15-210: Parallelism in the Real World

- Types of paralellism
- Parallel Thinking
- Nested Parallelism
- Examples (Cilk, OpenMP, Java Fork/Join)
- Concurrency

Cray-1 (1976): the world's most expensive love seat

Data Center: Hundred's of thousands of computers

Since 2005: Multicore computers

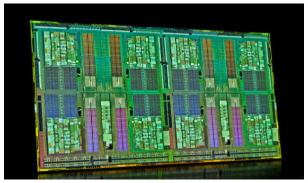
AMD Opteron (sixteen-core) Model 6274
by AMD
★★★★☆ (1 customer review)

List Price: ~~$693.00~~
Price: $599.99 ✓Prime
You Save: $93.01 (13%)

Only 1 left in stock (more on the way).
Ships from and sold by **Amazon.com**. Gift-wrap available.

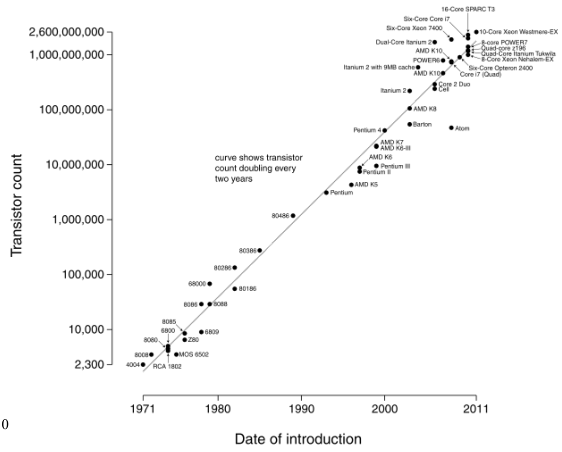
Want it delivered Monday, November 5? Order it in the next 14 hours and 37 minutes
Delivery may be impacted by Hurricane Sandy. Proceed to checkout to see estimated

43 new from $599.99

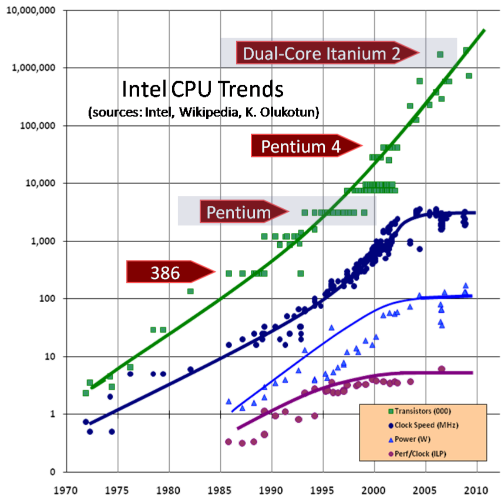


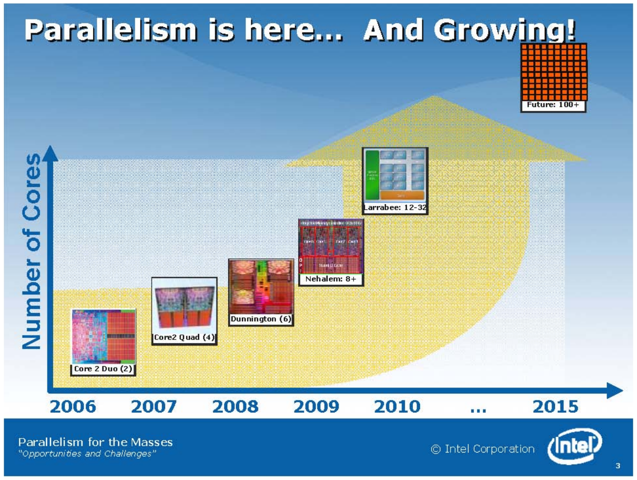
Moore's Law

Microprocessor Transistor Counts 1971-2011 & Moore's Law



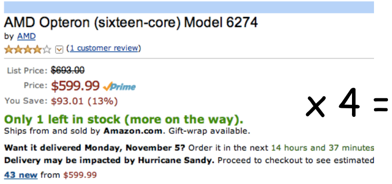
Moore's Law and Performance





Andrew Chien, 2008

64 core blade servers ($6K) (shared memory)



AMD Opteron (sixteen-core) Model 6274

by AMD

★★★★☆ (1 customer review)

List Price: ~~$693.00~~

Price: $599.99 ✓Prime

You Save: $93.01 (13%)

Only 1 left in stock (more on the way).

