Lesson 11

Practical Examples

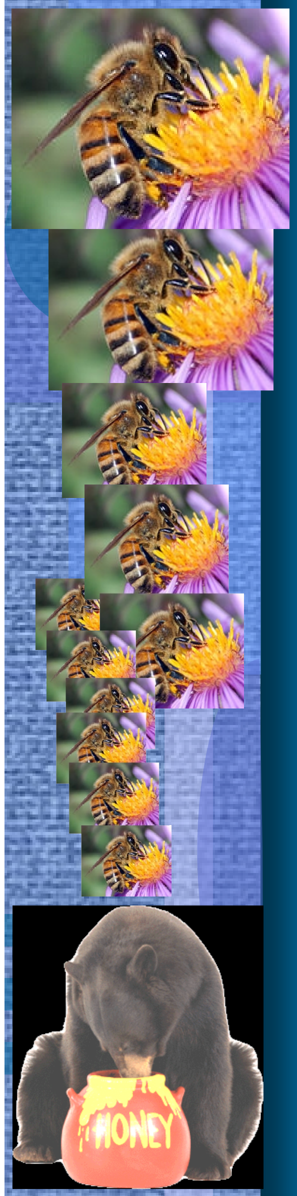
(Ch 5-9 [BenA 06])

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 <u>permission to fill</u> the pot, actually <u>filling</u> the pot, and possibly <u>signalling</u> 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 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
}   }
```

Discussion A

Semaphore Solution

```
process bee[i=1 to N] {
      while (true) {
         collect_honey();
         into_pot();  # deposit one honey portion into the honey pot
         }
}
```

```
process bear {
      while (true) {
         sleep();          # wait until the pot is full
         empty_pot();   # eat all the honey
         }
}
```


[Liisa Marttinen]

Semaphore Solution (contd)

```
sem mutex = 1,       # mutual exlusion
    pot_full = 0;    #  is the pot full of honey?
int portions;        # portions in the pot
```

```
procedure into_pot() { # bee deposits one honey portion
  P (mutex);
  fill_pot(); portions++;
  if (portions == H)  V (pot_full);  # let bear eat honey, pass baton
  else  V (mutex);   # let other bees fill the pot
}
```

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

```
procedure empty_pot() {   # bear eats all honey from the pot
   eat_all_honey ();
   portions=0;
   V (mutex); # let bees fill the pot again
}
```

Semaphore Solution (combined)

```
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
}        }
```

Monitor Solution

- 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();
   }
}
```

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


[Auvo Häkkinen]

Monitor Solution (contd)

```
monitor pot {
   int portions=0;    cond pot_full, pot_empty;

   procedure into_pot () {
      while (portions == H) waitC (pot_empty);
      portions++;   fill_pot();     # deposit honey in pot
      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 waiting bees
      }
}
```

Why "while" and not "if"? Would "if" work?

Why "if" and not "while"? Would "while" work?

What if some other type (not IRR) of signalling semantics?

All Work Included in Monitor

```
monitor pot {
   int portions=0;
   cond pot_full, pot_empty;

   procedure collect_into_pot() {
      collect_honey();
      while (portions==H) waitC(pot_empty);
      portions=portions+1; fill_pot();
      if (portions==H) signalC(pot_full);
      }

   procedure sleep_and_eat() {
      if (portions < H) waitC(pot_full);
      eat_honey();
      portions=0;
      signal_allC(pot_empty)
      }
}
```

```
process bee [i=1 to N] {
   while (true)
        pot.collect_into_pot();
}
```

```
process bear() {
   while (true)
       pot.sleep_and_eat();
}
```

Which is better?

What is the problem?

Better Monitor Solution ?

- Use monitor only for mutex and synchronization
  - Do fill_pot and all other real work outside monitor?

```
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();
   }
}
```

Another Monitor Solution (only synchronization, 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();
}   }
```


Discussion C

Monitor Solution (only sync, one bee fills at a time)

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

  procedure fill_perm () {
    while (fill+portions == H) wait (pot_empty);
    fill++;
    if (bee_filling) wait (fill_turn);
    bee_filling = true;
    }
  procedure fill_done () {
    fill--; portions++; bee_filling = false;
    if (portions == H) signal (pot_full);
    else signal (fill_turn);
    }
  procedure wait_full () {
    if (portions < H) wait (pot_full);
    }
  procedure empty_pot () {
    portions = 0;
    signal_all (pot_empty);  # wake up all
}   }
```

