Foundations:
Concurrency Concerns
Synchronization Basics

Chris Rossbach

CS378

Today

- Questions?
- Administrivia
  - You’ve started Lab 1 right?
- Foundations
  - Parallelism
  - Basic Synchronization
  - Threads/Processes/Fibers, Oh my!
  - Cache coherence

- Acknowledgments: some materials in this lecture borrowed from
  - Emmett Witchel (who borrowed them from: Kathryn McKinley, Ron Rockhold, Tom Anderson, John Carter, Mike Dahlin, Jim Kurose, Hank Levy, Harrick Vin, Thomas Narten, and Emery Berger)
  - Mark Silberstein (who borrowed them from: Blaise Barney, Kunle Olukoton, Gupta)
  - Andy Tannenbaum
  - Don Porter
  - me…
  - Photo source: https://img.devrant.com/devrant/rant/r_10875_uRYQF.jpg

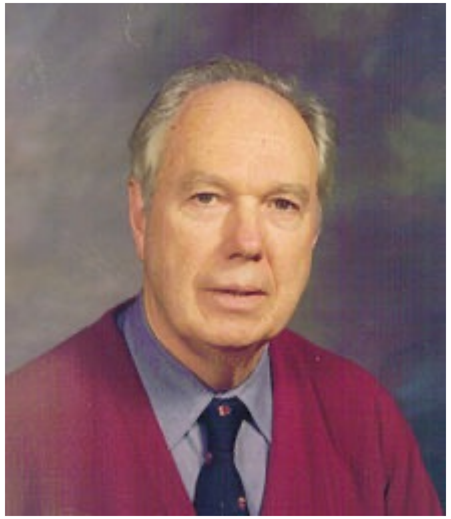
Faux Quiz (answer any 2, 5 min)

- Who was Flynn? Why is her/his taxonomy important?
- How does domain decomposition differ from functional decomposition? Give examples of each.
- Can a SIMD parallel program use functional decomposition? Why/why not?
- What is an RMW instruction? How can they be used to construct synchronization primitives? How can sync primitives be constructed without them?

Who is Flynn?

Michael J. Flynn

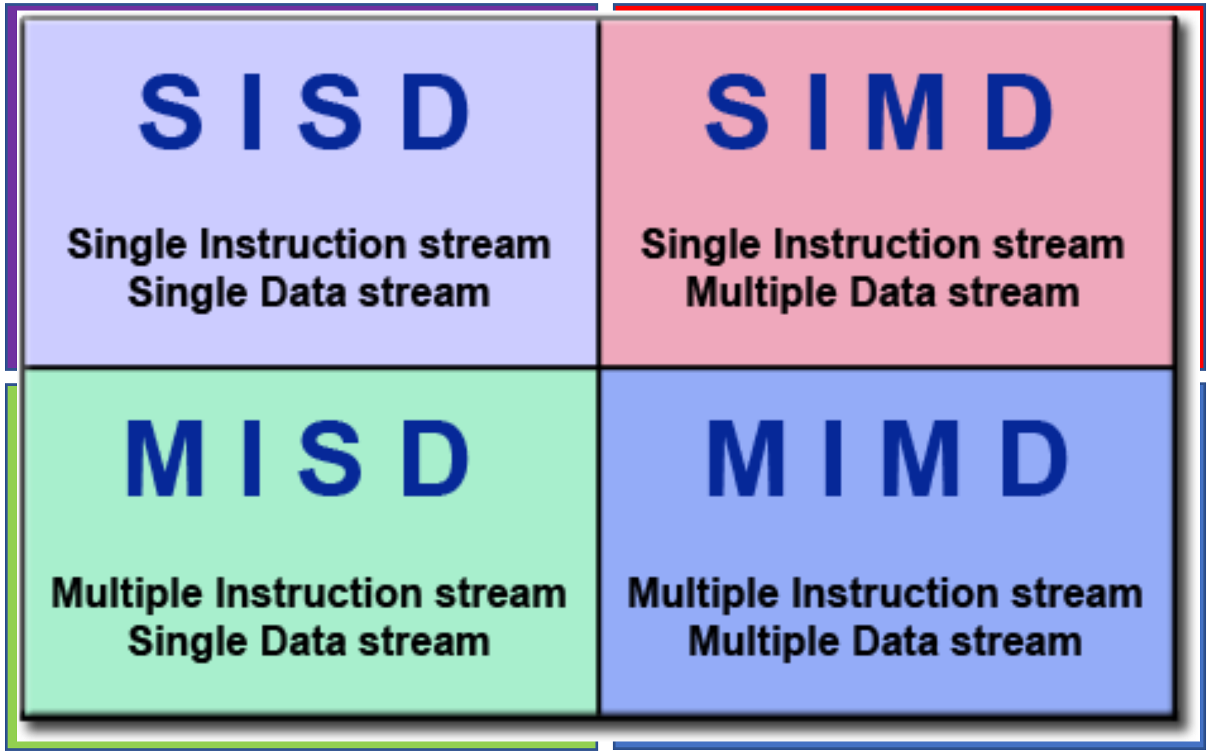
- Emeritus at Stanford
- Proposed taxonomy in 1966 (!!)
- 30 pages of publication titles
- Founding member of SIGARCH



- (Thanks Wikipedia)

Review: Flynn’s Taxonomy

Y AXIS:
Instruction
Streams

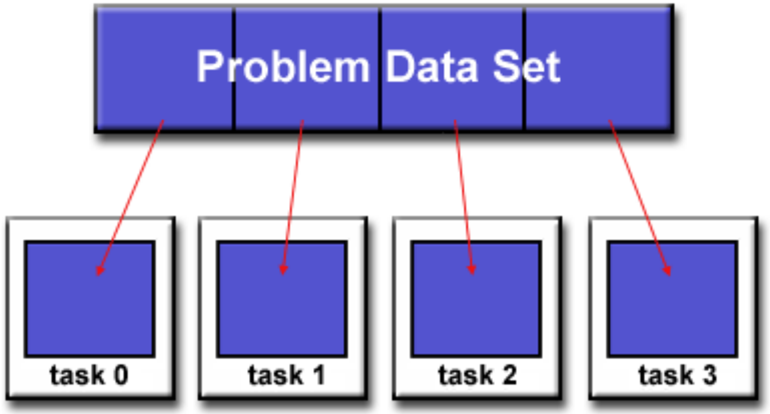


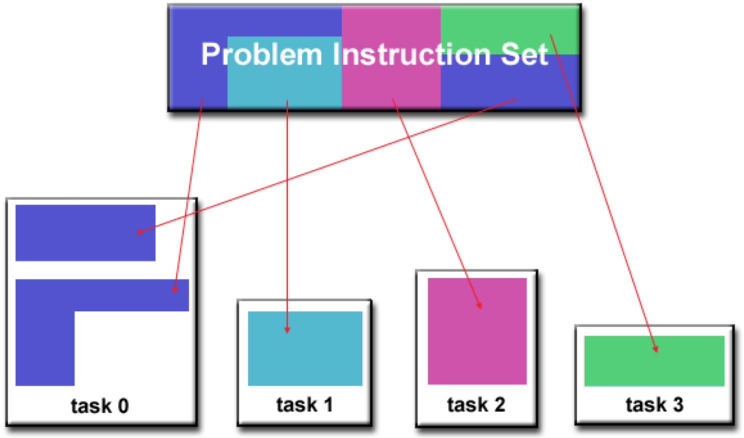
SISD Single Instruction stream Single Data stream	**SIMD** Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	**MIMD** Multiple Instruction stream Multiple Data stream

X AXIS:
Data Streams

Review: Problem Partitioning

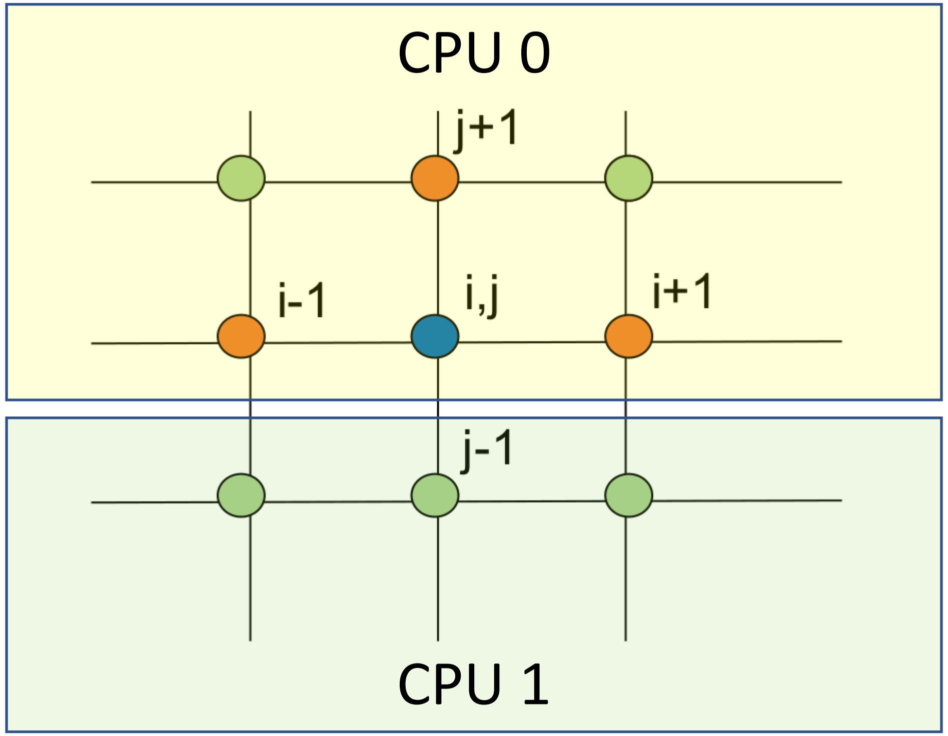
- Domain Decomposition
  - SPMD
  - Input domain
  - Output Domain
  - Both
- Functional Decomposition
  - MPMD
  - Independent Tasks
  - Pipelining





Domain decomposition

- Each CPU gets part of the input

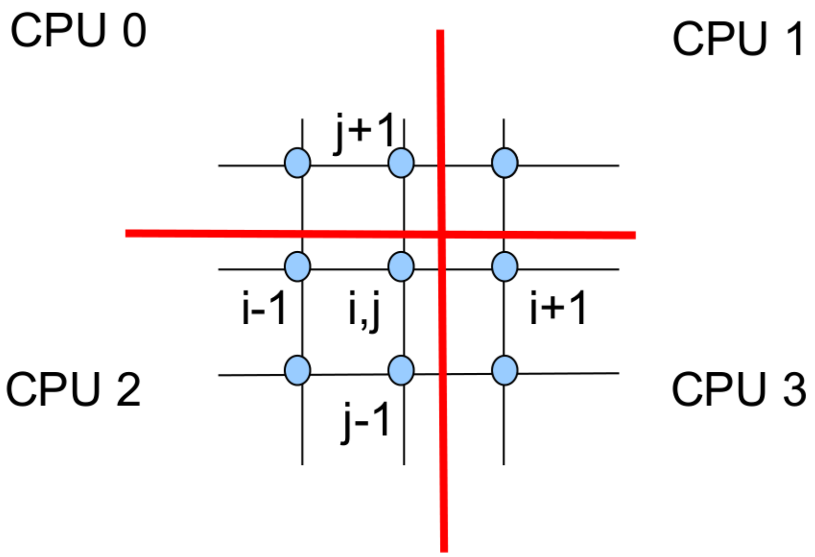


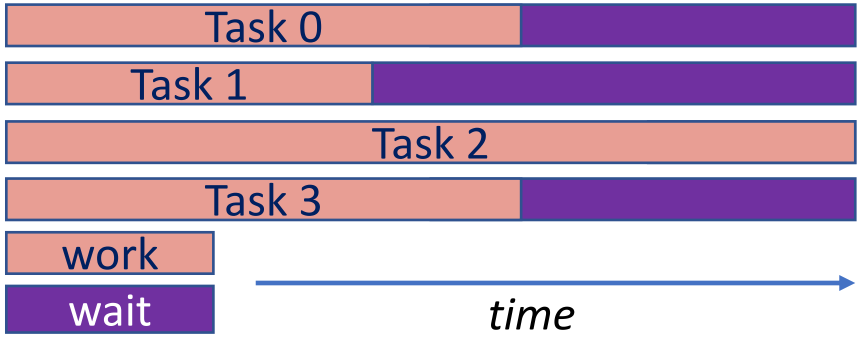
Issues?