Ships from and sold by **Amazon.com**. Gift-wrap available.

Want it delivered Monday, November 5? Order it in the next 14 hours and 37 minutes

Delivery may be impacted by Hurricane Sandy. Proceed to checkout to see estimated

43 new from $599.99

x 4 =

1024 "cuda" cores

amazon.com

Hello. Sign in to get personalized recommendations. New customer? Start here.

Your Amazon.com | Today's Deals | Gifts & Wish Lists | Gift Cards

EVGA GeForce GTX 590 Classified 3DVI/Mini-Display Port SLI Ready Li 03G-P3-1596-AR

by EVGA

(16 customer reviews) | Like (29)

Price: $924.56

In Stock.

Ships from and sold by J-Electronics.

Only 1 left in stock--order soon.

5 new from $749.99 2 used from $695.00

Samsung Galaxy S IV to feature Exynos 28nm quad-core processor?

Written by Andre Yoskowitz @ 01 Nov 2012 18:02

It has been a few weeks but there is a new rumor regarding the upcoming Samsung Galaxy S IV.

According to reports, Samsung will pack next year's flagship device with its "Adonis" Exynos processor, a quad-core ARM 15 beast that uses efficient 28nm tech.

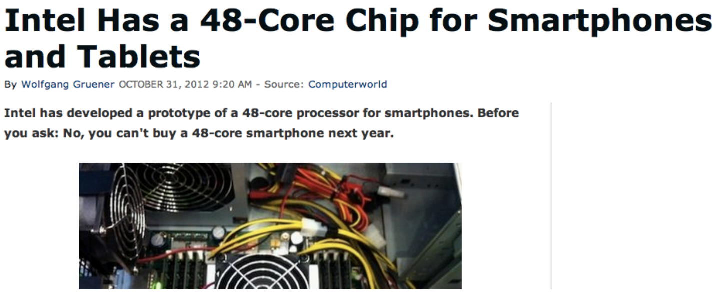
Samsung is supposedly still testing the application processor, but mass production is scheduled for the Q1 2013 barring any delays.

Circa November 2012

Samsung Galaxy S IV is now Official: Octa-Core CPU, 5" Full HD Display & 13MP Camera

Follow: Phones GT-I9500 Samsung Display Samsung Exynos Samsung Galaxy S IV Samsung Mobile Unpacked 2013

Samsung has just announced the Samsung Galaxy S4 at their Mobile Unpacked Event 2013 Episode 1 in New York, USA. The Galaxy S4 features a stunning 4.99" Full HD (1920×1080) SuperAMOED display. With a 441 ppi pixel density, your eyes won't be able to distinguish the pixels, which ensures excellent visual comfort. Even though the Galaxy S4 has a large display and a massive battery of 2,600 MAh, it's only 7.9mm thick. Samsung's latest flagship device is PACKED with powerful components, consisting of Samsung's latest Exynos 5 Octa-Core (5410) CPU based on ARM's big.LITTLE technology with Quad Cortex-



Intel Has a 48-Core Chip for Smartphones and Tablets

By Wolfgang Gruener OCTOBER 31, 2012 9:20 AM - Source: Computerworld

Intel has developed a prototype of a 48-core processor for smartphones. Before you ask: No, you can't buy a 48-core smartphone next year.

Parallel Hardware

Many forms of parallelism
- Supercomputers: large scale, shared memory
- Clusters and data centers: large-scale, distributed memory
- Multicores: tightly coupled, smaller scale
- GPUs, on chip vector units
- Instruction-level parallelism

Parallelism is important in the real world.

Key Challenge: Software (How to Write Parallel Code?)

At a high-level, it is a two step process:
- Design a work-efficient, low-span parallel algorithm
- Implement it on the target hardware

In reality: each system required different code because programming systems are immature
- Huge effort to generate efficient parallel code.
  - Example: Quicksort in MPI is 1700 lines of code, and about the same in CUDA
- Implement one parallel algorithm: a whole thesis.

