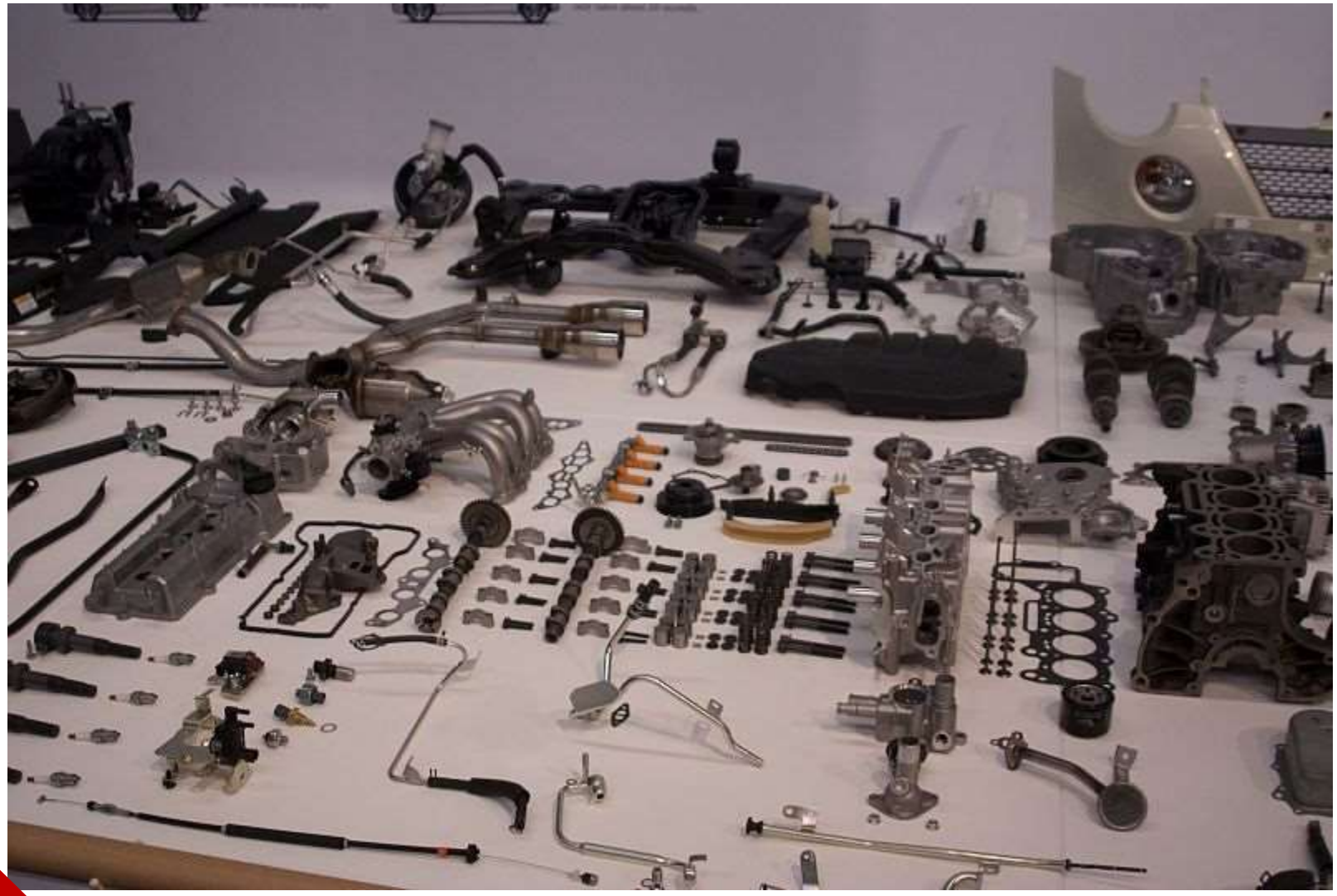
18-447 Lecture 25: Synchronization

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - be introduced to synchronization concepts
  - see hardware support for synchronization
- Notices
  - HW5, due Friday 4/28 midnight
  - Lab 4, due this week
  - **Final Exam,** May 4 Thu, 8:30am-11:30am
- Readings
  - P&H Ch2.11, Ch6
  - *Synthesis Lecture: Shared-Memory Synchronization*, 2013 (advanced optional)