- Accessing Data
  - Can we access v(i+1, j) from CPU 0
    - …as in a "normal" serial program?
    - Shared memory? Distributed?
  - Time to access v(i+1,j) == Time to access v(i-1,j) ?
  - *Scalability vs Latency*
- Control
  - Can we assign one vertex per CPU?
  - Can we assign one vertex per process/logical task?
  - *Task Management Overhead*
- *Load Balance*
- Correctness
  - order of reads and writes is non-deterministic
  - synchronization is required to enforce the order
  - *locks, semaphores, barriers, conditionals….*

Load Balancing

- Slowest task determines performance

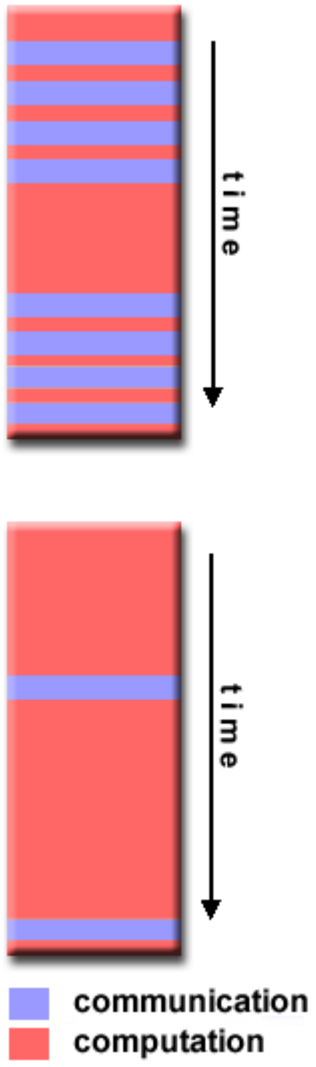




Granularity

$$G = \frac{Computation}{Communication}$$

- Fine-grain parallelism
  - G is small
  - Good load balancing
  - Potentially high overhead
  - Hard to get correct
- Coarse-grain parallelism
  - G is large
  - Load balancing is tough
  - Low overhead
  - Easier to get correct



Performance: Amdahl’s law

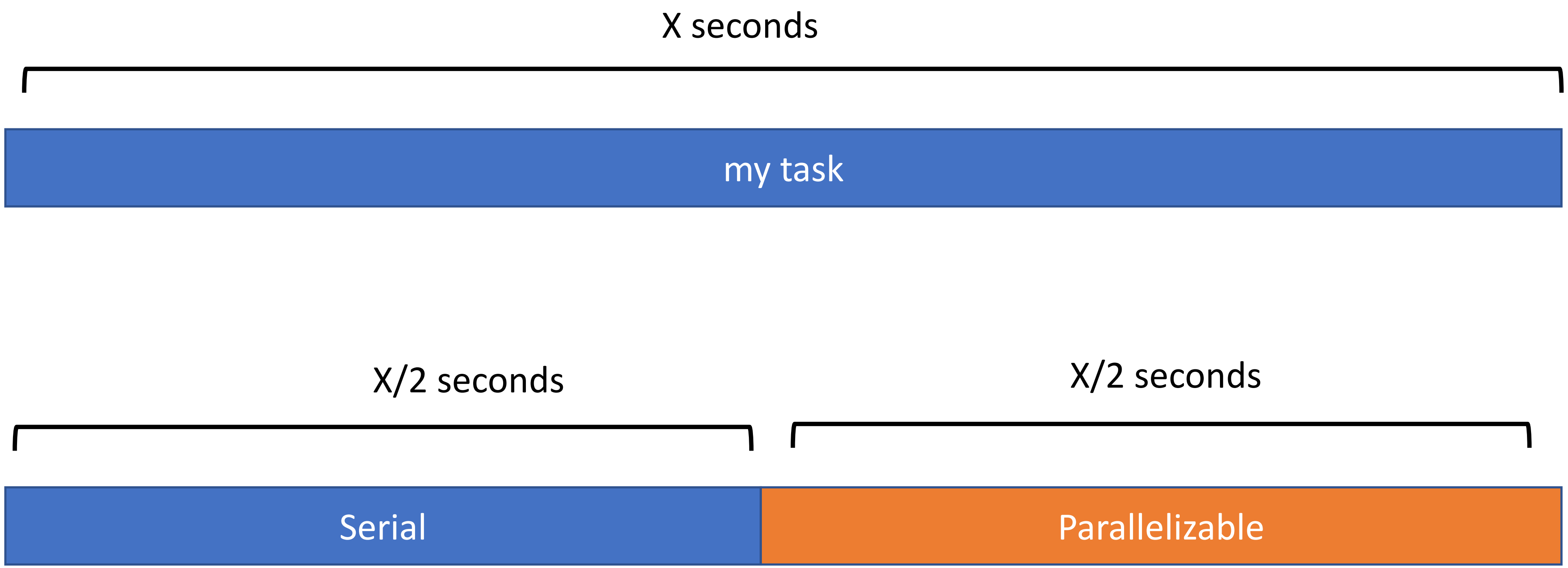
- Speedup is bound by serial component
- Sp

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}}$$

$\#CPUs$

$$Speedup(\#CPUs) = \frac{T_{serial}}{T_{parallel}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)}$$

Amdahl’s law



X seconds

my task

X/2 seconds

X/2 seconds

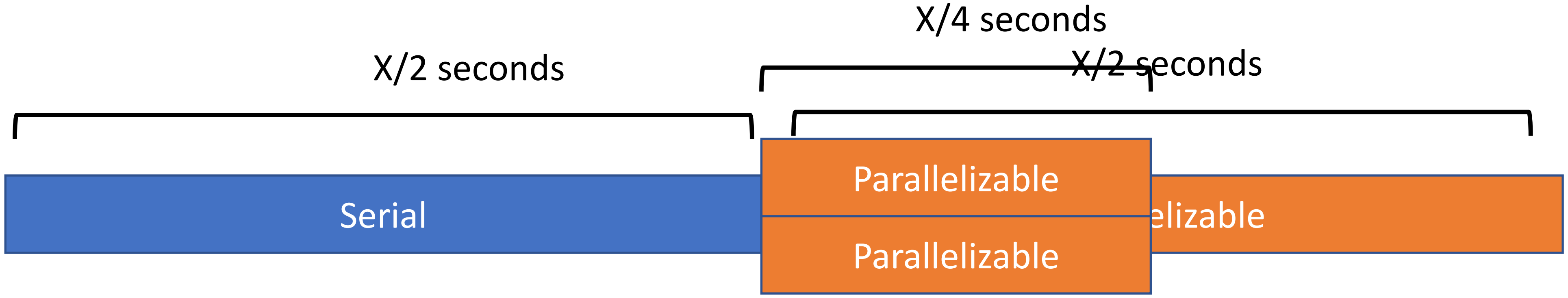
Serial

Parallelizable

What makes something “serial” vs. parallelizable?

Amdahl’s law

2 CPUs



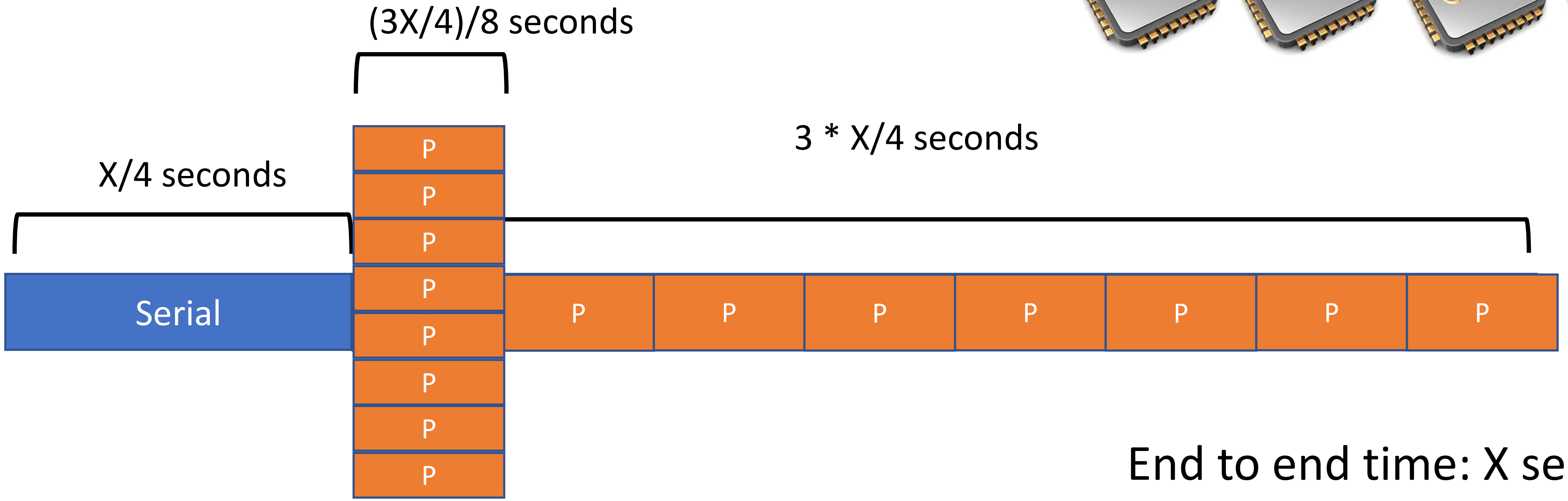
End to end time: (X/2 + X/4) = (3/4)X seconds

What is the “speedup” in this case?

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{\frac{.5}{2\ \text{cpus}} + (1\text{-}.5)} = 1.333$$