15-210 Approach

Enable parallel thinking by raising abstraction level

I. Parallel thinking: Applicable to many machine models and programming languages

II. Reason about correctness and efficiency of algorithms and data structures.

Parallel Thinking

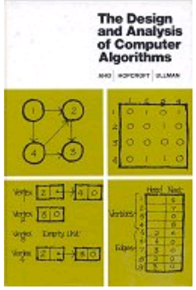
Recognizing true dependences: unteach sequential programming.

Parallel algorithm-design techniques
- Operations on aggregates: map/reduce/scan
- Divide & conquer, contraction
- Viewing computation as DAG (based on dependences)

Cost model based on work and span

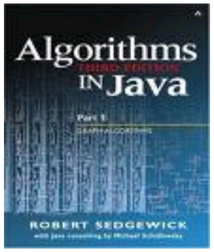
Quicksort from Aho-Hopcroft-Ullman (1974)

```
procedure QUICKSORT(S):
  if S contains at most one element then return S
  else
    begin
      choose an element a randomly from S;
      let $S_1$, $S_2$ and $S_3$ be the sequences of
          elements in S less than, equal to,
          and greater than a, respectively;
      return (QUICKSORT($S_1$) followed by $S_2$
        followed by QUICKSORT($S_3$))
end
```



Quicksort from Sedgewick (2003)

```
public void quickSort(int[] a, int left, int right) {
    int i = left-1;  int j = right;
    if (right <= left) return;
    while (true) {
      while (a[++i] < a[right]);
      while (a[right] < a[--j])
        if (j==left) break;
      if (i >= j) break;
      swap(a,i,j); }
    swap(a, i, right);
    quickSort(a, left, i - 1);
    quickSort(a, i+1, right); }
```



Styles of Parallel Programming

Data parallelism/Bulk Synchronous/SPMD
Nested parallelism : what we covered
Message passing
Futures (other pipelined parallelism)
General Concurrency

Nested Parallelism

Nested Parallelism =
arbitrary nesting of parallel loops + fork-join
- Assumes no synchronization among parallel tasks except at joint points.
- Deterministic if no race conditions

Advantages:
- Good schedulers are known
- Easy to understand, debug, and analyze
- Purely functional, or imperative…either works

Nested Parallelism: parallel loops

```
cilk_for (i=0; i < n; i++)
   B[i] = A[i]+1;
```

Cilk

```
Parallel.ForEach(A, x => x+1);
```

Microsoft TPL (C#,F#)

```
B = {x + 1 : x in A}
```

Nesl, Parallel Haskell

```
#pragma omp for
for (i=0; i < n; i++)
  B[i] = A[i] + 1;
```

OpenMP

Nested Parallelism: fork-join

```
cobegin {
  S1;
  S2;}
```

Dates back to the 60s. Used in dialects of Algol, Pascal

```
coinvoke(f1,f2)
Parallel.invoke(f1,f2)
```

Java fork-join framework
Microsoft TPL (C#,F#)

```
#pragma omp sections
{
  #pragma omp section
  S1;
  #pragma omp section
  S2;
}
```

OpenMP (C++, C, Fortran, …)

Nested Parallelism: fork-join

```
spawn S1;
S2;
sync;
```

cilk, cilk+

```
(exp1 || exp2)
```

Various functional languages

```
plet
  x = exp1
  y = exp2
in
  exp3
```

Various dialects of ML and Lisp

Cilk vs. what we've covered

Fork Join

ML: val (a,b) = par(fn () => f(x),
fn () => g(y))
Psuedocode: val (a,b) = (f(x) || g(y))
Cilk: cilk_spawn f(x);
g(y);
cilk_sync;

Parallel loops

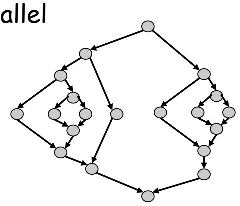
ML: S = tabulate f(i) n
Psuedocode: S = <f(i) : i in <0,..n-1>>
Cilk: cilk_for (int i = 0; i < n; i++)
S[i] = f(i)

Cilk vs. what we've covered

ML: S = tabulate f(i) n
Psuedocode: S = <f(i) : i in <0,..n-1>>
Cilk: cilk_for (int i = 0; i < n; i++)
    S[i] = f(i)

Serial Parallel DAGs

Dependence graphs of nested parallel computations are series parallel



Two tasks are parallel if not reachable from each other.