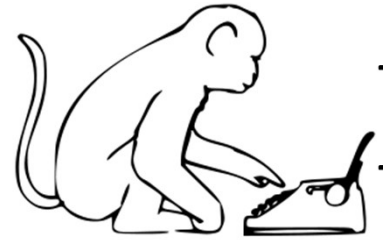
This is not 18-447



[Wikimedia Creative Commons]

Recall

What is 18-447

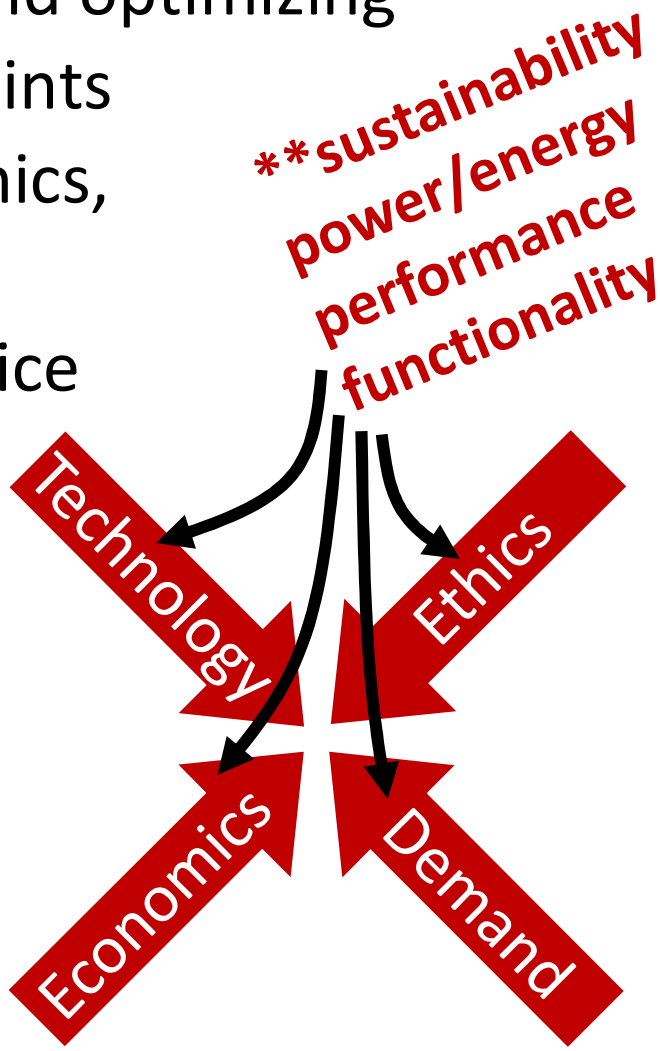


[Wikimedia Common]

- **Lab 1~3**: knowledge and skill
  - anyone with a wrench can take apart a car
  - Google Lens can tell you what each part is
  - trained person can put back a working car
- **Lab 4**: analyze and optimize
  - what design decisions make for a car that is fast vs good mileage?
  - how to decide how fast or efficient to make it?
- **Think, Ask, Invent**: what is the "right" future for personal transport?