Speedup exercise

8 CPUs

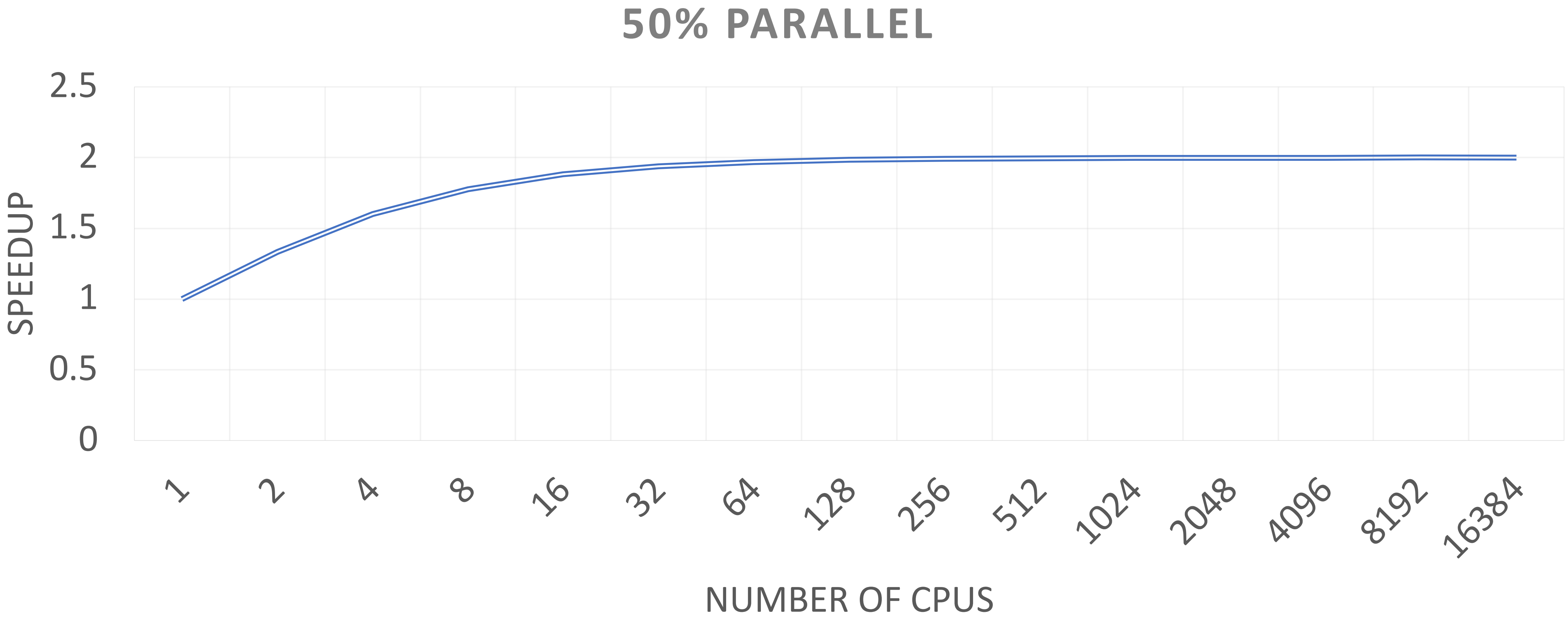


End to end time: X seconds

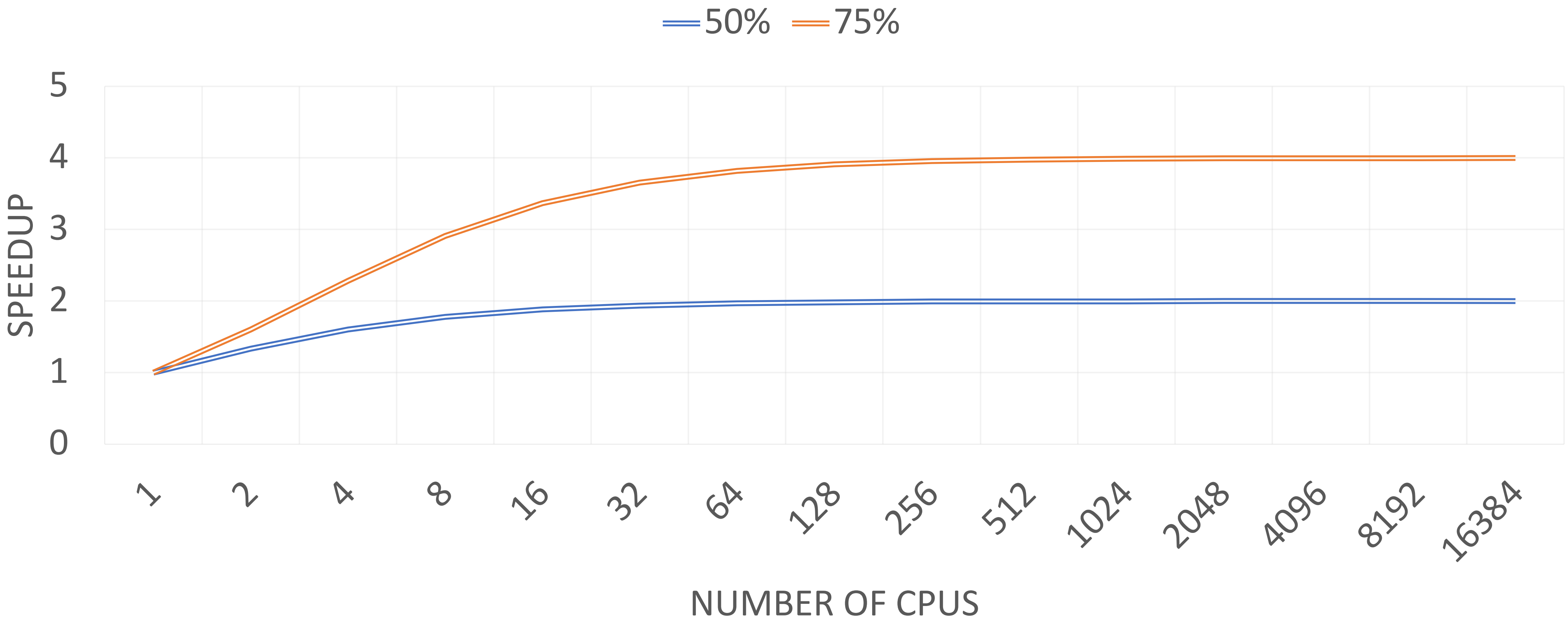
What is the “speedup” in this case?

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{.75/8 + (1-.75)} = 2.91\text{x}$$

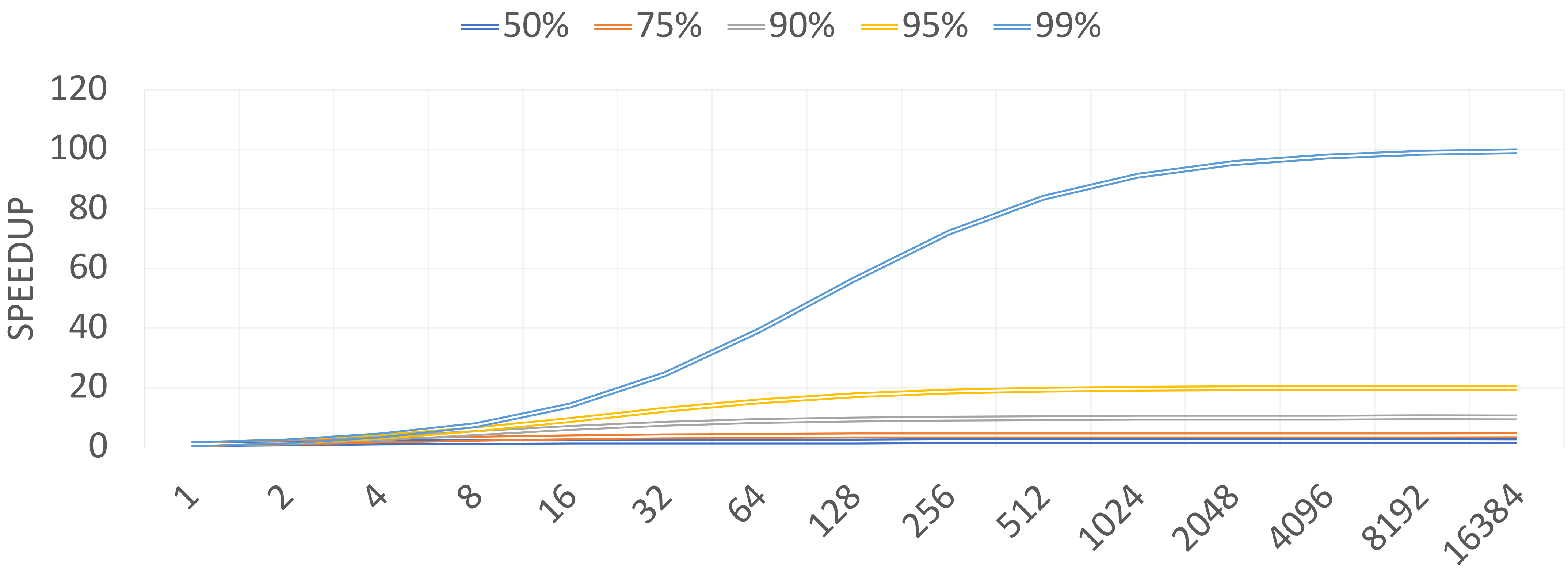
Amdahl Action Zone



Amdahl Action Zone



Amdahl Action Zone



Strong Scaling vs Weak Scaling

Amdahl vs. Gustafson

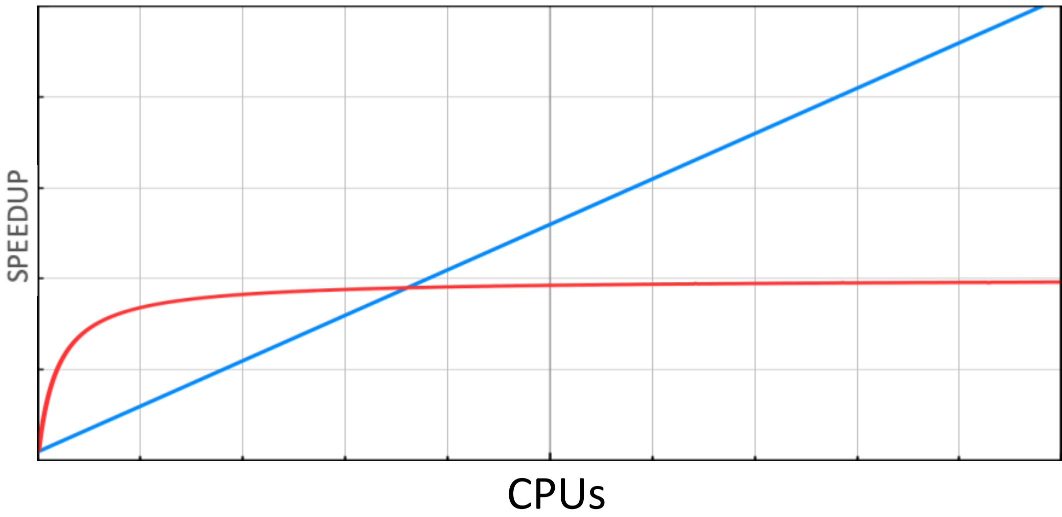
- $N = \#CPUs,\ S = serial\ portion = 1 - A$
- Amdahl's law: $Speedup(N) = \frac{1}{\frac{A}{N}+S}$
  - **Strong scaling:** $Speedup(N)$ calculated given total amount of work is fixed
  - Solve same problems faster when problem size is fixed and #CPU grows
  - Assuming parallel portion is fixed, speedup soon seizes to increase

- Gustafson's law: *Speedup(N)* = S + (S-1)*N
  - **Weak scaling:** *Speedup(N) calculated given work per CPU is fixed*
  - Work/CPU fixed when adding more CPUs keeps granularity fixed
  - Problem size grows: solve larger problems
  - **Consequence:** speedup upper bound is much higher
  - Given work W on n CPUs, with α serial
    - Incremental work W' on (n+1) CPUs:

      $W'=\alpha W+(1-\alpha)nW$
    - Speedup based on case where (1-α) scales perfectly:

$$S(n) = \frac{\alpha W+(1-\alpha)nW}{\alpha W+\frac{(1-\alpha)nW}{n}}$$

$$S(n)=\alpha+(1-\alpha)n$$



When is Gustavson's law a better metric?
When is Amdahl's law a better metric?

Super-linear speedup

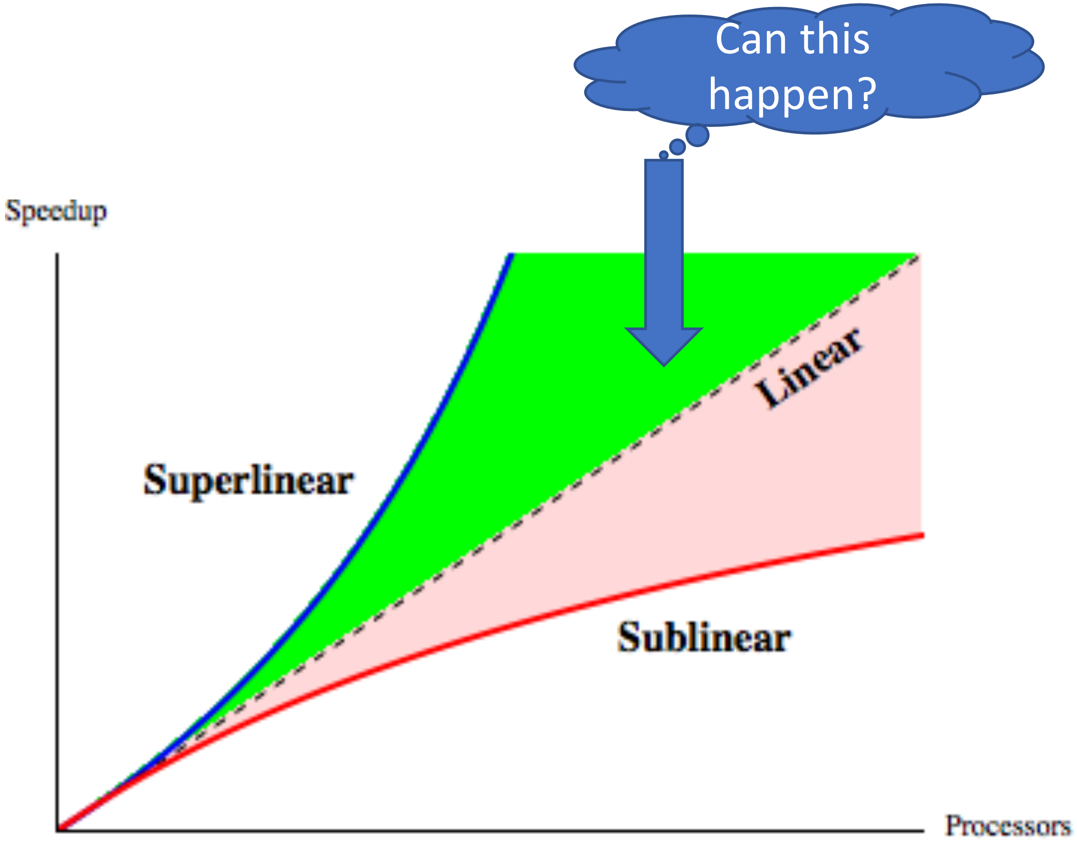
- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

Efficient **bubble sort**

- *Serial: 150s*
- *Parallel 40s*
- *Speedup:* $\frac{150}{40} = 3.75$?

