestedEvents, watchedEvents, blockedEvents);*
- *We just need to wait for the events (watchedEvents) on which current task is dependent on.*

BThread States: Car Manufacturing

BThread	State
ApplyChassis	N, A
ApplyEngine	N,W, A
ApplyPainting	N,W, A
ApplySeats	N,W, A
ApplyTheTopCover	N,W1,W2,W3, A
ApplyTires	N,W, A

N denote not selected for execution
W waiting for some event
A applying state

Scenarios(Sample Runs, state sequences)

<ApplyChassis, ApplyEngine, ApplyPainting, ApplySeats, ApplyTheTopCover, ApplyTires>

Init -> <N,N,N,N,N,N>

Chasis -> <A,W,W,W,W1,W>

Tires -> <N,W,W,W,W1,A>

Seats -> <N,W,W,A,W2,N>

Engine -> <N,A,W,N,W3,N>

TheTopCover -> <N,N,W,N,A,N>

Painting -> <N,N,A,N,N,N>

End-> <N,N,N,N,N,N>

Dining Philosophers

- There are several philosophers sitting on a round table numbered 1..n
- There is a large bowl of spaghetti at the center
- There is a fork between each pair of philosopher
- Each philosopher may only use the fork on her right and the fork on her left
- A philosopher needs the two forks to eat
- Each philosopher when finishes eating she puts down the forks and starts thinking again
- We use BPJ’s synchronization and concurrency handling properties and program their behaviors.

Dining Philosophers

- Events
  - PickUp: There are n instances of pickUpRightFork and n of pickUpLeftFork
  - PutDown: There are n instances
- Bthreads and Priorities

Bthreads	Priority
Phil (n instances for n philosophers)	1.0-n.0
Fork (n instances for there are n forks for n philosophers)	(n+1).0-(2*n).0
Logger	(2*n+1).0
DeadLockScenario	(2*n+2).0
DeadLockRightHanded	(2*n+2).0

Dining Philosophers contd..

- Scenario
- ********* Starting 10 scenarios **********
- Application Parameters: nPhils=4, leftyPhil=true
- Event #1: PickUp-F0-by-P0(ID=1) requested by P-0
- LOG:PickUp-F0-by-P0(ID=1)
- Event #2: PickUp-F3-by-P0(ID=11) requested by P-0
- LOG:PickUp-F3-by-P0(ID=11)
- Event #3: PutDown-F0(ID=3) requested by P-0
- LOG:PutDown-F0(ID=3)
- Event #4: PutDown-F3(ID=12) requested by P-0
- LOG:PutDown-F3(ID=12)
- Event #5: PickUp-F1-by-P1(ID=4) requested by P-1
- LOG:PickUp-F1-by-P
- Event #6: PickUp-F0-by-P1(ID=2) requested by P-1
- LOG:PickUp-F0-by-P1(ID=2)
- Event #7: PutDown-F1(ID=6) requested by P-1
- LOG:PutDown-F1(ID=6)
- 1(ID=4)
- Event #8: PutDown-F0(ID=3) requested by P-1
- LOG:PutDown-F0(ID=3)
- Event #9: PickUp-F2-by-P2(ID=7) requested by P-2
- LOG:PickUp-F2-by-P2(ID=7)
- Event #10: PickUp-F1-by-P2(ID=5) requested by P-2
- LOG:PickUp-F1-by-P2(ID=5)
- Event #11: PutDown-F2(ID=9) requested by P-2
- LOG:PutDown-F2(ID=9)
- Event #12: PutDown-F1(ID=6) requested by P-2
- LOG:PutDown-F1(ID=6)
- Event #13: PickUp-F2-by-P3(ID=8) requested by P-3
- LOG:PickUp-F2-by-P3(ID=8)
- Event #14: PickUp-F3-by-P3(ID=10) requested by P-3
- LOG:PickUp-F3-by-P3(ID=10)
- Event #15: PutDown-F3(ID=12) requested by P-3
- LOG:PutDown-F3(ID=12)
- Event #16: PutDown-F2(ID=9) requested by P-3
- LOG:PutDown-F2(ID=9)

Dining Philosophers contd..

- Phil States
  - T (Thinking)
  - 1(one fork Up)
  - E (Two fork Up, Eating)
  - F (Finished Eating)
- Fork States
  - UL (Up in left hand)
  - UR (Up in right hand)
  - D (Down)
- Program states will be the Cross product of the BThread states. E.g. in case of 3 philosophers <T,D,T,D,T,D> is the initial state when all philosophers are thinking and no-one has lifted any fork.

Dining Philosophers contd..