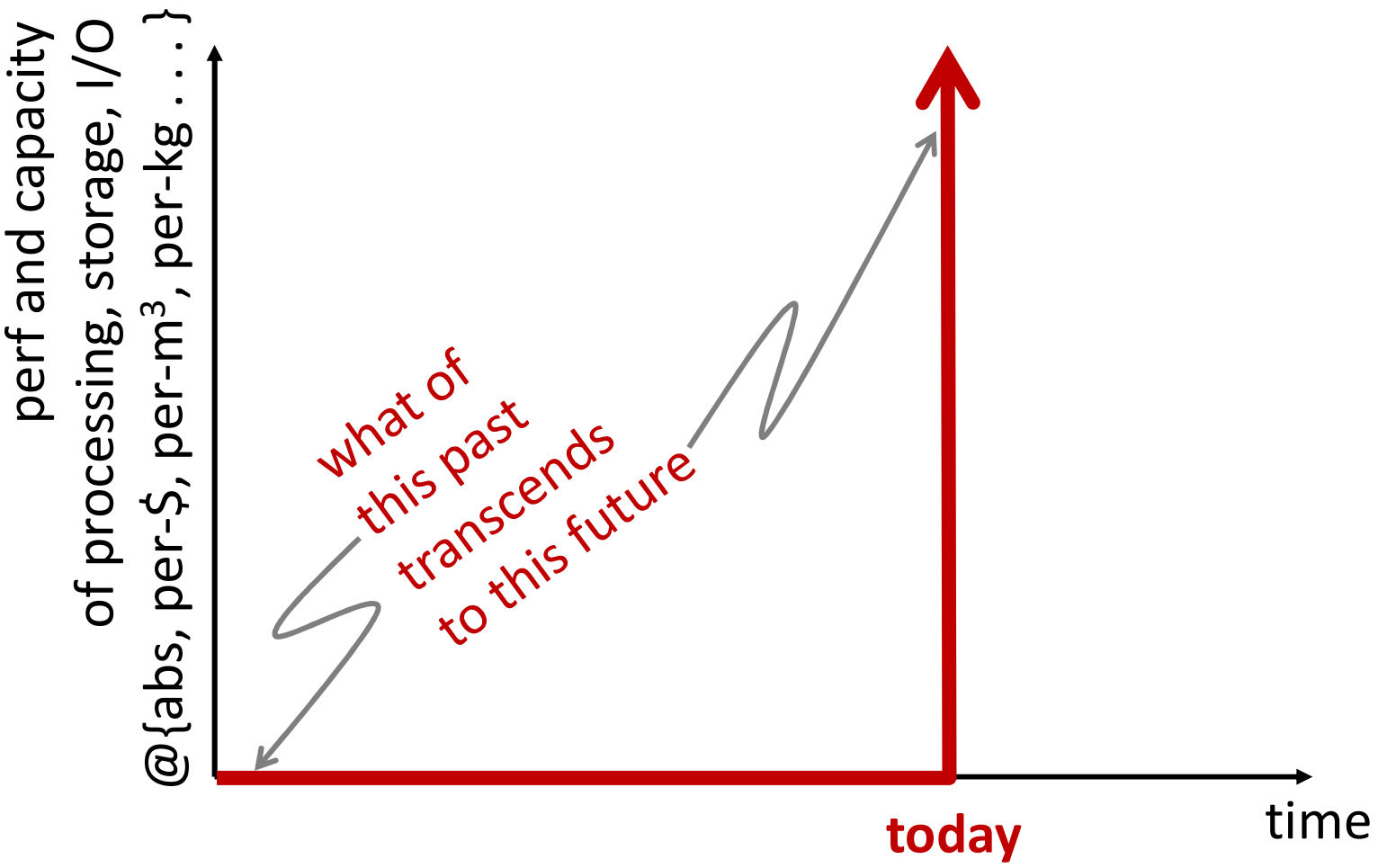
Computer Architecture is Engineering

- An applied discipline of finding and optimizing solutions under the joint constraints of demand, technology, economics, and ethics
- Thus, instances of what we practice evolve continuously
- Need to learn the principles that govern how to develop solutions to meet constraints
- **Don't memorize instances; understand why it is that way**



Recall

What exponential really looks like



You just have to stand far back enough to see it

Returning to normally scheduled programming

A simple example: producer-consumer

- Consumer waiting for result from producer in shared-memory variable **D**ata
- Producer uses another shared-memory variable **R**eady to indicate readiness (**R**=0 initially)

(upper-case for shared-mem **V**ariables)

producer:

```
      ……
      compute into D
      R=1
      ……
```

consumer:

```
      ……
      while(R!=1);
      consume D
      ……
```

- Straightforward if SC; if WC, need memory fences to order operations on **R** and **D**

Data Races

- E.g., threads T1 and T2 increment a shared-memory variable V initially 0 (assume SC)

```
T1:
        t=V
        t=t+1
        V=t
```

```
T2:
        t=V
        t=t+1
        V=t
```

Both threads both read and write V

- What happens depends on what T2 does in between T1's read and write to V (and vice versa)
- Correctness depends on T2 not reading or writing V between T1's read and write ("critical section")

Mutual Exclusion: General Strategy

- **Goal:** allow only either **T1** or **T2** to execute their respective critical sections at one time

  No overlapping of critical sections!

