Lesson 11

Practical Examples

(Ch 5-9 [BenA 06])

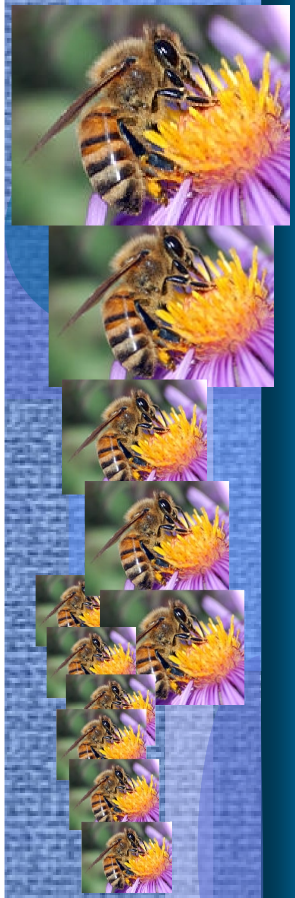
Example Problem

Problem Features

System Features

Various Concurrency Solutions

A Bear, Honey Pot and Bees



- Friendly bees are feeding a trapped bear by collecting honey for it. The life of the trapped bear is just eating and sleeping.
- There are N bees and one bear. The size of the pot is H portions.
- The bees carry honey to a pot, one portion each bee each time until the pot is full. Or maybe more?
- When the pot is full, the bee that brought the last portion wakes up the bear.
- The bear starts eating and the bees pause filling the pot until the bear has eaten all the honey and the pot is empty again. Then the bear starts sleeping and bees start depositing honey again.

[Andrews 2000, Problem 4.36]

Problem Features

- Thousands or millions of bees (N bees), one bear
  - Collecting honey (1 portion) may take very long time
  - Eating a pot of honey (H portions) may take some time
  - Filling up the pot with one portion of honey is fast
  - Same solution ok with N=1000 or N=100 000 000 ?
  - Same solution ok with H=100 or H=1 000 000 ?
  - Same solution ok for wide range of N & H values?
- Unspecified/not well defined feature
  - Could (should) one separate permission to fill the pot, actually filling the pot, and possibly signalling the bear
  - If (one bee) filling the pot is real fast, this may not matter
  - If (one bee) filling the pot takes time, then this may be crucial for performance
  - Can pot be filled from far away?
- What if more than one bears?

Maximize Parallelism

- All bees concurrently active, no unnecessary blocking
- Bees compete only when filling up the pot
  - Must wake up bear when H portions of honey in pot
  - Must fill up the pot one bee at a time
    - Is this important or could we modify specs?
    - How big is the mouth of the pot?
  - Competing just to update the counter would be more efficient?
    - Is waking up the bear part of critical section?
  - What is the real critical section?

Why?

Maximize Parallelism (contd)

- Bear wakes up only to eat and only when pot is full
- Bees blocked (to fill the pot) only
  - When bear is eating
  - When waiting for their turn to fill the pot
    - Or to synchronize with other bees

Concurrency Needs

- When is mutex (critical section) needed?
  - A bee is filling the pot or the bear is eating
- When is synchronization needed?
  - Bees wait for earlier bee to fill the pot
    - Each bee may wait before filling the pot
  - Bees wake up the bear to eat
    - Last (H^{th}) bee wakes up bear after filling the pot
  - Bear lets all bees to resume filling the pot
    - Bear allows it after emptying the pot
- When is communication needed?
  - Must know when pot is full? Nr portions in pot now?
  - What if “honey” would be information in buffer?

Environment

- Computational object level
  - Bees and bear are threads in one application?
    - Threads managed by programming language?
    - Threads managed by operating system?
  - Bees and bear are processes?
    - Communication with progr. language utilities?
    - Communication with oper. system utilities?
- System structure
  - Shared memory uniprocessor/multiprocessor?
  - Distributed system?
  - Networked system?

