Concurrency

Concurrency vs. Parallelism

- The less wise often use concurrent and parallel interchangeably
- They are not the same and should not be treated as such
- I will seek maximum penalty of anybody misusing the terms in their reports ☺

Concurrency

- Concurrently mean that two or more tasks can take place
  - At the same time or
  - One before the other, but we don’t know in which order
- Two individual phones are natural concurrent
  - They may receive a call at the same time
    - But I’d like to see that happen!!!

Parallelism

- Parallel mean that two or more operations run at the same time
- They have to run at the same time!

- Imagine an assembly-line
  - All workers work in parallel
  - What happens if a worker takes a break?

Concurrency vs. Parallelism

- Designing parallel systems is very hard
  - But sometimes necessary (check out the cluster computing class next quarter)
  - 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 Pentium 4 processor is actual processing power?
- Next generation 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: Interrupts
- How many programming models have a natural model for handling interrupts?
  - Were interrupts hard to handle in the OS class?
- In a concurrent design model we simply have a thread that is waiting for the interrupt to occur
  - In fact it will see the interrupt at an arriving message

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

```
for i in clients:
    listen(socket)
    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

```
while (1){
    listen(socket,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……

Multiprocessing

- Writing multiprocess systems is probably the easiest thing
- Any (well most) operating systems are basic multiprocessing systems

Multithreading

- multithreading is similar to multiprocessing but with at few ”added features”
  - Thread shifting is faster then process shifting
  - Threads can easily share memory
    - Including stack data

Why ”added features”

- Because threading is not necessarily better than processing
  - While processes die individually faulting threads most often kills the other threads too
  - It is very easy to share data unintentionally

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 …"***

Niagra

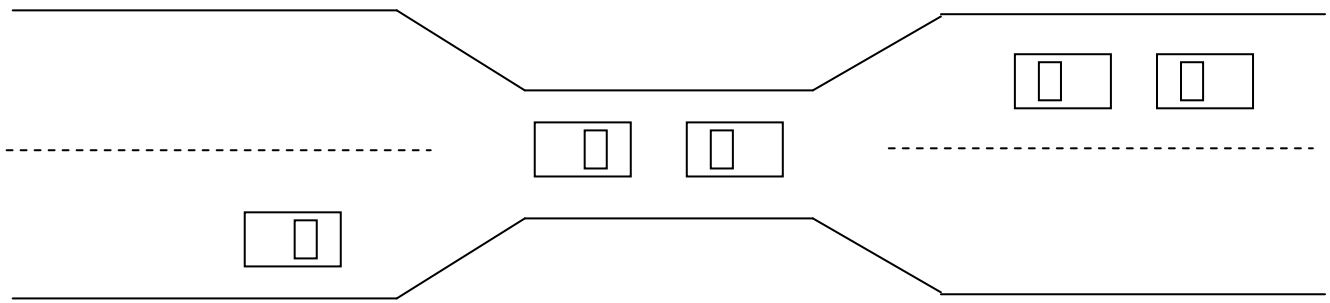
- Suns new CPU
  - Designed for Java
- 8 processor cores on a CPU
- 32 HW threads on a CPU
- So while threading should be avoided - according to Sun- you do need at least 8 and probably 32 to make the thing perform
- We are going to aim at millions – just to be sure we are ready for the next generation CPU

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Mexican standoff
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

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.

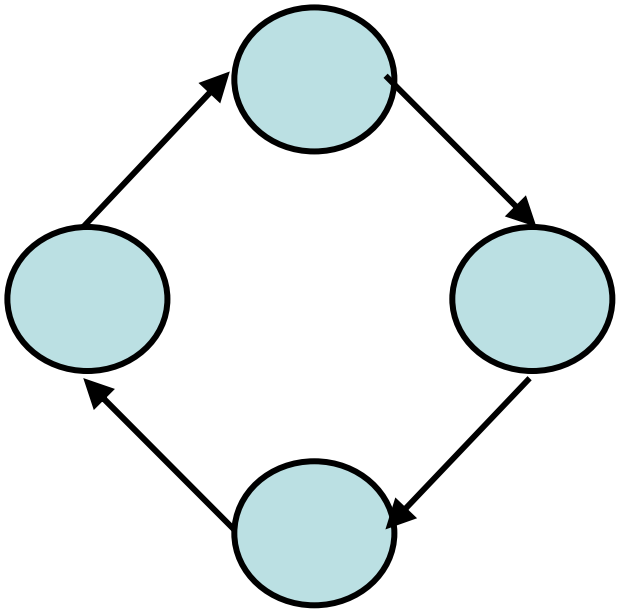
Deadlocks

- 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:

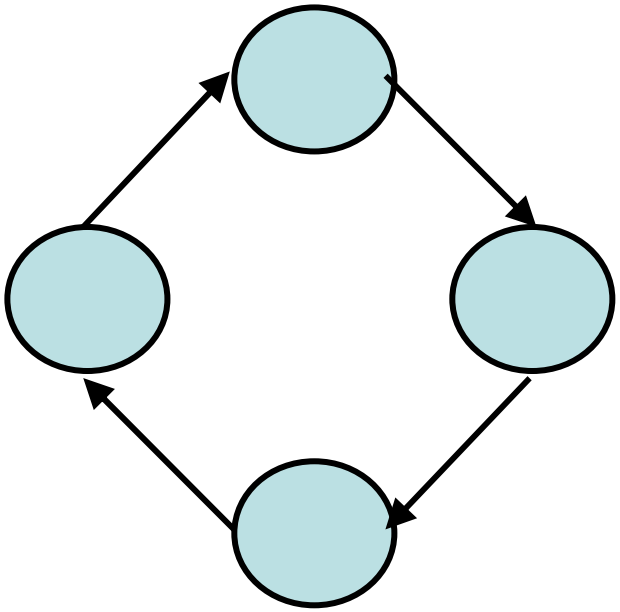
Send(a)
Recv(b)



Basic problem

Every process does:

Send(a)
Recv(b)



Solution: Allow send and receive at the same time!

Live-locks

- Similar to deadlocks but much harder to detect
- The system appears to be active
- But due to cyclic waiting no progress is made

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
- The we need to protect that data against parallel access
  - It only takes one unprotected write to destroy the system

Race-conditions

- Example:

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

- How many executions before we see this?

Fairness

- For some applications it is essential that all processes are allowed to make progress
  - Games for instance
- So if we have a system with many components we have to avoid starvation
  - This is easily done in a multiprocessing environment

Scheduling

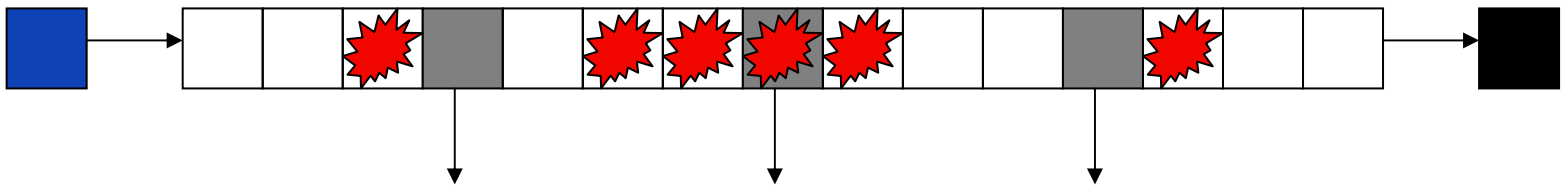
- In other context fairness is less interesting and urgency is more interesting
  - Robotics
- So if we have a system with many components where some are more important than others we need to ensure the the important ones are chosen
  - We do that by introducing priorities