- **Idea:** use a shared-memory variable `Lock` to indicate whether a thread is already in critical section and the other thread should wait
- **Conceptual Primitives:**
  - **wait-on**: to check and block if `L` is already set
  - **acquire**: to set `L` before a thread enters critical sect
  - **release**: to clear `L` when a thread leaves critical sect

Mutual Exclusion: 1st Try

- Assume `L=0` initially

```
T1:
        while(L!=0);
        L=1;
critical{
        t=V
        t=func1(t,…)
        V=t
}
        L=0;
```

```
T2:
        while(L!=0);   <- wait
        L=1;           <- acquire
critical{
        t=V
        t=func2(t,…)
        V=t
}
        L=0;           <- release
```

But now have same problem with data race on `L`

Mutual Exclusion: Dekker's

- Using 3 shared-memory variables: Clear1=1, Clear2=1, Turn=1 or 2 initially (assumes SC)

```
C1=0;
while(C2==0)
     if (T==2) {
          C1=1;
          while(T==2);
          C1=0;
     }
{ . . . Critical Section . . . }
T=2;
C1=1;
```

```
C2=0;
while(C1==0)
     if (T==1) {
          C2=1;
          while(T==1);
          C2=0;
     }
{ . . . Critical Section . . . }
T=1;
C2=1;
```

(hint: T is the tie breaker)

- Can you decipher this? Extend to 3-way?

Need an easier, more general solution

Aside: what happens in Dekker's w/o T

- Using shared-memory variables: Clear1=1, Clear2=1 initially (assumes SC)

```
C1=0;
while(C2==0) {
    C1=1;
    some delay;
    C1=0;
}
{ . . . Critical Section . . . }
C1=1;
```

```
C2=0;
while(C1==0) {
    C2=1;
    some delay;
    C2=0;
}
{ . . . Critical Section . . . }
C2=1;
```

- Above is safe—if one side in C.S., the other isn't
- Either or both loop forever if pathological timing

Livelock possible

Aside: Dumb it down more

- Using shared-memory variables: Clear1=1, Clear2=1 initially (assumes SC)

```
C1=0;
while(C2==0) {
     some delay;
}

{ . . . Critical Section . . . }

C1=1;
```

```
C2=0;
while(C1==0) {
     some delay;
}

{ . . . Critical Section . . . }

C2=1;
```

- Above is still safe—if one side in C.S., the other isn't
- Both loop forever if tried at same time

Deadlock possible

Atomic Read-Modify-Write Instruction

- Special class of memory instructions to facilitate implementations of lock synchronizations
- Effects executed "atomically" (i.e. not interleaved by other reads and writes)
  - reads a memory location
  - performs some simple calculation
  - writes something back to the same location

HW guarantees no intervening read/write by others

E.g.,

```
<swap>(addr,reg):
    temp←MEM[addr];
    MEM[addr]←reg;
    reg←temp;
```

```
<test&set>(addr,reg):
    reg←MEM[addr];
    if (reg==0)
       MEM[addr]←1;
```

Expensive to implement and to execute

Acquire and Release

- Could rewrite earlier examples directly using **<swap>** or **<test&set>** instead loads and stores
- Better to hide ISA-dependence behind portable **Acquire()** and **Release()** routines

```
T1:
        Acquire(L);
critical {
        t=V
        t=func1(t,V,…)
        V=t
        Release(L);
```

```
T2:
        Acquire(L);
critical {
        t=V
        t=func2(t,V,…)
        V=t
        Release(L);
```

Note: implicit in **Acquire(L)** is to wait on **L** if not free

Acquire and Release

- Using **<swap>**, **L** initially 0

```
void Acquire(L) {
  do {
    reg=1;
    <swap>(L,reg);
  } while(reg!=0);
}
```

```
void Release(L) {
  L=0;
}
```