Busy Wait or Suspended Wait

- Bear waits a long time for full pot?
  - Suspended wait would be better (unless lots of processors)
- Bees wait for their turn to fill the pot?
  - Waiting for turn takes relatively long time
    - Earlier bees fill the pot
    - Bear eats the honey
  - Suspended wait ok
- Bees wait for their turn only to update counters?
  - Relatively long time to wait for turn
  - Suspended wait ok
  - If mutex is only for updating counters (not for honey fill-up turn, or bear eating), busy wait might be ok

Evaluate Solutions

- Does it work correctly?
  - Mutex ok, no deadlock, no starvation
- Does it allow for maximum parallelism?
  - Minimally small critical sections
  - Could bees fill up the jar in parallel?
- Is this optimal solution?
  - Overall processing time? Overall communication time?
  - Processor utilization? Memory usage?
  - Response time? Investments/return ratio?
- Is this solution good for current problem/environment?
  - Bees and bear are threads in Java application in 4-processor system running Linux?
  - There are 20000 bees, collecting honey takes 15 min, depositing one portion in pot takes 10 sec, 5000 portions fill the pot, and bear eats the honey in pot in 10 minutes?

Solution with Busy Wait Locks

- Can use locks both for mutex and for synchronization
  - – Problem: busy wait for bear
    - Bear waits a long time for full honey pot (some bears do not like waiting!)

```
Int          portions = 0;         # portions in the pot
Lock_var D = 0 = ”open”;     # mutex to deposit honey in pot
             E = 1 = ”closed”; # permission to eat honey
```

```
implem. dependent:
Lock_var D = 1; #open
               E = 0; #locked
```

Solution with Locks (contd)

```
Int         portions = 0; # portions in the pot
Lock_var D = 0; # mutex to deposit honey in pot
            E = 1; # permission to eat honey
```

```
process bee [i=1 to N] () {
while (true) {
    collect_honey();
    lock (D);  # only one bee advances at a time
    portions++;
    fill_pot();
    if (portions == H) unlock (E); # wakeup bear, keep lock
    else unlock (D)  # let next bee deposit honey
   }
 }
```

```
process bear () {
while (true) {
    lock (E);       # busy-wait, hopefully OK?
    eat_honey();
    portions = 0;
    unlock (D);   # let next bee deposit honey
}   }
```

Discuss

Semaphore

```
sem mutex = 1,     # mutual exlusion
      pot_full = 0;  # synchr bear/bees
int portions;          # portions in the pot
```

```
process bee [i=1 to N] {
   while (true) {
      collect_honey();
      P (mutex);
         fill_pot();
         portions++;
         if (portions == H)
            V (pot_full);  # let the bear eat honey, pass mutex baton
         else
            V (mutex);                   # let other bees to fill the pot
      }
}
```

```
process bear {
   while (true) {
         P (pot_full);   # wait until the pot is full   -- sleep
            eat_all_honey();                       #  -- eat
            portions=0;
         V (mutex);      # let bees start filling the pot again
}        }
```

Discuss

Monitor

- Use monitor only for mutex and synchronization
  - Automatic mutex
  - Use of monitor condition variables for synchronization solution for bees and bear
- What type of signalling semantics is in use?
  - E < S < W, i.e., IRR? Assume now no-IRR.

```
process bee [i=1 to N] {
   while (true) {
        collect_honey();
       pot.into_pot();
        depostit_honey();
       pot.deposit_done();
} }
```

```
process bear() {
   while (true) {
      pot.wait_full();
      eat_honey();
      pot.empty_pot();
}   }
```

Monitor
(many bees can fill at a time)

```
monitor pot {   # no IRR
   int fill=0, portions=0; cond pot_full, pot_empty;

   procedure fill_perm () {
      while (fill+portions == H) waitC (pot_empty);
      fill++;   # nr of bees with fill permission
      }

   procedure fill_done () {
      fill--; portions++;
      if (portions == H) signalC (pot_full);
      }

   procedure wait_full () {
      if (portions < H) waitC (pot_full);
      }

   procedure empty_pot () {
      portions = 0;
      signal_allC (pot_empty)  # wake up all
}     }
```

```
process bee [i=1 to N] {
while (true) {
     collect_honey();
     pot.fill_perm();
          fill_pot();
     pot.fill_done();
}   }
```

```
process bear() {
while (true) {
     pot.wait_full();
       eat_honey();
     pot.empty_pot();
}   }
```

Discuss

ADA Protected Object (many fills at a time)

```
..
private   perms :=0, portions := 0;
…
protected body pot  is
   entry get_perm when perms < H is
      begin
          perms=perms+1;
      end get_perm;
procedure filled is
      begin
          portions = 0;
      end filled;
entry wait_full  when portions == H is
      begin    # empty body
      end wait_full;
procedure empty_pot  is
      begin
          perms = 0; portions = 0;
     end empty_pot;
end pot;
```

```
process bee [i=1 to N] {
   while (true) {
         collect_honey();
         pot.get_perm;
         fill_pot()
         pot.filled()
   }
}     # not Ada syntax
```

```
process bear() {
   while (true) {
     pot.wait_full;
     eat_honey();
     pot.empty_pot;
   }
}
```

How to modify to do fill_pot() one at a time?