Blood Clot Modeling

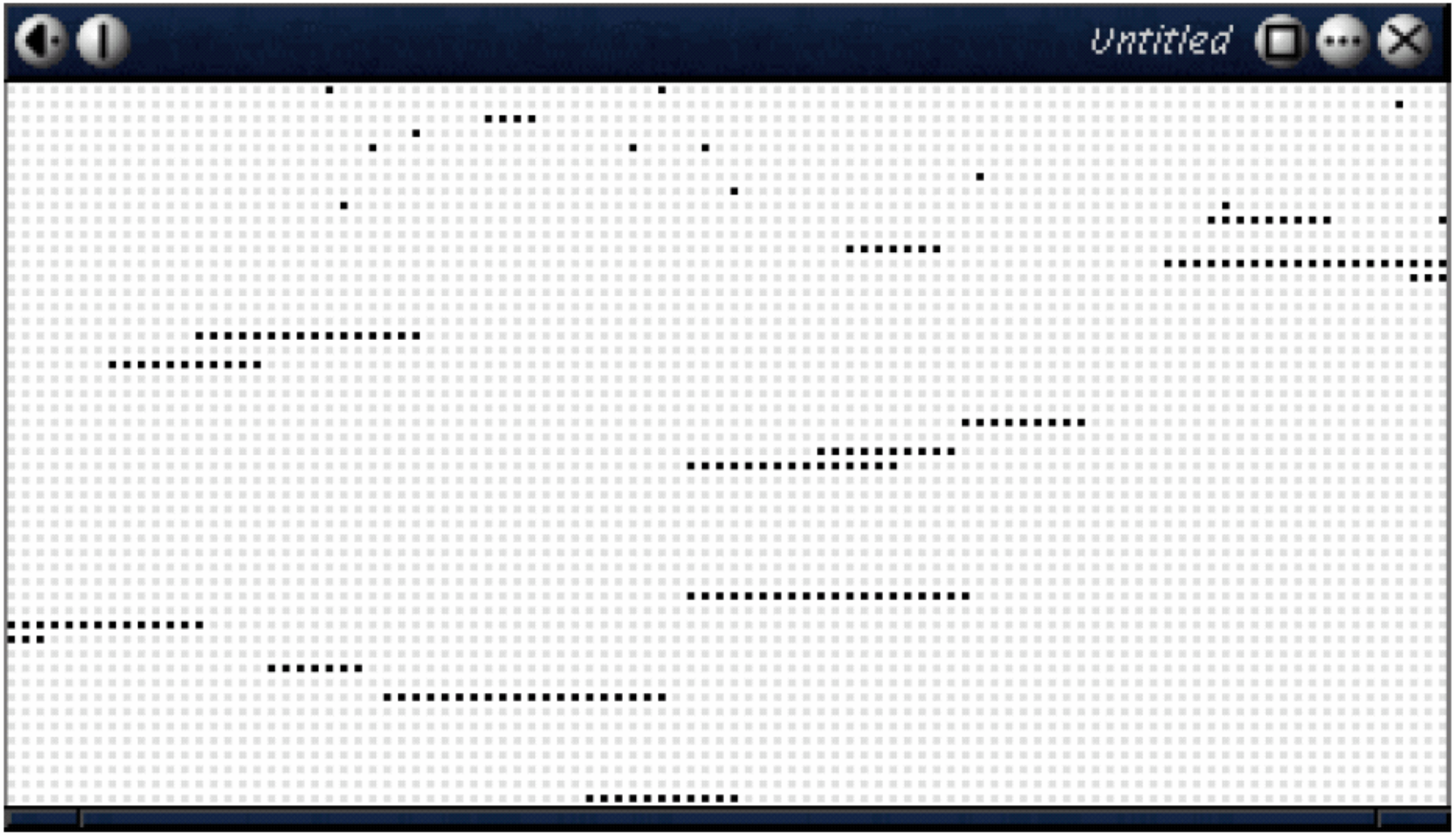
- Overall research project seeks to develop nanites for artificial blood clot generation
  - But first the nature of blood clots must be modeled mathematically
- The experiments presented here are **extremely** early results
  - With one-dimensional models of blood-clots only

The simulation

- The model is written entirely in Occam
- In this model every discrete cell in the channel is a process in it's own right
- Blood clots are entered at random
  - With a varying probability
- Blood clots move at random
  - If they meet they merge to one clot
  - Larger clots move slower