NO NO NO!

- *Serial quicksort: 30s*
- *Speedup = 30/40 = 0.75X*



Can this happen?

Why insist on best serial algorithm as baseline?

Concurrency and Correctness

If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Thread 2

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

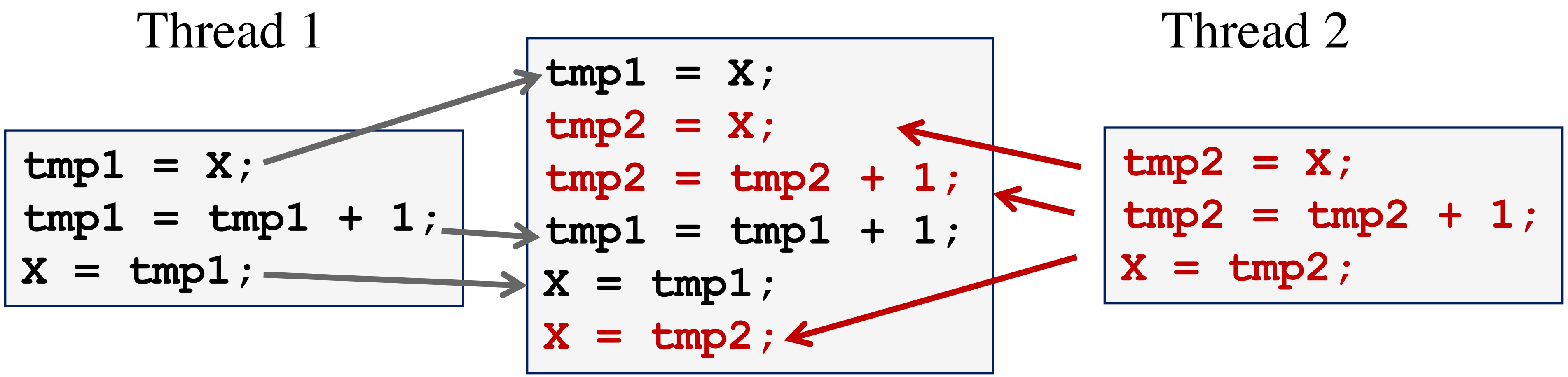
Answer:

A. 0
B. 1
C. 2
D. More than 2

Schedules/Interleavings

Model of concurrent execution

- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, synchronization is needed



Thread 1

```
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;
```

```
tmp1 = X;
tmp2 = X;
tmp2 = tmp2 + 1;
tmp1 = tmp1 + 1;
X = tmp1;
X = tmp2;
```

Thread 2

```
tmp2 = X;
tmp2 = tmp2 + 1;
X = tmp2;
```

If X==0 initially, X == 1 at the end. WRONG result!

Locks fix this with Mutual Exclusion

```
void increment() {
   lock.acquire();
   int temp = X;
   temp = temp + 1;
   X = temp;
   lock.release();
}
```

Mutual exclusion ensures only safe interleavings

- *But it limits concurrency, and hence scalability/performance*

Is mutual exclusion a good abstraction?

Why are Locks "Hard?"

- Coarse-grain locks
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

```
// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Fine-grain locks
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

```
       Thread 0                   Thread 1
move(a, b, key1);

                          move(b, a, key2);
```

DEADLOCK!

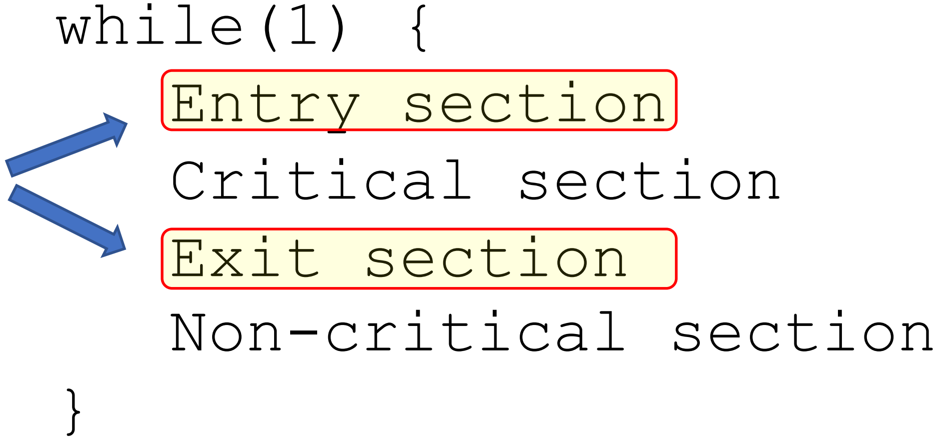
Review: correctness conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region
- Bounded waiting
  - ~~A thread that enters the entry section enters the critical section within some bounded number of operations.~~
  - *If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i's request is granted*

Theorem: Every property is a combination of a safety property and a liveness property.

-Bowen Alpern & Fred Schneider

https://www.cs.cornell.edu/fbs/publications/defliveness.pdf



Mutex, spinlock, etc.
are ways to implement

```
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

Did we get all the important conditions?

Why is correctness defined in terms of locks?

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
  while (*lock == 1)
     ; //spin
  *lock = 1;
}
```

```
Lock::Release() {
    *lock = 0;
}
```

Completely and utterly broken.
How can we fix it?

What are the problem(s) with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work

HW Support for Read-Modify-Write (RMW)

IDEA: hardware implements something like:

```
bool rmw(addr, value) {
  atomic {
    tmp = *addr;
    newval = modify(tmp);
    *addr = newval;
   }
}
```

Why is that hard? How can we do it?

Preview of Techniques:

- Bus locking
- Single Instruction ISA extensions
  - Test&Set
  - CAS: Compare & swap
  - Exchange, locked increment, locked decrement (x86)
- Multi-instruction ISA extensions:
  - LLSC: (PowerPC,Alpha, MIPS)
  - Transactional Memory (x86, PowerPC)

More on this later…

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
while (test&set(lock) == 1)
   ; //spin
}
```

```
Lock::Release() {
    *lock = 0;
}
```

(test & set ~= CAS ~= LLSC)

TST: ***Test&set***

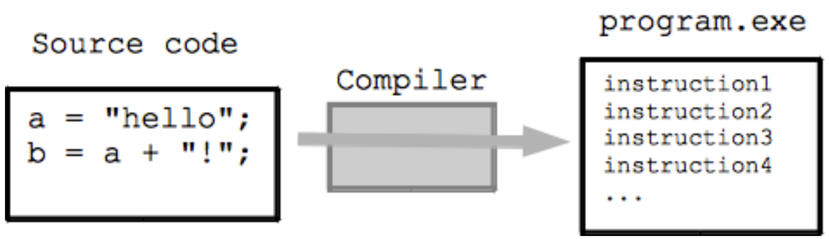
- Reads a value from memory
- Write “1” back to memory location

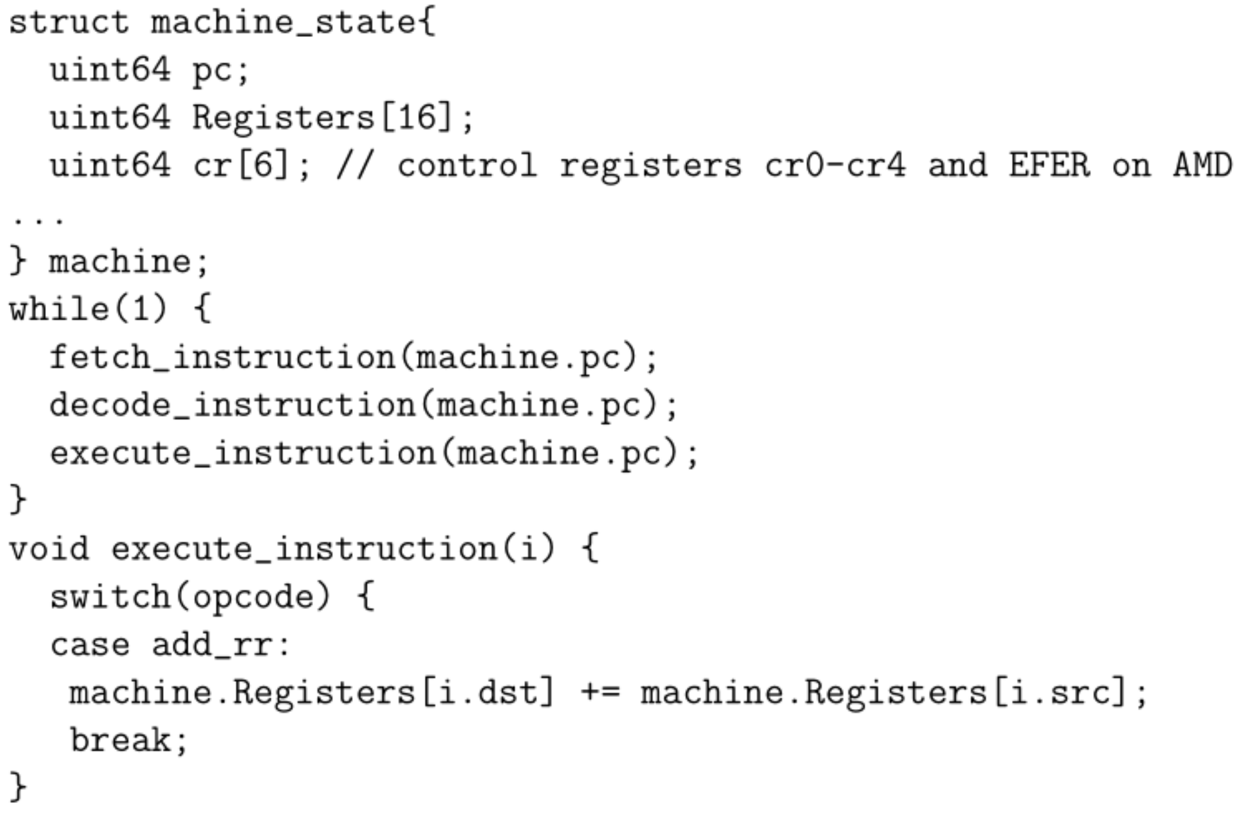
What are the problem(s) with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work

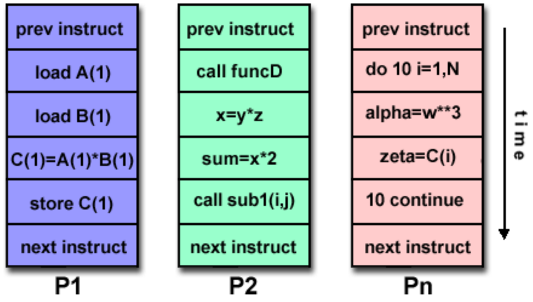
More on this later…

Programming and Machines: a mental model





```
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
   machine.Registers[i.dst] += machine.Registers[i.src];
   break;
}
```