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

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

Simpler monitor solution
only sync, one bee fills at a time

```
monitor pot {   # no IRR
  int portions=0; cond pot_full, fill_turn;
  boolean bee_filling=false;

  procedure fill_perm () {
    while (portions == H or bee_filling)
          wait (fill_turn);
    portions++;
    bee_filling = true;
    }
  procedure fill_done () {
    bee_filling = false;
    if (portions == H) signal (pot_full);
    else signal (fill_turn);
    }
  procedure wait_full () {
    if (portions < H) wait (pot_full);
    }
  procedure empty_pot () {
    portions = 0;
    signal (fill_turn);  # wake up one
    }
}
```

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

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

ADA Protected Object Solution

```
..
private   portions := 0
…
protected body pot  is
   entry into_pot when portions < H is
      begin
           portions=portions+1; fill_pot()
      end deposit_into_pot;

  entry wait_full  when portions == H is
      begin     # empty body
      end wait_full;

  procedure empty_pot is
      begin
          portions = 0;
     end empty_pot;
end pot;
```

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

```
process bear() {
   while (true) {
     pot.wait_full;
     eat_honey();
     pot.empty_pot;
   }
}      # not Ada syntax
```

What if lots of bees (>> H) waiting in *into_pot*?

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

Channel Solution

- Processes communicate via <u>messages to/from</u> <u>channels</u>
  - 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 <u>communication</u>
    - 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
```

Channel Solution

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

Is it ok to do fill_pot() in distributed fashion?

```
process bear () {
  send (deposit_perm, 0);   # let first bee deposit honey

  while (true) {
    receive (wakeup, dummy);
    eat_honey ();
    send (deposit_perm, 0);   # let next bee deposit honey
    }
}
```

Message Solution

- Processes communicate via messages to/from processes
- Bear wakes up with wake-up message to it
  - Easy, just one bear
- Messages used only for synchronization or also for communication?
- How to keep track of honey portions
  - Must use messages
- How to send messages to other bees?
  - Too many receivers, not practical
  - Need msg server

Server Solution

- All synchronization problems solved by server
- Server process *pot* gives turns to bees and bear
- Correct bee must get permission to fill up pot
- Centralized solution, like monitor…

```
chan  pot_req (int id),             # request from a bee
      pot_perm [i =1 to n] (),      # permission for each bee
      bear_wakeup (),               # permission to eat for the bear
      bear_done ();                 # bear finished eating
```

Server Solution

```
chan  pot_req (int id),
   pot_perm [i =1 to n] (),
   bear_wakeup (), bear_done ();
```

```
process bee [i=1 to N] () {
     while (true) {
          collect_honey();
          send pot_req(i); # request to deposit
          receive pot_perm[i](); # deposit done
          fill_pot();  bee needs to do it!
     }
}
```

Who actually deposits honey in pot? When?

```
process pot () {     # server
     int  id,   # bee id
     portions=0;  # portions in the pot
     while (true) {
        receive pot_req(id);
        portions++;
        send pot_perm[id]();
        if (portions == H) { # pot is full
             send bear_wakeup();
             receive bear_done();
             portions=0;
             }
}    }
```

Error - "will be" Honey might not be there yet!

```
process bear () {
     while(true) {
          receive bear_wakeup();
          empty_honeypot();
          send bear_done();
          }
}
```

Server Solution Comments

- Who actually deposits honey in pot and when?
- How to separate *permission to deposit honey* <u>before</u> honey is deposited and *waking up the bear* <u>after</u> honey is deposited?

```
chan  pot_req (int id),                  # request from a bee
      pot_perm [i =1 to n] (),           # permission to deposit
      bee_honey_in_pot (),              # deposit done
      bee_can_proceed [i =1 to n] (),   # bee can start
                                         # collecting again
      bear_wakeup (),                    # permission to eat for bear
      bear_done ();                      # bear finished eating
```

- What if just <u>one</u> request channel and multiple reply channels?
  – Request for turn to deposit, turn to collect, turn to eat, turn to sleep?
  – Replies to bees and bear

Correct Server Solution

- ????

Rendezvous Solution

- Solution with Rendezvous-server Control_pot
  - Accepts all synchronization requests

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

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

Rendezvous Solution (contd)

```
module Control_Pot
  op into_pot(), deposit_pot(), sleep(), empty_pot();  # services
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 () and portions == MAXSIZE → portions=0;
    ni
  }
end Control_pot
```

Is this part needed?

- Solution with Ada similarly

Discussion D

RPC Server Solution (2)

- Distributed system over LAN?

```
process bee [i=1 to N] {
    while (true) {
        collect_honey();
        call Remote_pot.into_pot();
            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 (contd)

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

```
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);
}
```

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

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?