Channels

- Processes communicate via messages to/from channels
  - Difficult to do in distributed environment
  - OK in shared memory systems
- Automatic mutex in message primitives
- Synchronization occurs at message send/receive
  - Messages act as tokens
  - Messages used for synchronization and communication
    - Number of portions in pot is transmitted in messages

```
chan deposit();  # bees receive from this channel
                 # a permission to deposit
                 # and nr of current portions in pot
chan wakeup();         # the bear receives from here
                       # a permission to eat
```

Channels

```
chan deposit();
chan wakeup();
```

```
process bee [i=1 to N] () {
   while (true) {
     collect_honey ();
     receive (deposit_perm, portions);  # only one bee advances at a time
     portions++;
     fill_pot ();                # Is it ok to do fill_pot() in distributed fashion?
     if (portions == H) send (wakeup, dummy); # pot is full, wakeup bear
         else  send (deposit_perm, portions);  # let next bee deposit honey
}   }
```

```
process bear () {
   send (deposit_perm, 0);   # let first bee deposit honey
   while (true) {
     receive (wakeup, dummy);
     eat_honey ();
     send (deposit_perm, 0);   # reset portions to 0
}      }
```

How to modify to do fill_pot() in parallel??

Rendezvous

```
module Control_Pot
op into_pot(), deposit_pot(),
      sleep(), empty_pot();
body
  process Pot {
  int portions = 0, deposits=0;
  while (true)
    in   into_pot () and portions+deposits < MAXSIZE
            → deposits++;
    [ ]  deposit_done()
            → deposits--; portions++ ;
    [ ]  sleep () and portions == MAXSIZE
            → ;
    [ ]  empty_pot ()
            → portions=0;
    ni
  }
end Control_Pot
```

```
process bee [i=1 to N] {
  while (true) {
      collect_honey();
      call Control_pot.into_pot();
      deposit_pot();
      call Control_pot.deposit_done();
  }
}
```

```
process bear() {
  while (true) {
    call Control_pot.sleep();
    eat_honey();
    call Control_pot.empty_pot()
  }
}
```

Discuss

RPC Server Solution

- Distributed system over LAN?

```
process bee [i=1 to N] {
      while (true) {
         collect_honey();
         call Remote_pot.get_perm();
            deposit_honey();
         call Remote_pot.deposit_done();
         }
}
```

```
process bear {
      while (true) {
          call Remote_pot.sleep();
              eat_honey();
          call Remote_pot.empty_pot();
      }
   }
```

RPC Server Solution

```
module Remote_pot
    op get_perm(),
         deposit_done(),
         sleep(),
         empty_pot();
body
    int portions;
    sem mutex=1
            pot_full=0
            pot = M;
```

```
proc  into_pot() {
   P(pot);
}
```

```
proc  sleep () {
  P(pot_full);
}
```

```
proc empty_pot() {
  portions=0;
  V(mutex);
  for (i=1 to M) V(pot)
}
```

```
proc  deposit_done() {
    P(mutex);
    portions++;
    if (portions==M)
         V(pot_full)    # bear can eat
    else
         V(mutex);
}
```

Evaluate Your Solution

- Same problem – many solutions – all correct?
- Does it work correctly?
- Does it allow for maximum parallelism?
- Is this optimal solution?
- Is this solution good for current problem/environment?
  - 25 000 - 250 000 000 bees,
    collecting honey takes 30-60 min,
    depositing one portion in pot takes 1-3 mins,
    10000-100000 portions fill the pot,
    and bear eats the honey in pot in 5-50 minutes?
  - You might get another bear next year? What if much more bees?
  - What if the pot allows for 100-1000 simultaneous fill-ups?
  - Bees and bear are threads in Java application in 4-processor system running Linux?
  - “Honey” is an 80-byte msg to be used by “bear”?

Summary

- Specify first your requirements
- What concurrency tools do you have at your disposal?
- Does your solution match your environment?
- Will some known solution pattern apply here?
  - Readers-writers, producers-consumers, bakery?
- Does it work?
- Is it optimal in time/space?
- Does it allow for maximum parallelism?
- Does it minimize waiting?