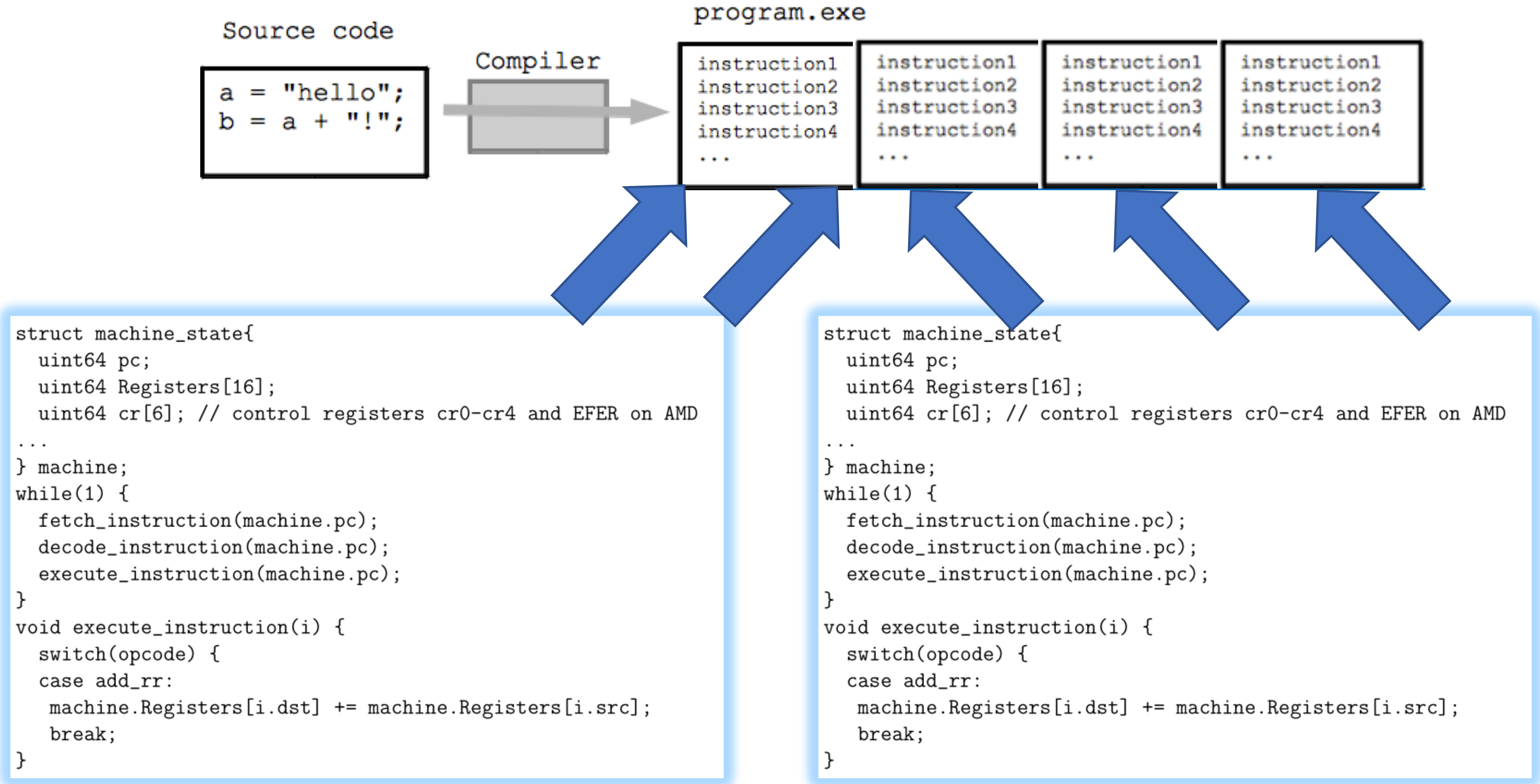
prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1

prev instruct
call funcD
x=y*z
sum=x*2
call sub1(i,j)
next instruct
P2

prev instruct
do 10 i=1,N
alpha=w**3
zeta=C(i)
10 continue
next instruct
Pn

time

Parallel Machines: a mental model



Source code

```
a = "hello";
b = a + "!";
```

Compiler

program.exe

```
instruction1
instruction2
instruction3
instruction4
...
```

```
instruction1
instruction2
instruction3
instruction4
...
```

```
instruction1
instruction2
instruction3
instruction4
...
```

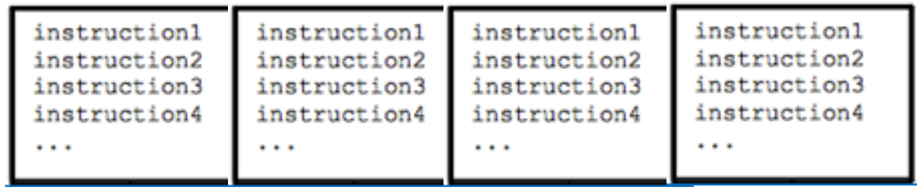
```
instruction1
instruction2
instruction3
instruction4
...
```

```
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
   machine.Registers[i.dst] += machine.Registers[i.src];
   break;
}
```

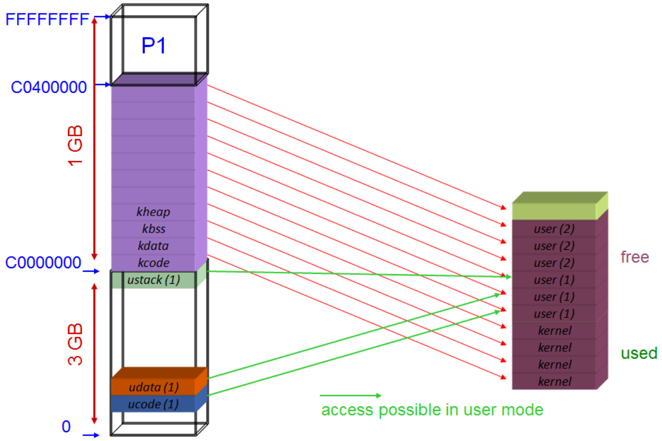
```
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
   machine.Registers[i.dst] += machine.Registers[i.src];
   break;
}
```

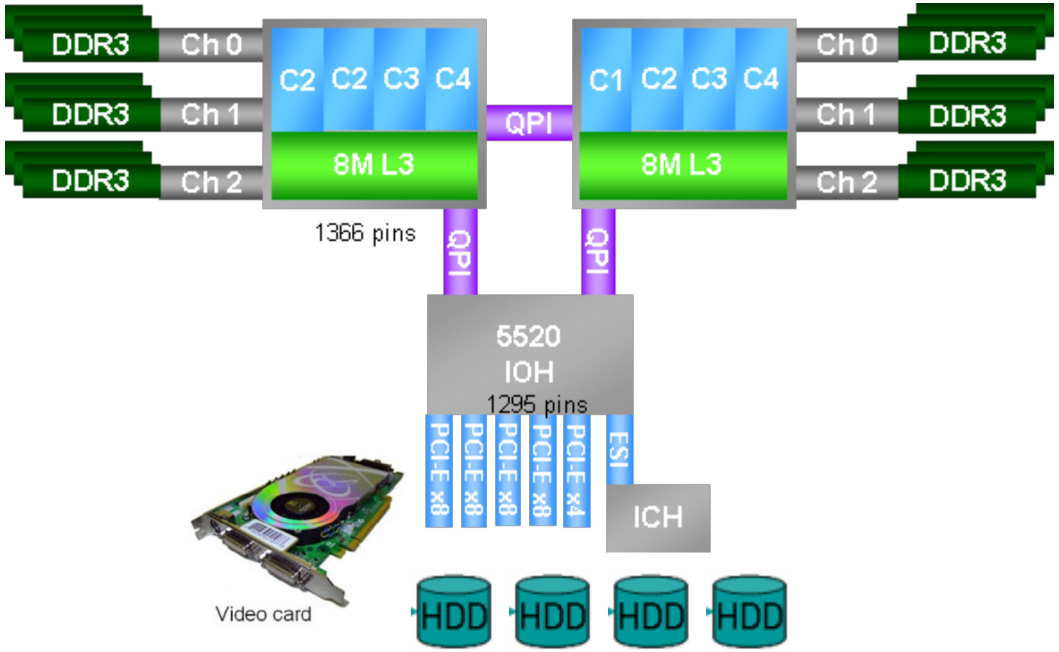
Processes and Threads and Fibers…

- Abstractions
- Containers
- State
    - Where is shared state?
    - How is it accessed?
    - Is it mutable?

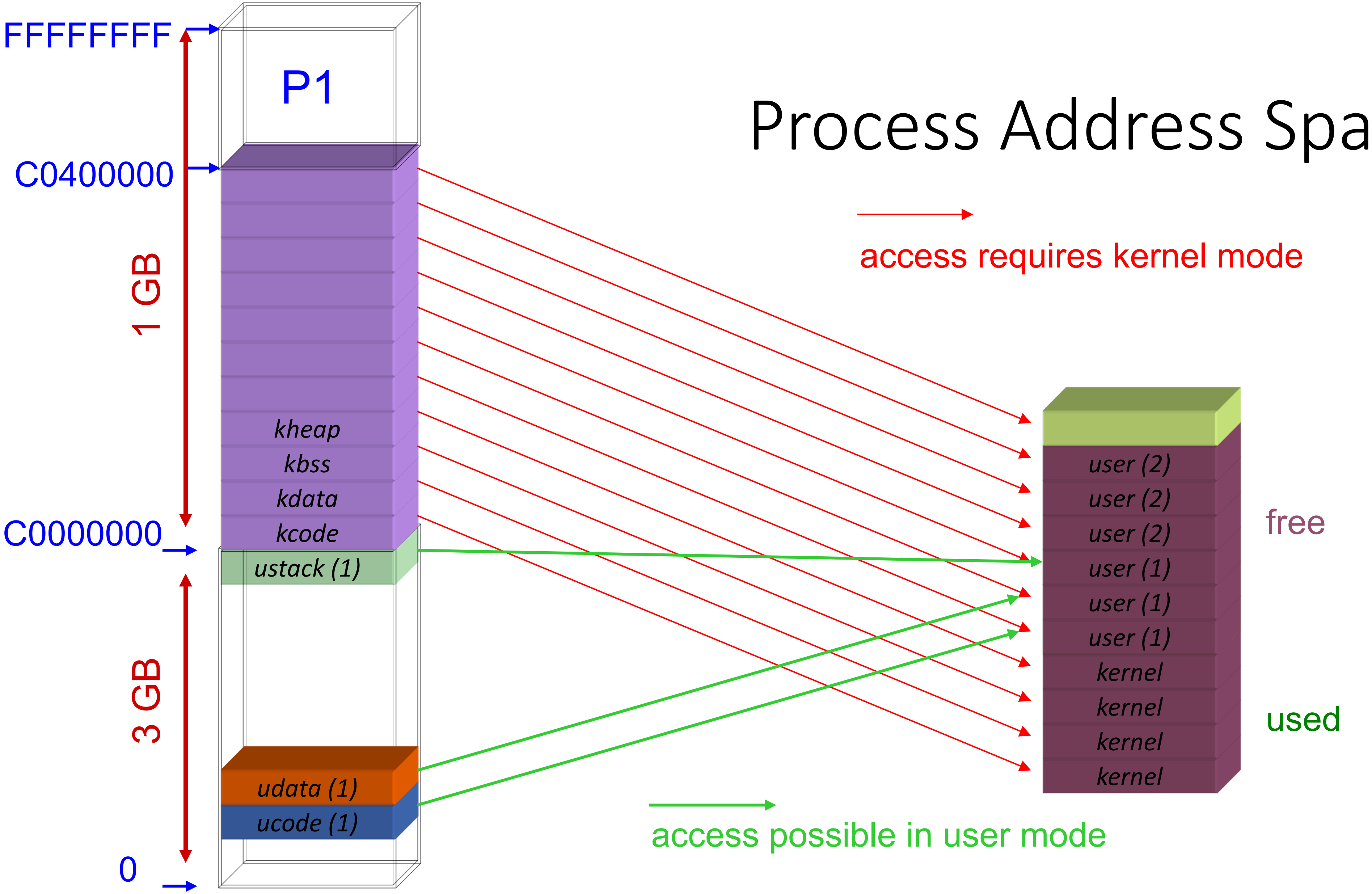


```
instruction1   instruction1   instruction1   instruction1
instruction2   instruction2   instruction2   instruction2
instruction3   instruction3   instruction3   instruction3
instruction4   instruction4   instruction4   instruction4
...            ...            ...            ...
```





Process Address Space



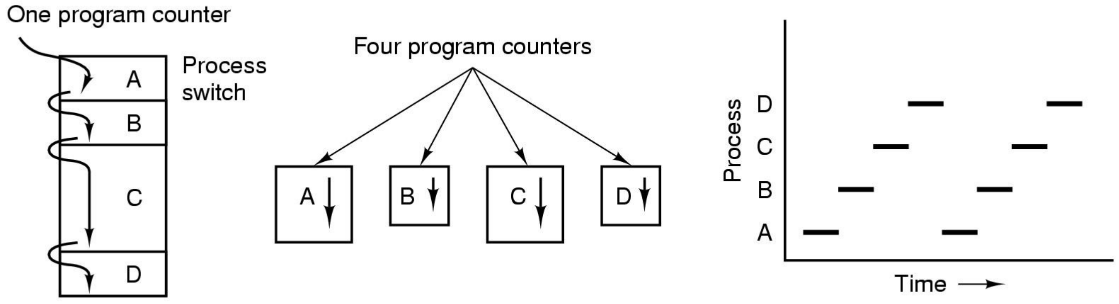
access requires kernel mode

access possible in user mode

Anyone see an issue?