A data race occurs if two parallel tasks are involved in a race if they access the same location and at least one is a write.

Cost Model (General)

Compositional:

Work : total number of operations
- costs are added across parallel calls

Span : depth/critical path of the computation
- Maximum span is taken across forked calls

Parallelism = Work/Span
- Approximately # of processors that can be effectively used.

Combining costs (Nested Parallelism)

Combining for parallel for:

```
pfor (i=0; i<n; i++)
  f(i);
```

$$W_{pexp}(\text{pfor } ...) = \sum_{i=0}^{n-1} W_{exp}(\text{f(i)})$$ work

$$D_{pexp}(\text{pfor } ...) = \max_{i=0}^{n-1} D_{exp}(\text{f(i)})$$ span

Why Work and Span

Simple measures that give us a good sense of efficiency (work) and scalability (span).

Can schedule in O(W/P + D) time on P processors.

This is within a constant factor of optimal.

Goals in designing an algorithm

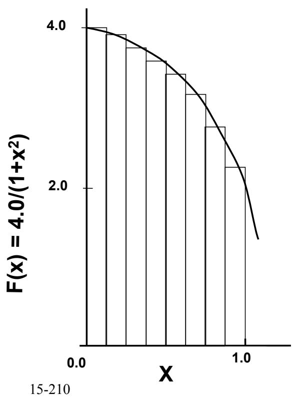
1. Work should be about the same as the sequential running time. When it matches asymptotically we say it is **work efficient.**
2. Parallelism (W/D) should be polynomial. $O(n^{1/2})$ is probably good enough

---

Example Cilk

```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-1);
    y = cilk_spawn fib(n-2);
    cilk_sync;
    return (x+y);
  }
}
```

Example OpenMP: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

The C code for Approximating PI

```
static long num_steps = 100000;
double step;
void main ()
{         int i;   double x, pi, sum = 0.0;

          step = 1.0/(double) num_steps;
           x = 0.5 * step;
          for (i=0;i<= num_steps; i++){
                   x+=step;
                   sum += 4.0/(1.0+x*x);
          }
          pi = step * sum;
}
```

The C/openMP code for Approx. PI

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;           double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
#pragma omp   parallel   for   private(i, x) reduction(+:sum)
     for (i=0;i<= num_steps; i++){
           x = (i+0.5)*step;
           sum = sum + 4.0/(1.0+x*x);
     }
     pi = step * sum;
}
```

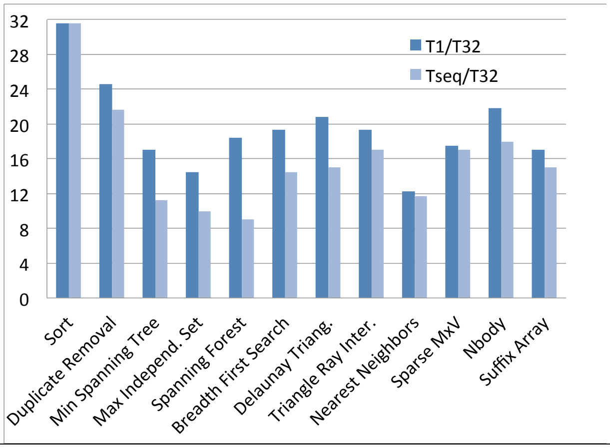
Private clause creates data local to a thread

Reduction used to manage dependencies

Example : Java Fork/Join

```
class Fib extends FJTask {
   volatile int result; // serves as arg and result
   int n;
   Fib(int _n) { n = _n; }

