Concurrency

Programmers are scared of concurrency, and rightly so. Managers are scared of concurrency even more. If programmers were electricians, parallel programmers would be bomb disposal experts. Both cut wires. Except that, when the latter cut the wrong wire, mayhem ensues. We make mistakes in coding and we have systems for debugging and testing programs — there is no such system for concurrency bugs. I should know because I worked for a company that made state of the art data race detector. But this detector couldn't prove that a program was 100% data-race free. There was always a chance that a data race could sneak into a released product and strike at the most inconvenient moment. The company's web site had a list of major disasters caused by data races. Among them was the famous Therac-25 disaster that killed three patients and injured several others, and the Northeast Blackout of 2003 that affected 55 million people. The fears are well founded!

Bartosz Milewski

Concurrency vs. Parallelism

- Designing parallel systems is very hard
  - But sometimes necessary
  - Designing concurrent systems is surprisingly easy
- Concurrent systems can transparently utilize an underlying parallel system
- If your program is concurrent, and with sufficient concurrent operations, you don’t have to design for a specific number of processors

Why multi-programming

- Three basic reasons
  - Parallelism
  - Natural concurrency
  - Verification

Parallelism

- News flash: Moore's law is dead
  - Since October 2004
- This means that increased performance must come from using more processors
- How much of a Core i7 processor is actual processing power?
- Modern processors are all multi-core

Concurrency

- All (most) new CPUs are hardware threaded
  - Why
- The basic assumption behind hardware threading is that your application is multithreaded
  - Otherwise there is little advantage….

Natural Concurrency

- When designing computing systems we often try to force a non-existing sequential structure onto a problem that is in its nature
  - Concurrent
  - Asynchronous

Natural Concurrency

- Example: Network server
- If a network server is servicing many clients
  - How to figure out which is active?
  - How to prioritize connections?

Network server: In olden times

```
listen(socket)
for i in clients:
  socket[i]=accept(socket)

for (i=0; i< num_clients; i++)
   FD_SET(sockets[i], &recvfds);

select(clients+1, recvfds, NULL, NULL, NULL)

j=0
for (i=0; i< num_clients; i++)
   if FD_ISSET(recvfds, i){
      add_to_active(i) //ok we will cheat here and use blackbox code
      j++
   }

my_qsort(active_set)//And some more blackbox code her

for (i=0; i<j; i++)
service(active_set[i].socket)
```

Network server: Now

```
listen(socket,1)
while (1){
    client = accept(socket)
    thread(priority, service, client)
}
```

And here the number of clients is not fixed beforehand!

Verification

- A powerful feature in concurrent design is composition
- If you design very small functionalities in each process you can verify their correctness
- If you combine such small processes in a communicating network you can verify that that network is correct
- You then have a new (larger) process you know to be correct and can be used to build larger components again

Motivating Example

- Most hardware is build using concurrency designs because:
  - Parallelism – all the hardware exist at all times
  - Concurrency – you cannot determine the order and timing of all events
  - Verification – you cannot distribute a patch to existing hardware
- The Pentium processor and Windows 95 were siblings
  - Both with app 15MLOC
- We found 5 errors in the Pentium……

Java Threading

<java.sun.com/products/jfc/tsc/articles/threads/threads1.html>

- ***"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."***
- ***"Component developers do not have to have an in-depth understanding of threads programming: toolkits in which all components must fully support multithreaded access, can be difficult to extend, particularly for developers who are not expert at threads programming."***

Java Threading

`<java.sun.com/products/jfc/tsc/articles/threads/threads1.html>`

- ***“It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications … use of threads can be very deceptive … in almost all cases they make debugging, testing, and maintenance vastly more difficult and sometimes impossible. Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism … this is particularly true in Java … we urge you to think twice about using threads in cases where they are not absolutely necessary …”***

Ultra SPARC T3

- Oracle’s new CPU
  - Designed for Java
- 16 processor cores on a CPU
- 8 HW threads on a core
- So while threading should be avoided -according to Sun- you do need at least 16
  - actually 128 to make the thing perform
- We are going to aim at millions – just to be sure we are ready for the next generation CPU