Processes

Model



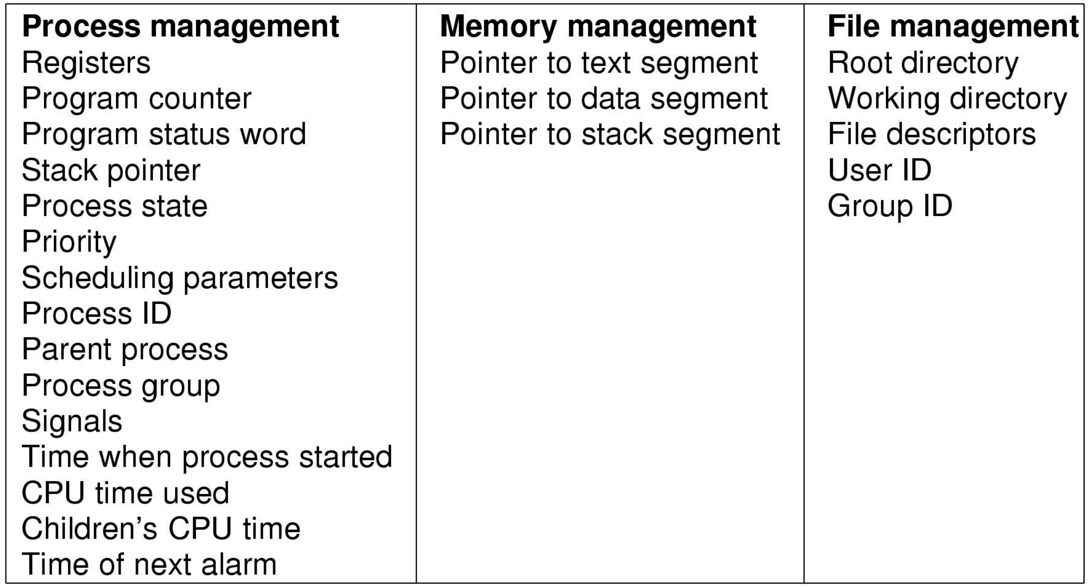
(a)

(b)

(c)

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Implementation

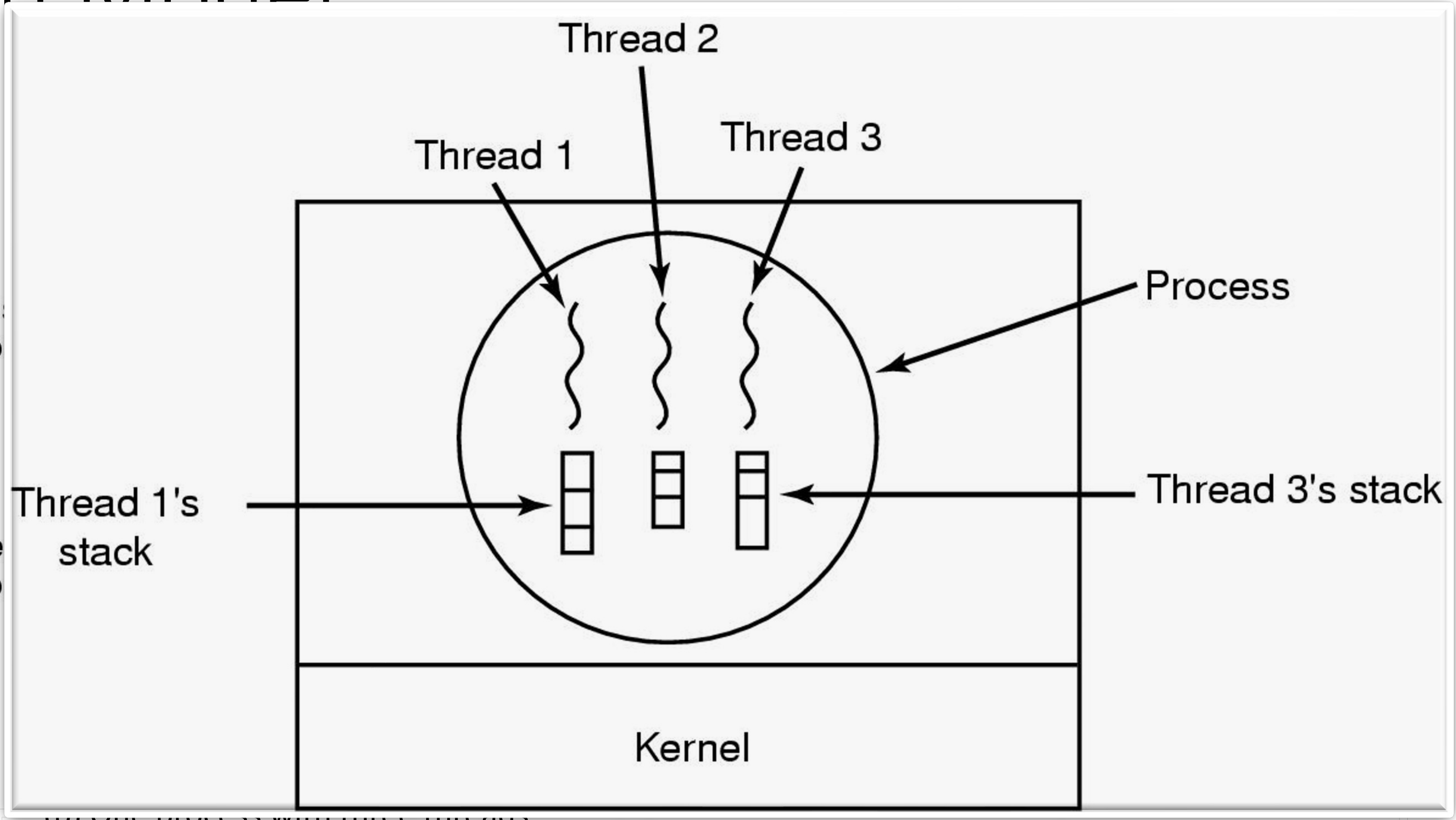


Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children’s CPU time		
Time of next alarm		

Thread Model

U
sp

Ke
sp

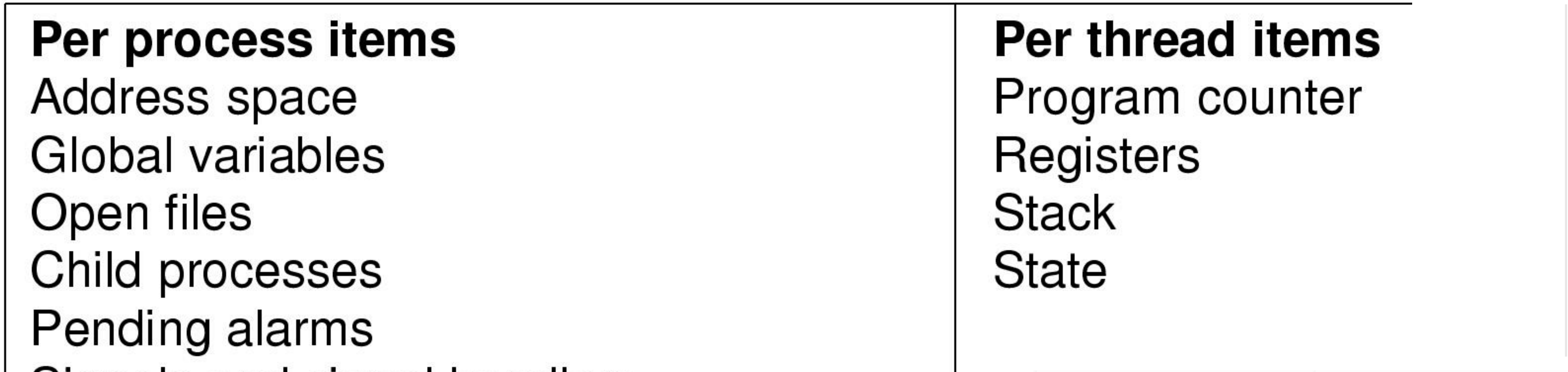


(b) One process with three threads

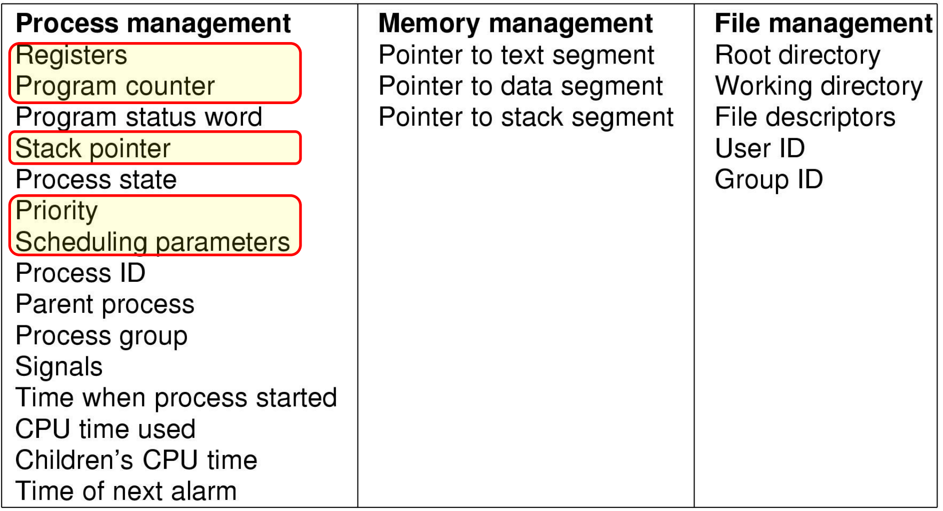
Each thread has its own stack

When might (a) be better than (b)? Vice versa?

The Thread Model



Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

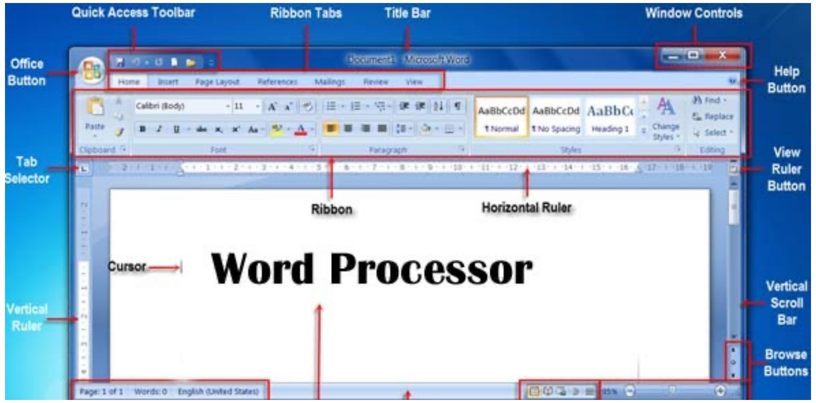


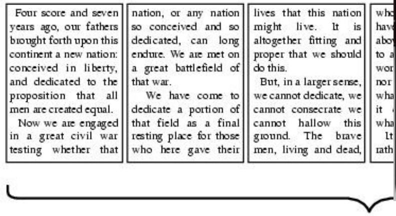
Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children’s CPU time		
Time of next alarm		