- We have implemented Resource Hierarchy Algorithm, where, for deadlock free execution, symmetry breaking is done by enforcing last philosopher to lift right fork first in case of left-handed philosophers, and enforcing that to lift left fork first in case of right-handed philosophers.
- Starvation and fairness we have not taken in consideration, since Resource Hierarchy Algorithm is not starvation free.
- Deadlock can be detected by introducing two BThreads, one for left handed philosophers and one for right handed philosophers. Deadlock occurs only in two scenarios (The two BThreads work on these principles only):
  - Either each philosopher picks up her left fork first and waits for the right fork
  - Or each philosopher picks up her right fork first and waits for the left fork

Dining Philosophers contd..

- Deadlock Scenario (n=3)
  - Init-> [T,D,T,D,T,D]
  - PickUp-F0-by-P0->[1,U,T,D,T,D]
  - PickUp-F1-by-P1->[1,U,1,U,T,D]
  - PickUp-F2-by-P2->[1,U,1,U,1,U]
- [1,U,1,U,1,U] is a deadlock state
- Here each line of the form <event> <State> describes a composite state of the entire program along the path and the event whose triggering led the program to transition from preceding state to the current.
- Deadlock detection mechanism we have implemented is the BThread which detects deadlock, waits for all philosophers lifting their left fork in case of leftyPhil is true, i.e. all philosophers are lefthanded, i.e., each philosopher lifts her left fork first, and if philosophers are right handed i.e. all philosophers lift their right fork first, same procedure with right instead of left is employed.

Producer-Consumer

- Two processes, producer and consumer share a common fixed size buffer used as a queue.
- Producer generates the data and puts into the buffer
- Consumer consumes the data by removing it from the buffer one at a time
- Constraints:
  - Producer shouldn’t add data to the buffer if it is full
  - Consumer shouldn’t remove data from empty buffer

Producer-Consumer Contd..

- General Solution
- Add two library routines sleep and wakeup
- When sleep is called , the caller is blocked until another process wakes it up by using wakeup routine
- Add a global variable itemCount to hold the number of items in the buffer. But this solution can lead to deadlock.
- So we solve this using two semaphores fillCount and emptyCount
  - fillCount : #items present in the buffer
  - emptyCount: #items spaces available in the buffer

Producer-Consumer Contd..

- When a new item is put into the buffer increment fillCount and decrement emptyCount
- Put producer to sleep if it tries to decrement emptyCount when it is 0.
- Wake producer up when item is consumed next time and emptyCount is incremented
- The solution can lead into two or more processes reading or writing into the same slot at the same time, so to overcome this execute a critical section with mutual exclusion.

```
Int itemCount = 0;
Procedure producer(){
        while(true){
                item = produceItem();
                if(itemCount == BUFFER_SIZE){
                        sleep();
                }
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if(itemCount == 1){
                wakeup(consumer);
        }
    }
 }
```

```
Procedure consumer(){
        while(true){
                if(itemCount == 0){
                        sleep();
                }
                item = removeItemFromBuffer();
                itemCount = itemCount -1 ;
                if(itemCount = BUFFER_SIZE - 1){
                        wakeup(producer);
                }
                consumeItem(item);
        }
}
```

Race Condition leading to deadlock

1. Consumer had just read the variable itemCount, noticed it's 0 and just about to move inside if block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases itemCount.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately the consumer wasn't yet sleeping, and wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again.
6. The producer will loop until the buffer is full, after which it will also go to sleep

Solution using semaphores

- Semaphores solve the problem of lost wakeup calls.

```
Semaphore fillCount = 0;
Semaphore emptyCount = BUFFER_SIZE;
Procedure producer(){
        while(true) {
                item = produceItem();
                down(emptyCount);
        putItemIntoBuffer(item);
                up(fillCount);
        }
}
```

```
Procedure consumer(){
        while(true){
                down(fillCount);
                item = removeItemFromBuffer();
                up(emptyCount);
                consumeItem(item);
        }
}
```

Producer-Consumer Our Soln.

- Events
  - Consume
  - Produce
  - ResourceEmpty
  - ResourceFull
- BThreads & Priority

BThread	Priority
Producer	3.0
Consumer	3.1
FullResourceChecker	2.0
EmptyResourceChecker	2.1
BlockProduce	1.0
BlockConsume	1.1

Behavior of Bthreads

Producer: Increment nResources and raise event Produce

FullResourceChecker: Wait for Produce event and check if nResources equals limitRes, raise ResourceFull event.

BlockProduce: Wait for ResourceFull event and Block Produce Event

Consumer: Decrement nResources and raise event Consume.

EmptyResourceChecker: Wait for Consume event and check if nResource equals 0, raise ResourceEmpty Event.