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

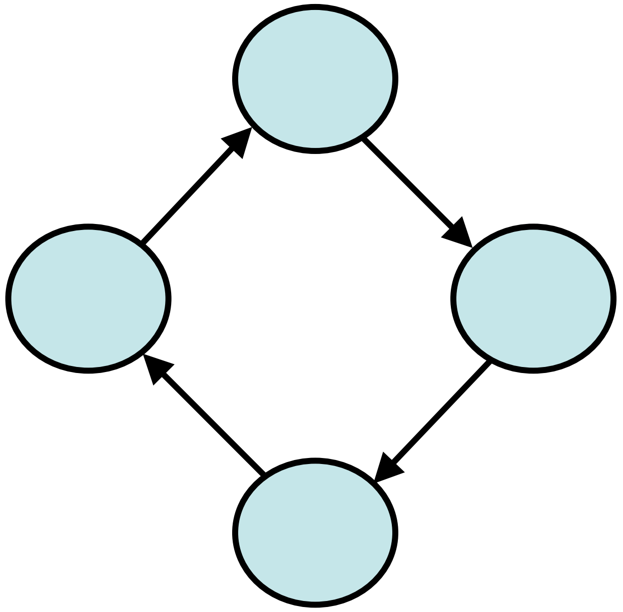
Deadlocks

- You would think that deadlocks could easily occur in multiprocessing systems
  - And they sure can…
- But we are only required to eliminate one of four conditions to break the chance of deadlock

Basic problem

Every process does:

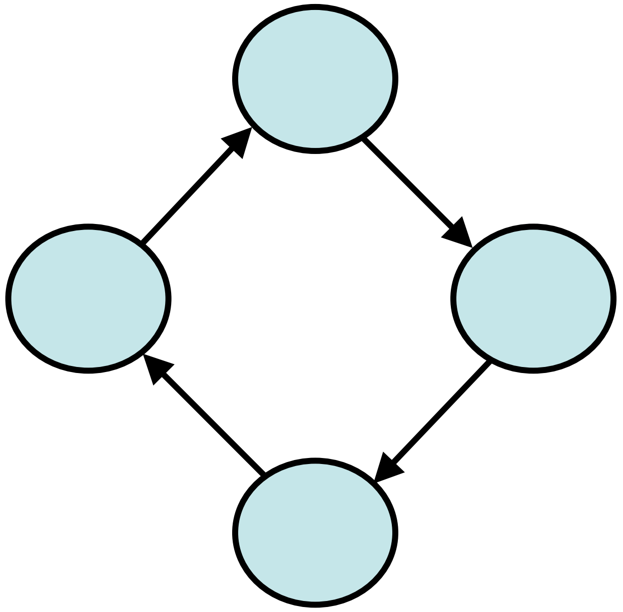
Send(a)
Recv(b)



Basic problem

Every process does:

Send(a)
Recv(b)