- Items shared by all threads in a process
- Items private to each thread

Using threads

Ex. How might we use threads in a word processo



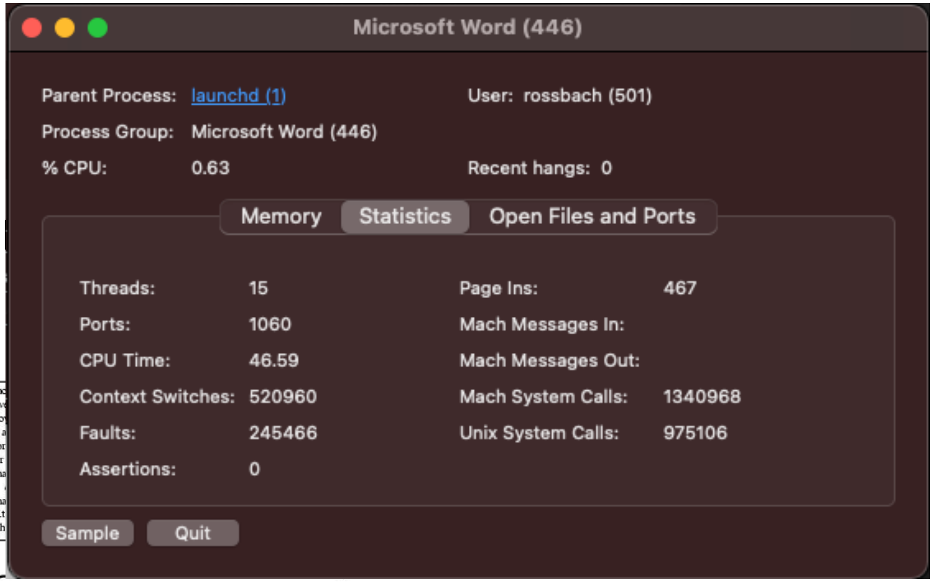


Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war testing whether that

nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field as a final resting place for those who here gave their

lives that this nation might live. It is altogether fitting and proper that we should do this. But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead,

who
have
abov
to a
wor
nor
wha
it
wha
It
rath

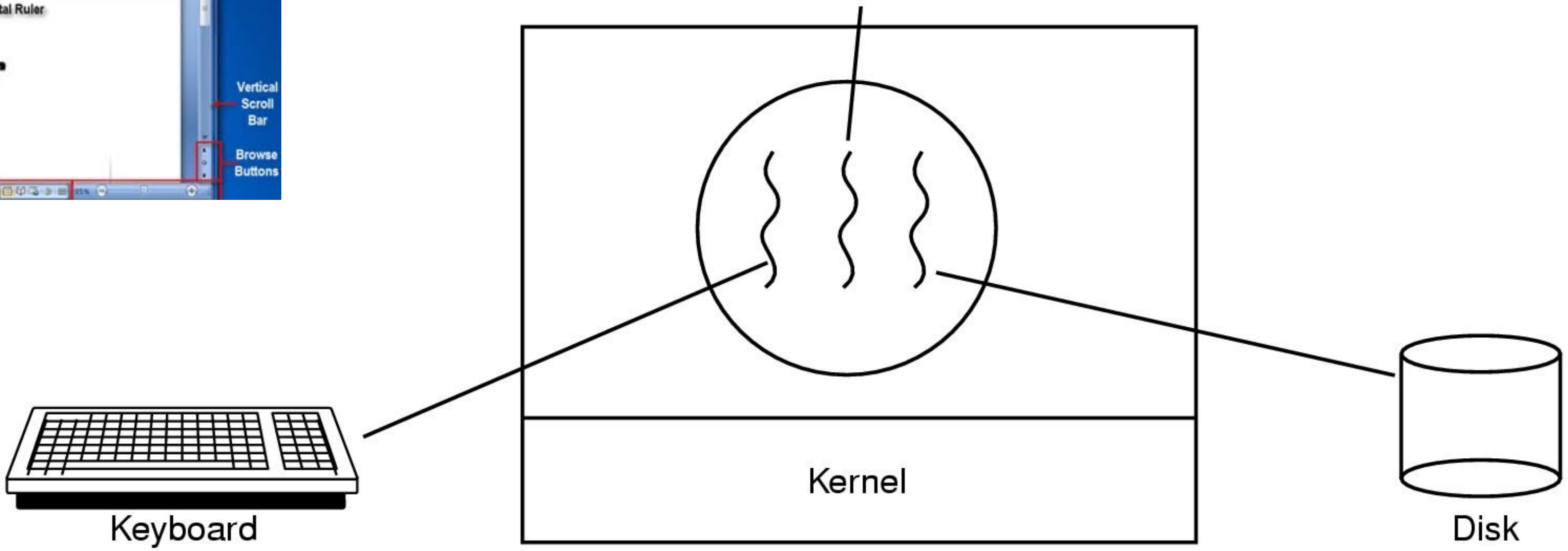


Microsoft Word (446)

Parent Process: launchd (1)
User: rossbach (501)
Process Group: Microsoft Word (446)
% CPU: 0.63
Recent hangs: 0

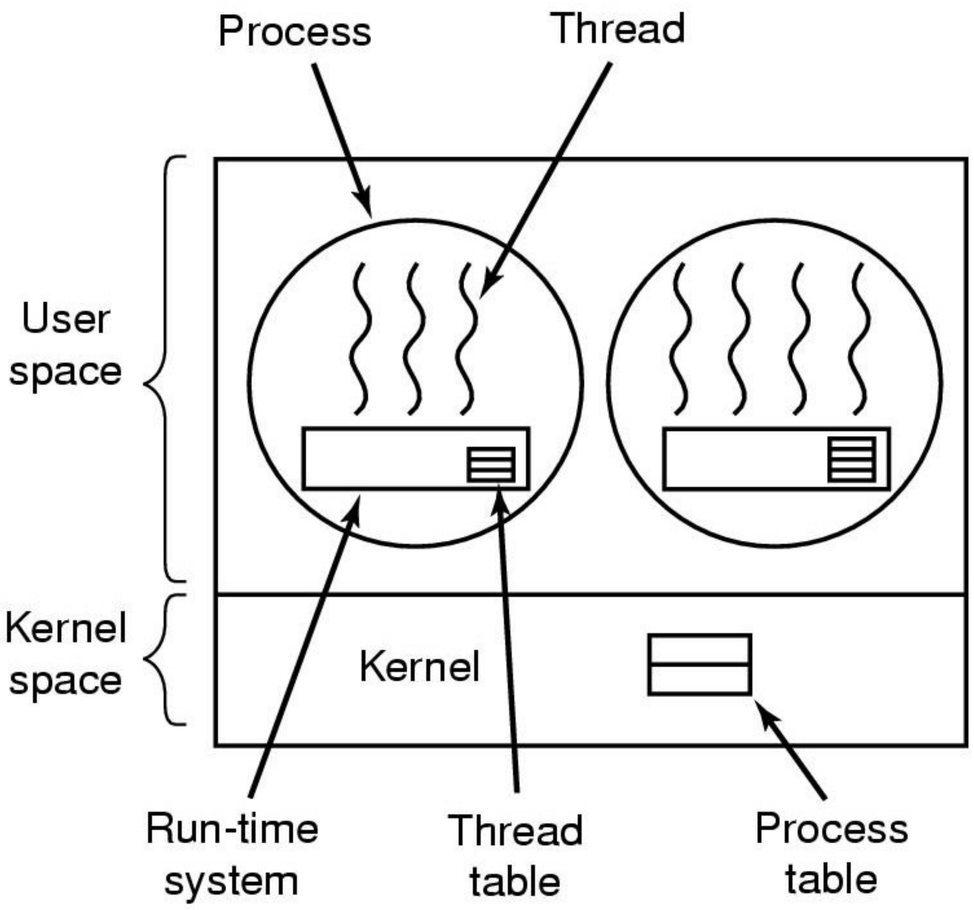
Memory | Statistics | Open Files and Ports

Threads:	15	Page Ins:	467
Ports:	1060	Mach Messages In:	
CPU Time:	46.59	Mach Messages Out:	
Context Switches:	520960	Mach System Calls:	1340968
Faults:	245466	Unix System Calls:	975106
Assertions:	0		



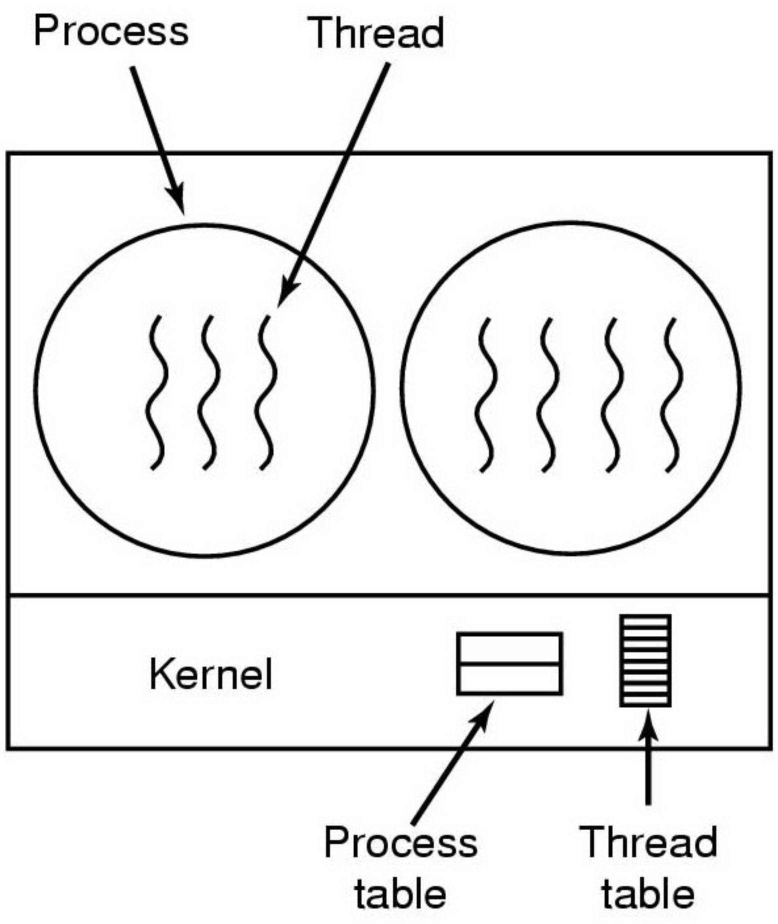
Where to Implement Threads:

User Space



A user-level threads package

Kernel Space

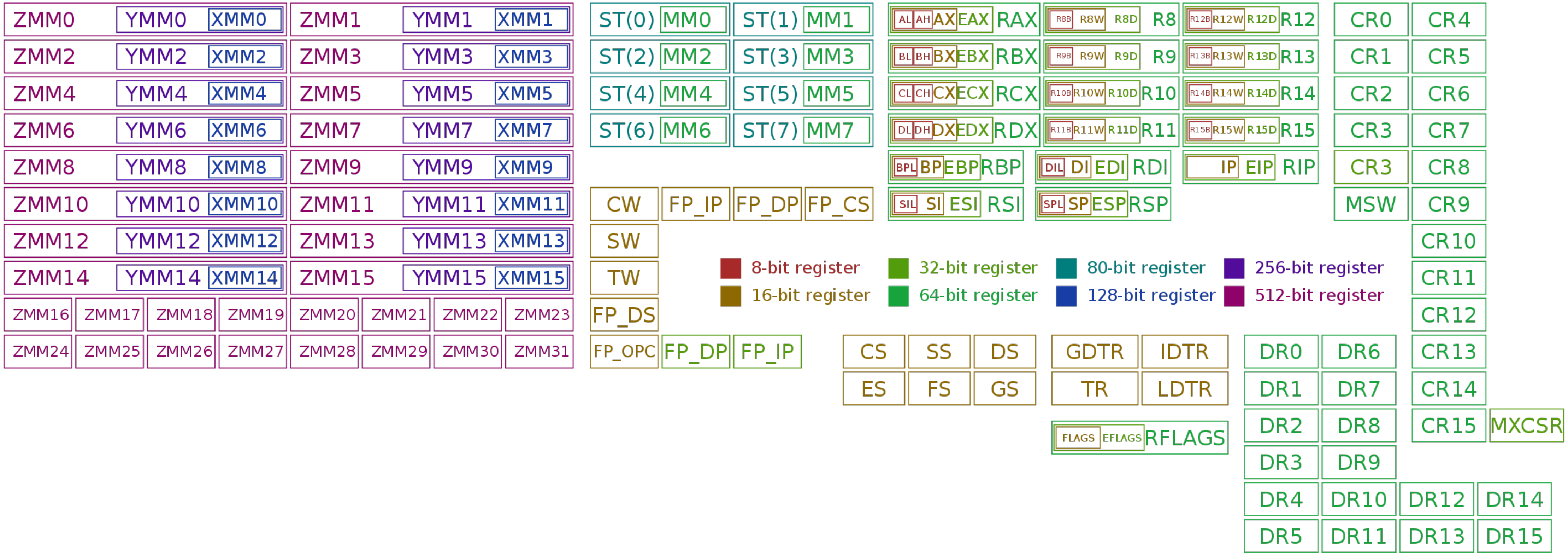


A threads package managed by the kernel

Threads vs Fibers

- Like threads, *just an abstraction* for flow of control
- *Lighter weight* than threads
  - In Windows, just a stack, subset of arch. registers, non-preemptive
  - *Not* just threads without exception support
  - stack management/impl has interplay with exceptions
  - Can be completely exception safe
- ***Takeaway***: diversity of abstractions/containers for execution flows

x86_64 Architectural Registers



ZMM0 YMM0 XMM0 | ZMM1 YMM1 XMM1 | ST(0) MM0 | ST(1) MM1 | AL AH AX EAX RAX | R8B R8W R8D R8 | R12B R12W R12D R12 | CR0 | CR4