- Using **<test&set>**, **L** initially 0

```
void Acquire(L) {
  do {
    <test&set>(L,reg);
  } while(reg!=0);
}
```

```
void Release(L) {
  L=0;
}
```

Many equally powerful variations of atomic RMW insts can accomplish the same

High Cost of Atomic RMW Instructions

- Literal enforcement of atomicity very early on
- In CC shared-memory multiproc/multicores
  - RMW requires a writeable **M**/**E** cache copy
  - lock cacheblock from replacement during RMW
  - expensive when lock contended by many concurrent acquires—a lot of cache misses and cacheblock transfers, just to swap "1" with "1"
- Optimization
  - check lock value using normal load on read-only **S** copy
  - attempt RMW only when success is possible

```
do {
    reg=1;
    if (!L) {
        <swap>(L,reg);
    }
} while(reg!=0);
```

RMW without Atomic Instructions

- Add per-thread architectural state: ***reserved***, ***address*** and ***status***

```
<ld-linked>(reg,addr):
      reg ← MEM[addr];
      reserved ← 1;
      address ← addr;
```

```
<st-cond>(addr,reg):
      if (reserved &&
                address==addr)
          M[addr] ← reg;
          status ← 1;
      else
          status ← 0;
```

- **<ld-linked>** requests **S**-copy (if not alrdy **S** or **M**)
- HW clears ***reserved*** if cached copy lost due to CC (i.e., store or **<st-cond>** at another thread)
- If ***reserved*** stays valid until **<st-cond>,** request **M**-copy (if not already **M**) and update; can be no other intervening stores to ***address*** in between!!

Acquire() by ld-linked and st-cond

```
void Acquire(L) {
  do
     reg_w=1;
     do {
        <ld-linked>(reg_r,L)
     while (reg_r!=0);
     <st-cond>(L,reg_w);
  } while (status==0);
}
```

if **L** is modified in between by another thread, <st-cond> will fail and you know to try again

Resolving Data Race without Lock

- E.g., two threads **T1** and **T2** increment a shared-memory variable **V** initially 0 (assume SC)

context switch okay?