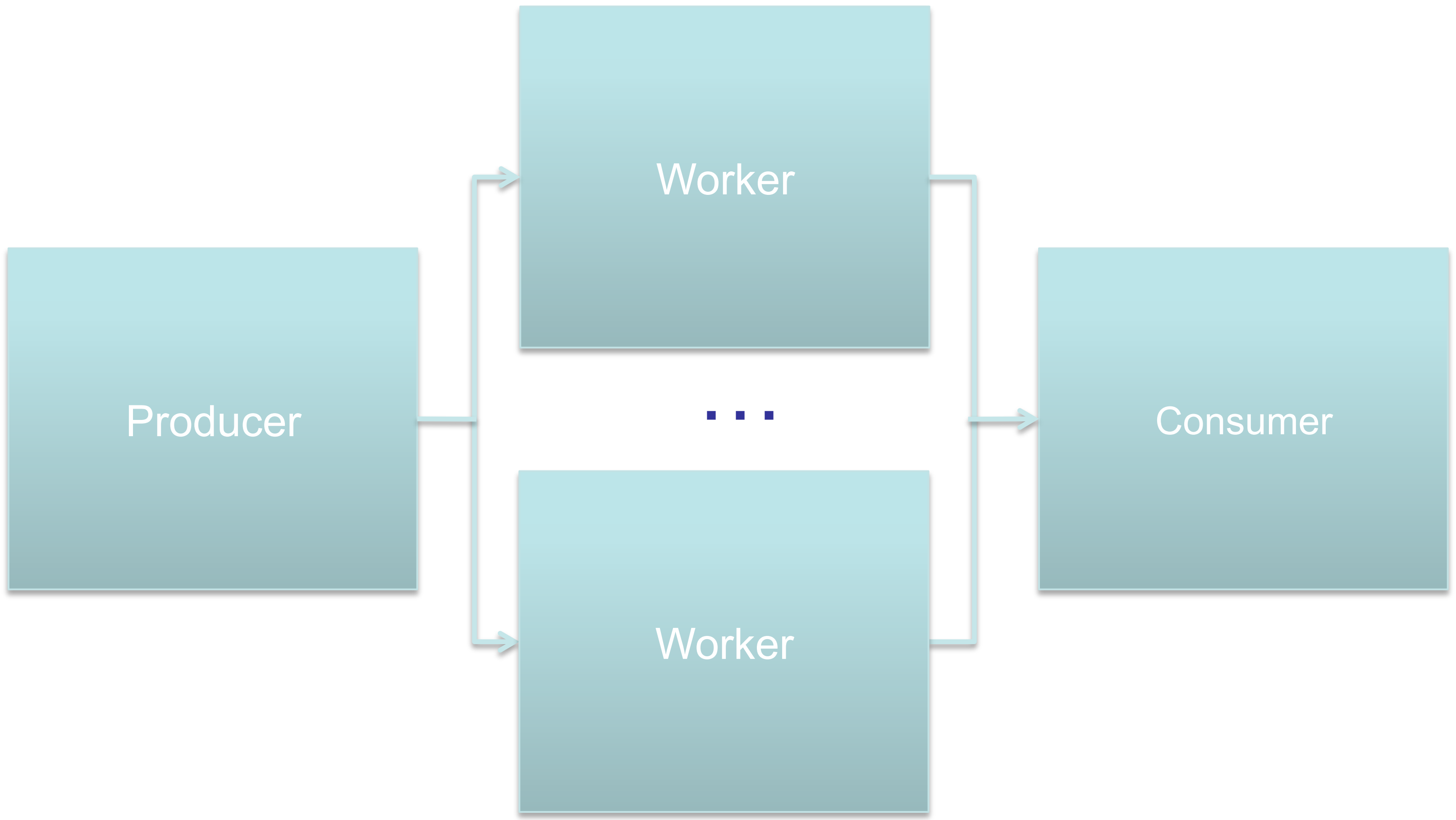


Solution: Allow send and receive at the same time!