   public void run() {
      if (n <= 1) result = n;
      else if (n <= sequentialThreshold) number = seqFib(n);
      else {
         Fib f1 = new Fib(n – 1);
         Fib f2 = new Fib(n – 2);
         coInvoke(f1, f2);
         result = f1.result + f2.result;
      }
   }
}
```

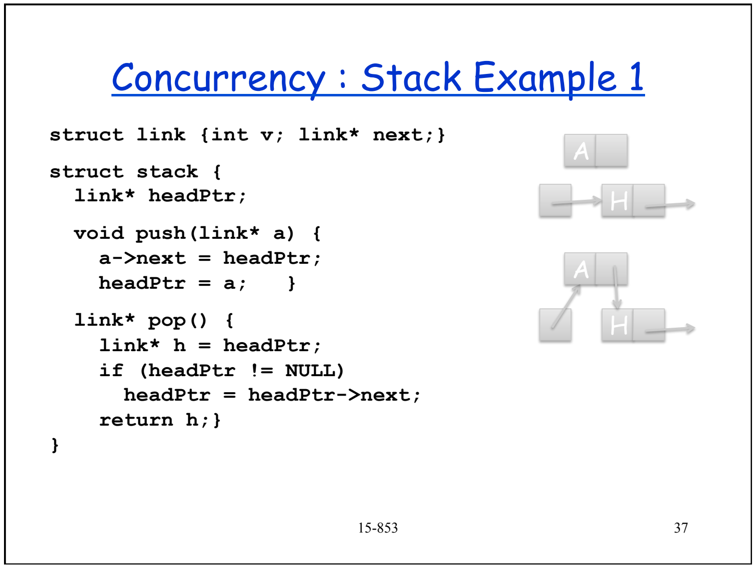
How do the problems do on a modern multicore



Parallelism vs. Concurrency

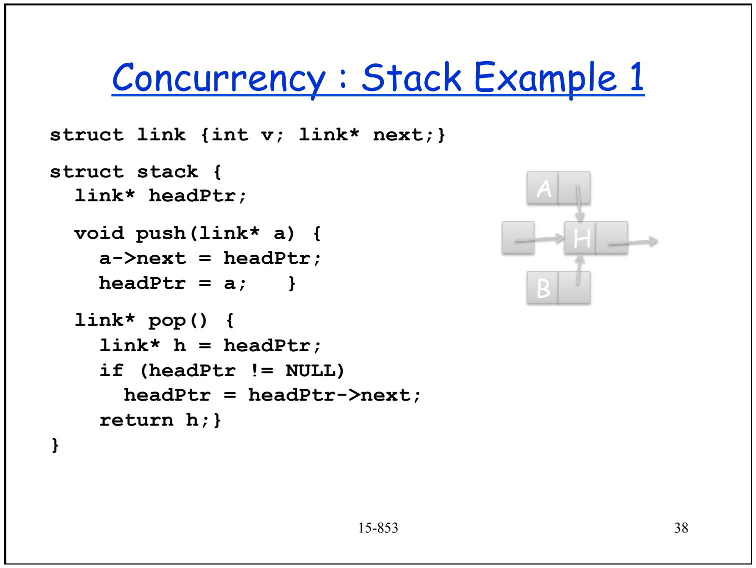
- Parallelism: using multiple processors/cores running at the same time. Property of the machine
- Concurrency: non-determinacy due to interleaving threads. Property of the application.

		Concurrency	
		sequential	concurrent
Parallelism	serial	Traditional programming	Traditional OS
	parallel	Deterministic parallelism	General parallelism



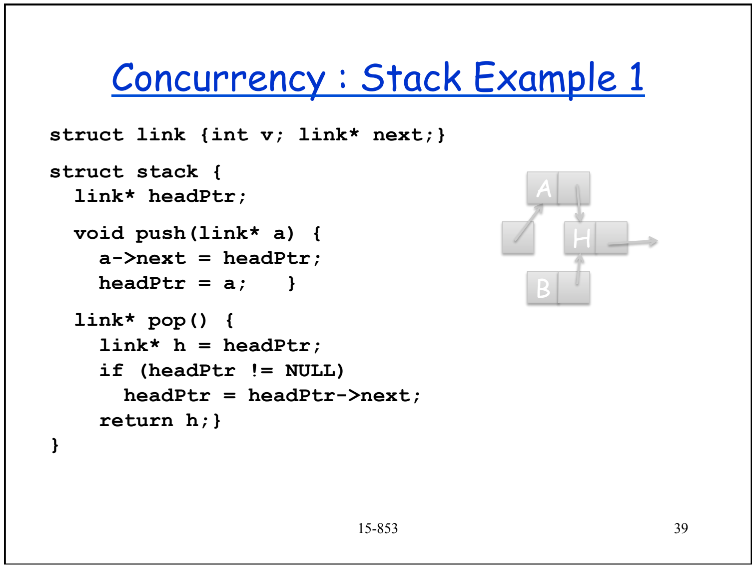
Concurrency : Stack Example 1

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    a->next = headPtr;
    headPtr = a;    }

  link* pop() {
    link* h = headPtr;
    if (headPtr != NULL)
      headPtr = headPtr->next;
    return h;}
}
```