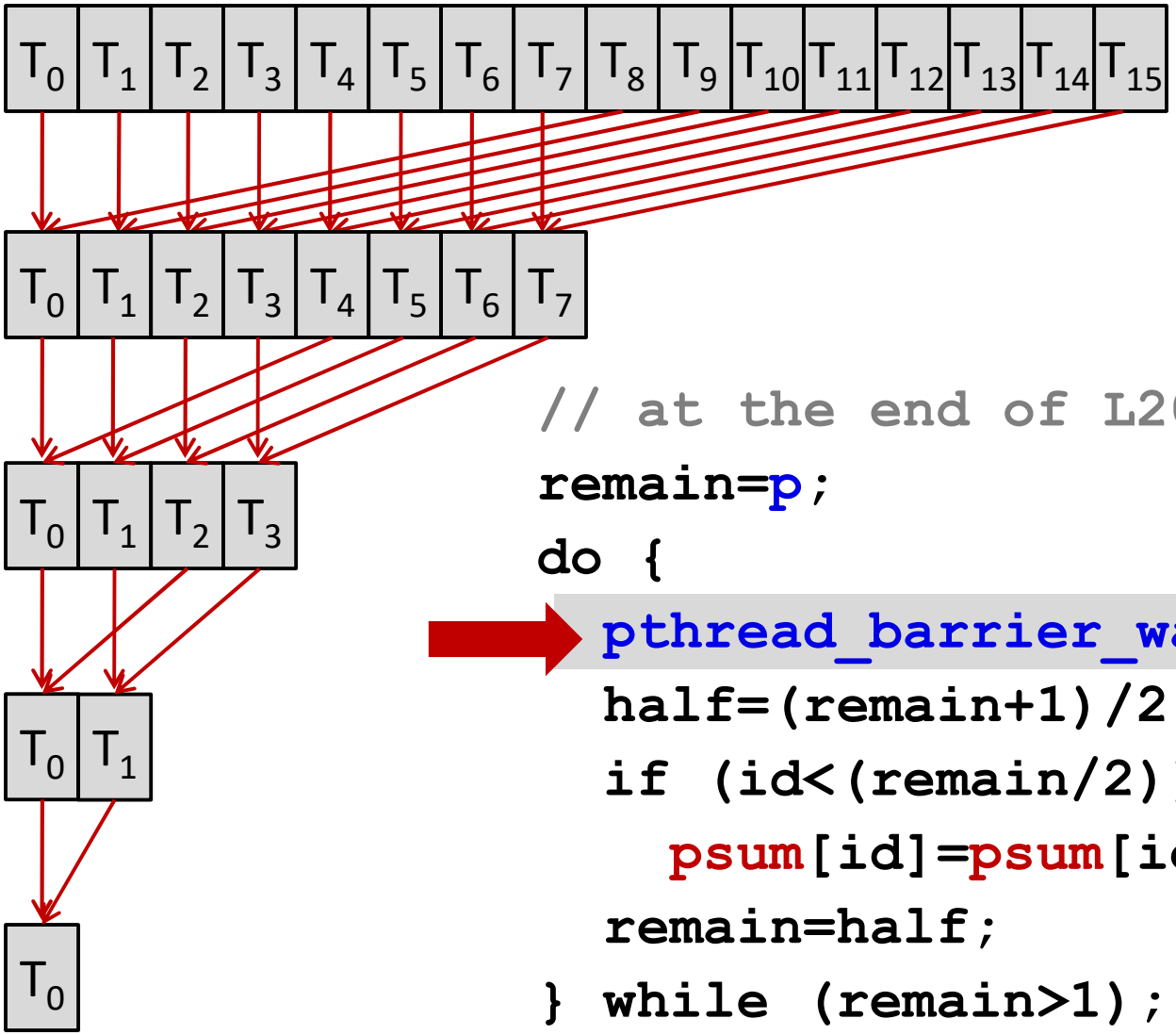
```
T1:
do {
   <ld-linked>(t,V)
   t=t+1
   <st-cond>(V,t)
} while(status==0)
```

```
T2:
do {
   <ld-linked>(t,V)
   t=t+1
   <st-cond>(V,t)
} while(status==0)
```

- Atomicity not guaranteed, but . . . .
- You know if you succeeded; no effect if you don't

Just try and try again until you succeed

Barrier Synchronization




```
// at the end of L20 sumParallel()
remain=p;
do {
  pthread_barrier_wait(&barrier);
  half=(remain+1)/2;
  if (id<(remain/2))
    psum[id]=psum[id]+psum[id+half];
  remain=half;
} while (remain>1);
```

(Blocking) Barriers

- Ensure a group of threads have all reached an agreed upon point
  - threads that arrive early have to wait
  - all are released when the last thread enters
- Can build from shared memory on small systems

  e.g., for a simple 1-time-use barrier (B=0 initially)

```
Acquire(L_B)
B=B+1;
Release(L_B)
while (B!=NUM_THREADS);
```

enter

wait

- Barrier on large systems are expensive, often supported/assisted by dedicated HW

Nonblocking Barriers

- Separate primitives for enter and exit
  - **enterBar()** is non-blocking and only records that a thread has reached the barrier

```
Acquire(L_B)
B=B+1;
Release(L_B)
```

  - **exitBar( )** blocks until the barrier is complete

```
while (B!=NUM_THREADS);
```

- A thread
  - calls **enterBar( )** then go on to independent work
  - calls **exitBar( )** only when no more work that doesn't depend on the barrier

Pass this point not on exams

For more, go read "Synthesis Lecture: Transactional Memory," 2nd Ed., 2010

Transactional Memory

```
T1:
      TxnBegin();

      t=V
      t=func1(t,V,…)
      V=t

      TxnEnd();
```

```
T2:
      TxnBegin();

      t=V
      t=func2(t,V,…)
      V=t

      TxnEnd();
```

- `Acquire(L)`/`Release(L)` say do one at a time
- `TxnBegin()`/`TxnEnd()` say "look like" done one at a time

Implementation can allow transactions to overlap and only fixes things if violations observable