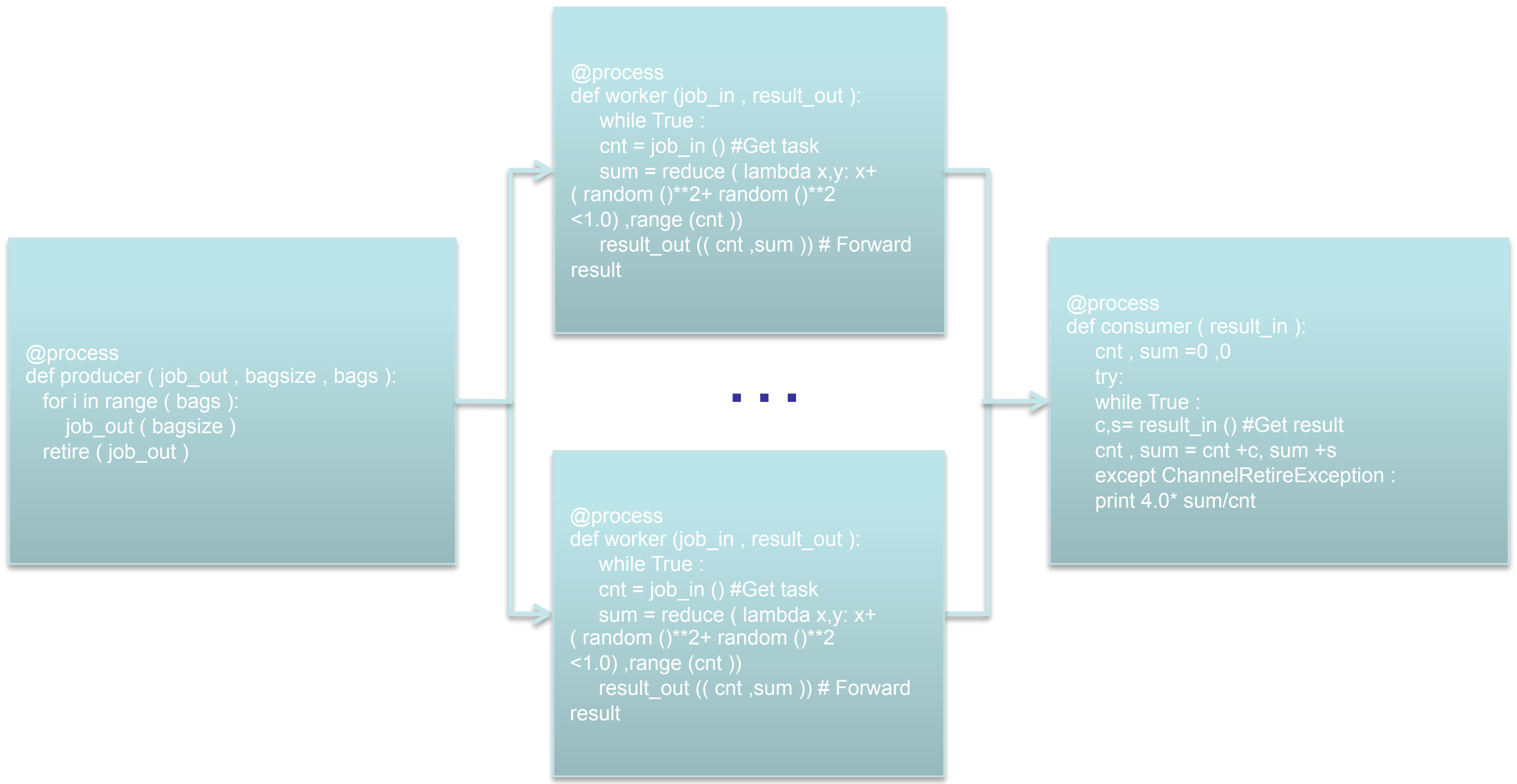
Race-conditions

- Race conditions are associated with shared data in a multiprocessing context
- If a piece of data is
  - Accessed by more than one process
  - Not exclusively read
- Then we need to protect that data against parallel access
  - It only takes one unprotected write to destroy the system

Monte Carlo Pi



Monte Carlo Pi



```
@process
def producer ( job_out , bagsize , bags ):
  for i in range ( bags ):
     job_out ( bagsize )
  retire ( job_out )
```

```
@process
def worker (job_in , result_out ):
    while True :
    cnt = job_in () #Get task
    sum = reduce ( lambda x,y: x+
( random ()**2+ random ()**2
<1.0) ,range (cnt ))
    result_out (( cnt ,sum )) # Forward
result
```

```
@process
def worker (job_in , result_out ):
    while True :
    cnt = job_in () #Get task
    sum = reduce ( lambda x,y: x+
( random ()**2+ random ()**2
<1.0) ,range (cnt ))
    result_out (( cnt ,sum )) # Forward
result
```

```
@process
def consumer ( result_in ):
    cnt , sum =0 ,0
    try:
    while True :
    c,s= result_in () #Get result
    cnt , sum = cnt +c, sum +s
    except ChannelRetireException :
    print 4.0* sum/cnt
```

An example…

```
from pycsp import *
from random import random
@process
def producer ( job_out , bagsize , bags ):
    for i in range ( bags ): job_out ( bagsize )
    retire ( job_out )
@process
def worker (job_in , result_out ):
    while True :
    cnt = job_in () #Get task
    sum = reduce ( lambda x,y: x+( random ()**2+ random ()**2 <1.0) ,range (cnt ))
    result_out (( cnt ,sum )) # Forward result
@process
def consumer ( result_in ):
    cnt , sum =0 ,0
    try:
    while True :
    c,s= result_in () #Get result
    cnt , sum = cnt +c, sum +s
    except ChannelRetireException :
    print 4.0* sum/cnt #We are done - print result

jobs = Channel ()
results = Channel ()
Parallel ( producer ( jobs.writer () ,1000 , 10000) ,
[ worker ( jobs.reader (), results.writer ()) for i in range (10)] ,
consumer ( results.reader ()))
```