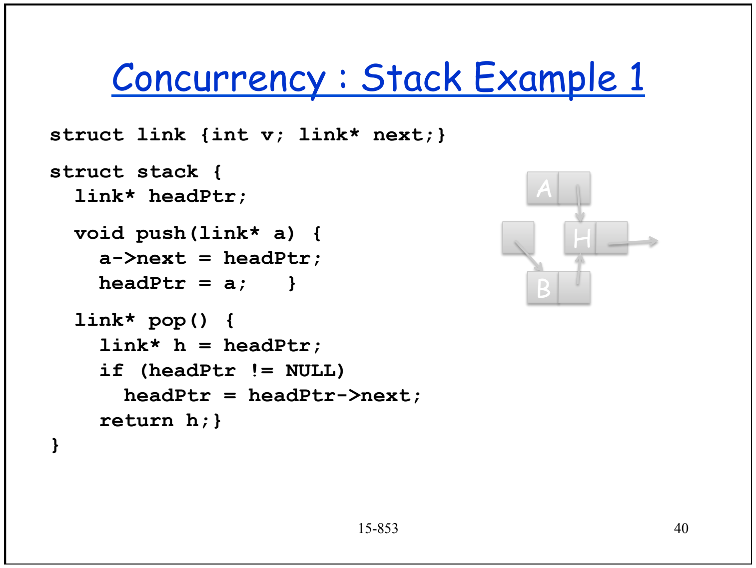
Concurrency : Stack Example 1

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    a->next = headPtr;
    headPtr = a;    }

  link* pop() {
    link* h = headPtr;
    if (headPtr != NULL)
      headPtr = headPtr->next;
    return h;}
}
```



Concurrency : Stack Example 1

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    a->next = headPtr;
    headPtr = a;    }

  link* pop() {
    link* h = headPtr;
    if (headPtr != NULL)
      headPtr = headPtr->next;
    return h;}
}
```

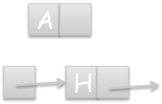


Concurrency : Stack Example 1

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    a->next = headPtr;
    headPtr = a;    }

  link* pop() {
    link* h = headPtr;
    if (headPtr != NULL)
      headPtr = headPtr->next;
    return h;}
}
```

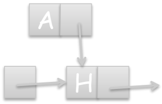
Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;
  void push(link* a) {
    do {
      link* h = headPtr;
      a->next = h;
    while (!CAS(&headPtr, h, a)); }
  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



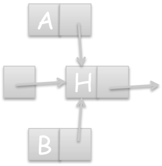
Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;
  void push(link* a) {
    do {
      link* h = headPtr;
⇨     a->next = h;
    while (!CAS(&headPtr, h, a)); }
  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



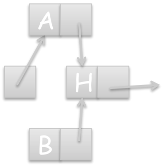
Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;
  void push(link* a) {
    do {
      link* h = headPtr;
⇨     a->next = h;
    while (!CAS(&headPtr, h, a)); }
  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;
  void push(link* a) {
    do {
      link* h = headPtr;
      a->next = h;
⇨   while (!CAS(&headPtr, h, a)); }
  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



Concurrency : Stack Example 2'

```
P1 : x = s.pop();  y = s.pop();  s.push(x);
P2 : z = s.pop();
```

Before:

After:

```
P2: h = headPtr;
P2: nxt = h->next;
P1: everything
P2: CAS(&headPtr,h,nxt)
```

The ABA problem

Can be fixed with counter and 2CAS, but…

Concurrency : Stack Example 3

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    atomic {
      a->next = headPtr;
      headPtr = a;   }}

  link* pop() {
    atomic {
      link* h = headPtr;
      if (headPtr != NULL)
        headPtr = headPtr->next;
      return h;}}
}
```

Concurrency : Stack Example 3'

```
void swapTop(stack s) {
  link* x = s.pop();
  link* y = s.pop();
  push(x);
  push(y);
}
```

Queues are trickier than stacks.