Optimistic Execution Strategy

- Allow multiple transaction executions to overlap
- Detect atomicity violations between transactions
- On violation, one of the conflicting transactions is aborted (i.e., restarted from the beginning)
  - TM writes are speculative until reaching **`TxnEnd`**
  - speculative TM writes not observable by others
- Effective when actual violation is unlikely, e.g.,
  - multiple threads sharing a large structure/array
  - cannot decide statically which part of structure/array touched by different threads
  - conservative locking adds a cost to every access
  - TM incurs a cost only when data races occur

Detecting Atomicity Violation

- A transaction tracks memory **RdSet** and **WrSet**
- **Txn_a** appears atomic with respect to **Txn_b** if
  - **WrSet**(**Txn_a**) ∩ (**WrSet**(**Txn_b**) ∪ **RdSet**(**Txn_b**)) = ∅
  - **RdSet**(**Txn_a**) ∩ **WrSet**(**Txn_b**) = ∅
- Lazy Detection
  - broadcast **RdSet** and **WrSet** to other txns at **TxnEnd**
  - waste time on txns that failed early on
- Eager Detection
  - check violations on-the-fly by monitoring other txns' reads and writes
  - require frequent communications

Oversimplified HW-based TM using CC

- Add **RdSet** and **WrSet** status bits to identify cacheblocks accessed since **TxnBegin**
- Speculative TM writes
  - issue **BusRdOwn**/**Invalidate** if starting in **I** or **S**
  - issue **BusWr**(old value) on first write to **M** block
  - on abort, silently invalidate **WrSet** cacheblocks
  - on reaching **TxnEnd**, clear **RdSet**/**WrSet** bits

  Assume **RdSet**/**WrSet** cacheblocks are never displaced
- Eager Detection
  - snoop for **BusRd**, **BusRdOwn**, and **Invalidation**
  - **M**→**S**, **M**→**I** or **S**→**I** downgrades to **RdSet**/**WrSet** indicative of atomicity violation

  Which transaction to abort?

sketch only

Why not transaction'ize everything?

```
void *sumParallel
      (void *_id) {
  long id=(long) _id;
  long i;
  long N=ARRAY_SIZE/p;

  TxnBegin();
  for(i=0;i<N;i++) {
    double v=A[id*N+i];
    if (v>=0)
      SumPos+=v;
    else
      SumNeg+=v;
  }
  TxnEnd();
}
```

```
void *sumParallel
      (void *_id) {
  long id=(long) _id;
  long i;
  long N=ARRAY_SIZE/p;

  for(i=0;i<N;i++) {
    TxnBegin();
    double v=A[id*N+i];
    if (v>=0)
      SumPos+=v;
    else
      SumNeg+=v;
    TxnEnd();
  }
}
```

p=2

```
void *sumParallel
      (void *_id) {
  long id=(long) _id;
  long i;
  long N=ARRAY_SIZE/p;

  for(i=0;i<N;i++) {
    double v=A[id*N+i];
    if (v>=0) {
      TxnBegin();
      SumPos+=v;
      TxnEnd();
    } else {
      TxnBegin();
      SumNeg+=v;
      TxnEnd();
    }
  }
}
```

Compute separate sums of positive and negative elements of **A** *in* **SumPos** *and* **SumNeg**

Better??

Overhead vs Likelihood of Succeeding

```
void *sumParallel
       (void *_id) {
  long id=(long) _id;
  long i;
  long N=ARRAY_SIZE/P;
  double psumPos=0;
  double psumNeg=0;

  for(i=0;i<N;i++) {
    double v=A[id*N+i];
    if (v>=0)
      psumPos+=v;
    else
      psumNeg+=v;
  }
  TxnBegin();
  if(psumPos) SumPos+=psumPos;
  if(psumNeg) SumNeg+=psumNeg;
  TxnEnd();
}
```

local non-shared

versus

```
if(psumPos) {
    Acquire(L_pos);
    SumPos+=psumPos;
    Release(L_pos);
}
if(psumNeg) {
    Acquire(L_neg);
    SumNeg+=psumNeg;
    Release(L_neg);
}
```

```
if(psumPos||psumNeg) {
    Acquire(L);
    SumPos+=psumPos;
    SumNeg+=psumNeg;
    Release(L);
}
```