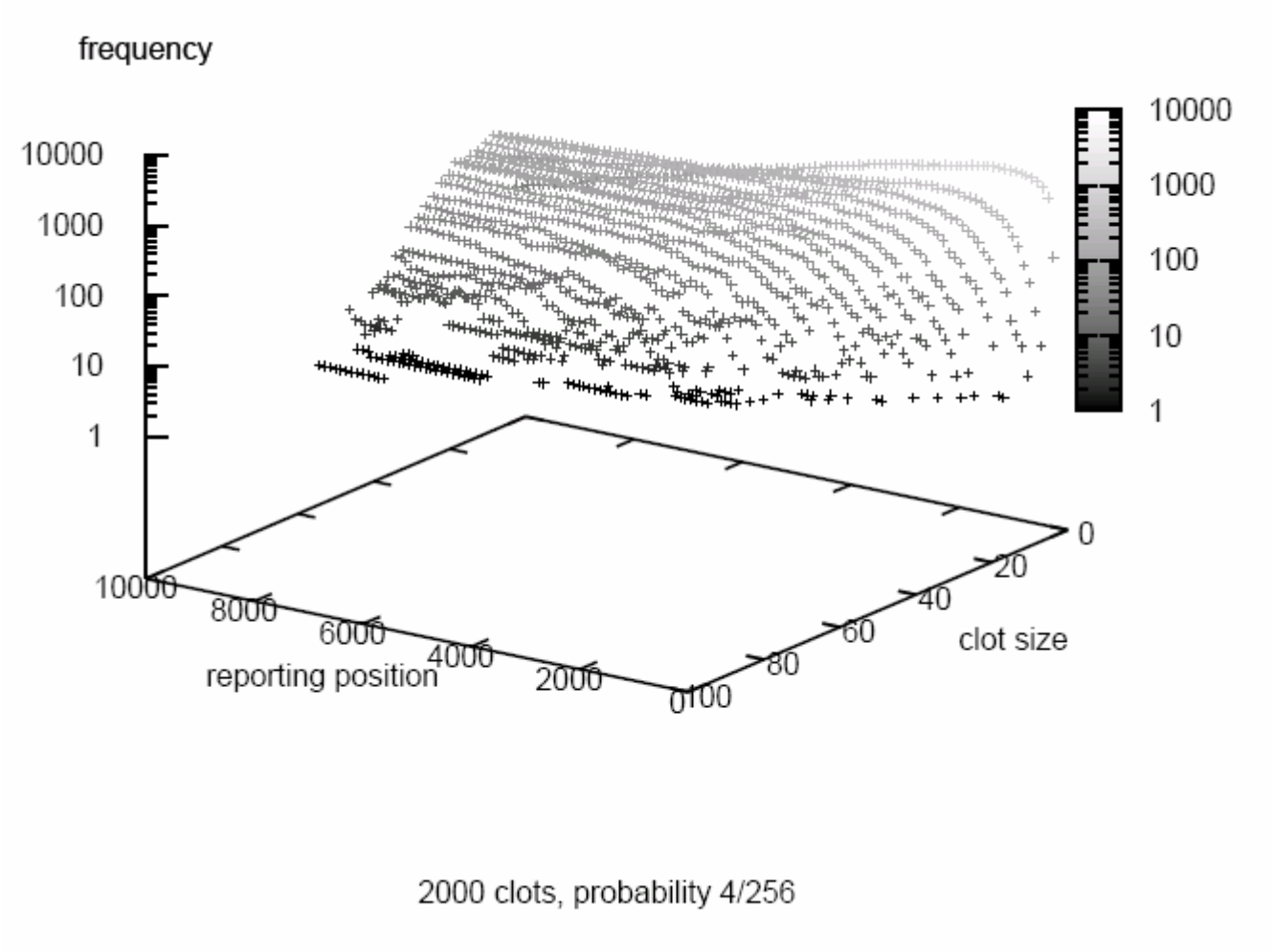
Clot modeling



The experiment

- Probability of blood clots is varied from 1/256 to 255/256 in each time-step
- Each probability is run 10 times to produce an average
- Runtime is 1-3 min per experiment
  - So Grid jobs are all 10 runs at one single probability
- A total of 254 jobs at an average at 20 min
  - Only 3.5 CPU days
- Actual runtime was less than 1h

Results


frequency
10000
1000
100
10
1
10000
8000
6000
4000
2000
0
reporting position
0
20
40
60
80
100
clot size
10000
1000
100
10
1
2000 clots, probability 4/256

Blood Clot Results

Initial Experiences with occam-pi Simulations of Blood Clotting on the Minimum Intrusion Grid, *Peter H. Welch, Brian Vinter, and Frederick R. M. Barnes*, in the proc. Of Parallel and Distributed Programming Techniques and Applications 2005