BlockConsume: Wait for ResourceEmpty Event and block Consume event.

Here nResources keeps track of currently available #items in the buffer, and limitRes is the number of maximum resources to be produced or buffer size limit.

BThreads & States

Bthreads	States
Producer	N, P
Consumer	N, C
FullResourceCheker	N,W, F
EmptyResourceChecker	N,W, E
BlockConsume	N,W, B
BlockProduce	N,W, B

N => Not Selected to run in current state, P => Producing, C => Consuming, W => Waiting state, F => Resource Full state, E => Resource Empty State, B => Requesting blocking of some event

Scenarios(Example ,Sample Run)

Init -> <P, W, W, N, W, W>

Produce Resource No. 1 -> <P, W, W, N, W, W>

.

.

Produce Resource No. 10 -> <N, F, W, N, W, W>

ResourceFull -> <N, F, B, N, W, W>

Consume Resource No. 10-> <N,W,W,C,W,W>

.

.

Consume Resource No. 1 -> <N,W,W,N,E,W>

ResourceEmpty -> <N,W,W,N,E, B>

States: <Producer,ResourceFull,BlockProduce,Consumer,ResourceEmpty,BlockConsume>

Conclusion: Producer-Consumer

- Only one producer and one consumer is considered
- The implementation allows first producer to produce till limit specified and then consumer starts consuming.

Print Prime Numbers till n

- A generator BThread requests events labelled with all integers starting from 2 to n
- Another BThread (PrintLabelBThread) prints the label associated with the current event.
- Yet another Bthread (BlockNonPrimesBThread) listens on the same event and blocks all the events labelled with the multiple of the label of the current event.
- Events
  - NumberEvent :has n instances

BThreads and their Priorities and states

BThread	Priority	State
GeneratorBThread	3.0	N,A
PrintLabelBThread	2.0	N,W,P
BlockNonPrimesBThreads	1.0	N,W,B

N denotes BThread is inactive(not chosen in current run)
A denotes Bthread is active (chosen and performing some task)
W denotes BThread is waiting on some event
B denotes BThread has requested to block some event

Sample output: Print Primes till n

- Application Parameters: n=20
- Event #1: 2(ID=2) requested by 2
- 2
- Event #2: 3(ID=3) requested by 3
- 3
- Event #3: 5(ID=5) requested by 5
- 5
- Event #4: 7(ID=7) requested by 7
- 7
- Event #5: 11(ID=11) requested by 11
- 11
- Event #6: 13(ID=13) requested by 13
- 13
- Event #7: 17(ID=17) requested by 17
- 17
- Event #8: 19(ID=19) requested by 19
- 19

Sample Scenario

< GeneratorBThread , PrintLabelBThread , BlockNonPrimesBThreads >

For n = 10,

Init -> <A,W,W>

NumberEvent[2] -> <A,P,B>

NumberEvent[3] -> <A,P,B>

NumberEvent[5] -> <A,P,B>

NumberEvent[7] -> <A,P,B>

End -> <N,N,N>

Conclusion: Print Prime numbers

- This was a simple program just to illustrate how we can do it in behavioral programming paradigm.

- Conclusion
  - We implemented four simple programs, in BPJ Library as a illustration of how to program applications in Behavioral Programming using BPJ Library, namely , dining philosopher problem(Resource Hierarchy Algorithm), Car Manufacturing Problem (To meet constraints of order of the different tasks), Producer Consumer Problem (Sample Execution) and Printing Prime Numbers till a specified limit.
  - We couldn’t verify the constraints through Programming by specifying anti-scenario (due to ,may be ,a bug in BPJ library, which is giving error as null-pointer exception while running the program with anti-scenario included).
  - Also we couldn’t do model-checking to improve our application behaviors.

Thank You

Bthread	Priority	States
Bob	1.0	RC
Abe	4.0	WC,RT,WCT,RP
Charlie	3.0	WC,RS
Dave	2.0	WC,RE,WS,WT,RCT

Execution Sequences: Car Manufacturing

Event -> <Bob,Abe,Charlie,Dave>

Init -> <RC,WC,WC,WC> : Only Bob is requesting an event and is Chassis, all other are waiting for Chassis

Chassis -> <N,RT,RS,RE> : Dave has highest priority over other two, so Engine is selected to be triggered

Engine -> <N,RT,RS,WS> : Charlie has higher priority than Abe, so Seats is selected to be triggered

Seats -> <N,RT,N,WT> : Only Abe has requested an event Tires

Tires -> <N,WCT,N,RCT> : Only Dave has requested an event TheTopCover

TheTopCover -> <N,RP,N,N> : Only Abe has requested an event Painting

Painting -> <N,N,N,N>