ZMM2 YMM2 XMM2 | ZMM3 YMM3 XMM3 | ST(2) MM2 | ST(3) MM3 | BL BH BX EBX RBX | R9B R9W R9D R9 | R13B R13W R13D R13 | CR1 | CR5

ZMM4 YMM4 XMM4 | ZMM5 YMM5 XMM5 | ST(4) MM4 | ST(5) MM5 | CL CH CX ECX RCX | R10B R10W R10D R10 | R14B R14W R14D R14 | CR2 | CR6

ZMM6 YMM6 XMM6 | ZMM7 YMM7 XMM7 | ST(6) MM6 | ST(7) MM7 | DL DH DX EDX RDX | R11B R11W R11D R11 | R15B R15W R15D R15 | CR3 | CR7

ZMM8 YMM8 XMM8 | ZMM9 YMM9 XMM9 | BPL BP EBP RBP | DIL DI EDI RDI | IP EIP RIP | CR3 | CR8

ZMM10 YMM10 XMM10 | ZMM11 YMM11 XMM11 | CW | FP_IP | FP_DP | FP_CS | SIL SI ESI RSI | SPL SP ESP RSP | MSW | CR9

ZMM12 YMM12 XMM12 | ZMM13 YMM13 XMM13 | SW | CR10

ZMM14 YMM14 XMM14 | ZMM15 YMM15 XMM15 | TW | CR11

ZMM16 | ZMM17 | ZMM18 | ZMM19 | ZMM20 | ZMM21 | ZMM22 | ZMM23 | FP_DS | CR12

ZMM24 | ZMM25 | ZMM26 | ZMM27 | ZMM28 | ZMM29 | ZMM30 | ZMM31 | FP_OPC | FP_DP | FP_IP | CS | SS | DS | GDTR | IDTR | DR0 | DR6 | CR13

ES | FS | GS | TR | LDTR | DR1 | DR7 | CR14

FLAGS EFLAGS RFLAGS | DR2 | DR8 | CR15 | MXCSR

DR3 | DR9

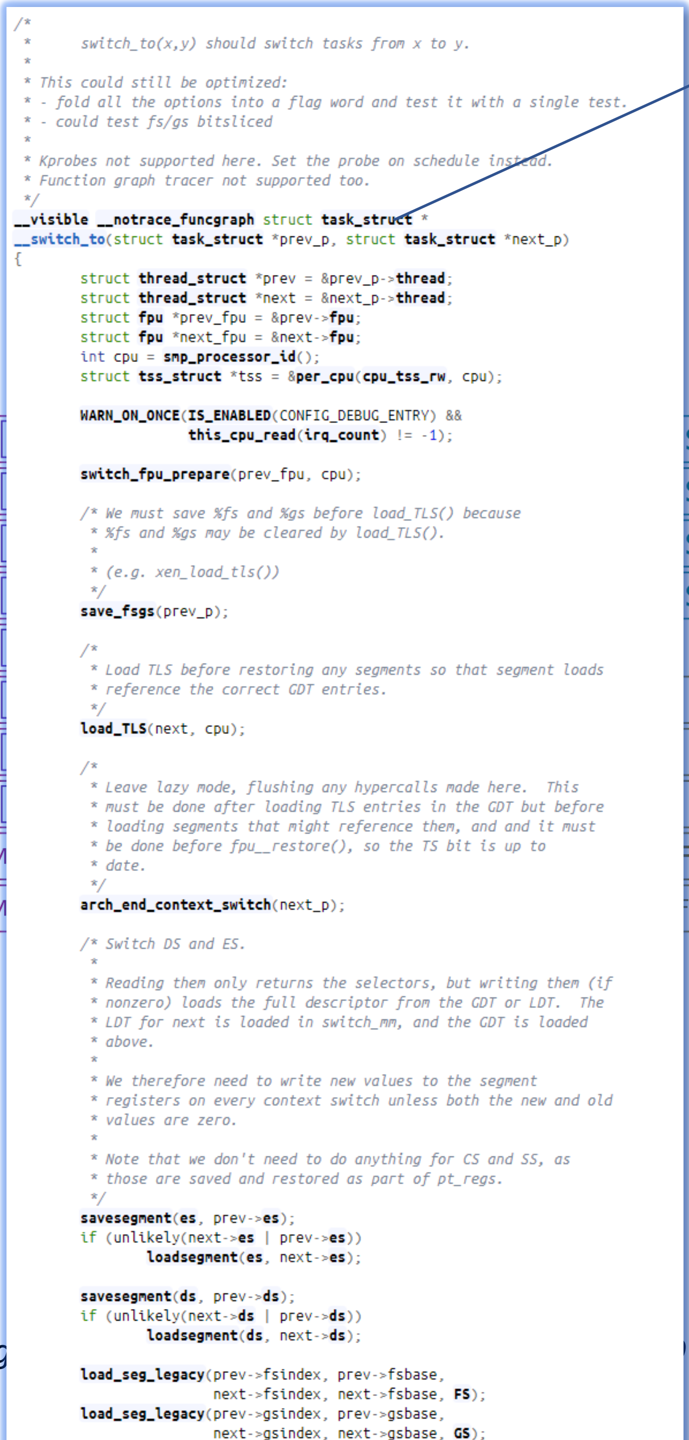
DR4 | DR10 | DR12 | DR14

DR5 | DR11 | DR13 | DR15

🟥 8-bit register
🟫 16-bit register
🟩 32-bit register
🟩 64-bit register
🟦 80-bit register
🟦 128-bit register
🟪 256-bit register
🟪 512-bit register

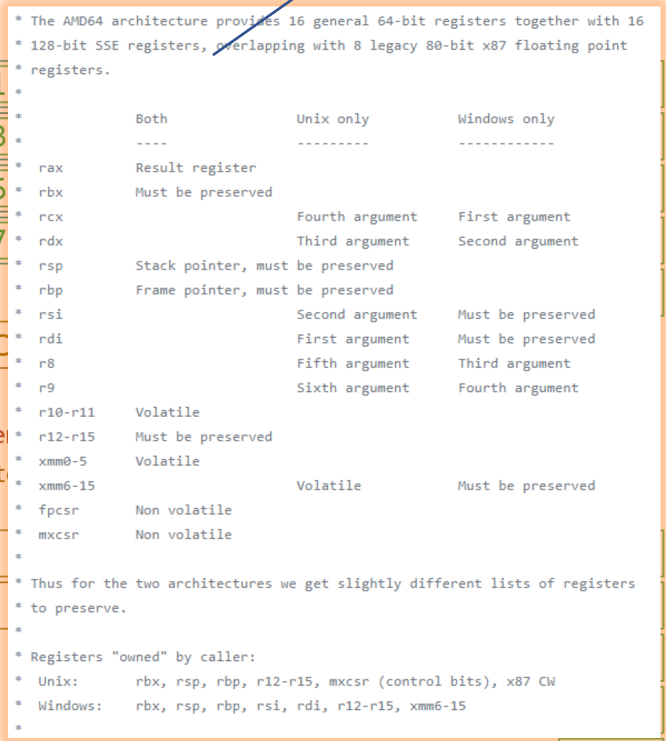
- *Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525*

Linux x86_64 context switch *excerpt*



```
/*
 *      switch_to(x,y) should switch tasks from x to y.
 *
 * This could still be optimized:
 * - fold all the options into a flag word and test it with a single test.
 * - could test fs/gs bitsliced
 *
 * Kprobes not supported here. Set the probe on schedule instead.
 * Function graph tracer not supported too.
 */
__visible __notrace_funcgraph struct task_struct *
__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
        struct thread_struct *prev = &prev_p->thread;
        struct thread_struct *next = &next_p->thread;
        struct fpu *prev_fpu = &prev->fpu;
        struct fpu *next_fpu = &next->fpu;
        int cpu = smp_processor_id();
        struct tss_struct *tss = &per_cpu(cpu_tss_rw, cpu);

        WARN_ON_ONCE(IS_ENABLED(CONFIG_DEBUG_ENTRY) &&
                     this_cpu_read(irq_count) != -1);

        switch_fpu_prepare(prev_fpu, cpu);

        /* We must save %fs and %gs before load_TLS() because
         * %fs and %gs may be cleared by load_TLS().
         *
         * (e.g. xen_load_tls())
         */
        save_fsgs(prev_p);

        /*
         * Load TLS before restoring any segments so that segment loads
         * reference the correct GDT entries.
         */
        load_TLS(next, cpu);

        /*
         * Leave lazy mode, flushing any hypercalls made here.  This
         * must be done after loading TLS entries in the GDT but before
         * loading segments that might reference them, and and it must
         * be done before fpu__restore(), so the TS bit is up to
         * date.
         */
        arch_end_context_switch(next_p);

        /* Switch DS and ES.
         *
         * Reading them only returns the selectors, but writing them (if
         * nonzero) loads the full descriptor from the GDT or LDT.  The
         * LDT for next is loaded in switch_mm, and the GDT is loaded
         * above.
         *
         * We therefore need to write new values to the segment
         * registers on every context switch unless both the new and old
         * values are zero.
         *
         * Note that we don't need to do anything for CS and SS, as
         * those are saved and restored as part of pt_regs.
         */
        savesegment(es, prev->es);
        if (unlikely(next->es | prev->es))
                loadsegment(es, next->es);

        savesegment(ds, prev->ds);
        if (unlikely(next->ds | prev->ds))
                loadsegment(ds, next->ds);

        load_seg_legacy(prev->fsindex, prev->fsbase,
                        next->fsindex, next->fsbase, FS);
        load_seg_legacy(prev->gsindex, prev->gsbase,
                        next->gsindex, next->gsbase, GS);
```

Complete fiber context switch on Unix and Windows



```
* The AMD64 architecture provides 16 general 64-bit registers together with 16
* 128-bit SSE registers, overlapping with 8 legacy 80-bit x87 floating point
* registers.
*
*               Both                 Unix only            Windows only
*               ----                 ---------            ------------
*  rax          Result register
*  rbx          Must be preserved
*  rcx                               Fourth argument      First argument
*  rdx                               Third argument       Second argument
*  rsp          Stack pointer, must be preserved
*  rbp          Frame pointer, must be preserved
*  rsi                               Second argument      Must be preserved
*  rdi                               First argument       Must be preserved
*  r8                                Fifth argument       Third argument
*  r9                                Sixth argument       Fourth argument
*  r10-r11      Volatile
*  r12-r15      Must be preserved
*  xmm0-5       Volatile
*  xmm6-15                           Volatile             Must be preserved
*  fpcsr        Non volatile
*  mxcsr        Non volatile
*
* Thus for the two architectures we get slightly different lists of registers
* to preserve.
*
* Registers "owned" by caller:
*  Unix:        rbx, rsp, rbp, r12-r15, mxcsr (control bits), x87 CW
*  Windows:     rbx, rsp, rbp, rsi, rdi, r12-r15, xmm6-15
*
```

ZMM0 ZMM2 ZMM4 ZMM6 ZMM8 ZMM10 ZMM12 ZMM14
ZMM16 ZMM ZMM24 ZMM

ST(0) MM0 ST(1) MM1
ST(2) MM2 ST(3) MM3
ST(4) MM4 ST(5) MM5
ST(6) MM6 ST(7) MM7

CW FP_IP FP_DP FP_C
SW
TW
P_DS
P_OPC FP_DP FP_IP

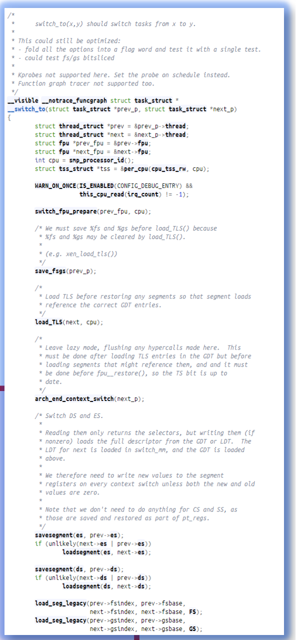
🟥 8-bit registe
🟫 16-bit regist

register
register

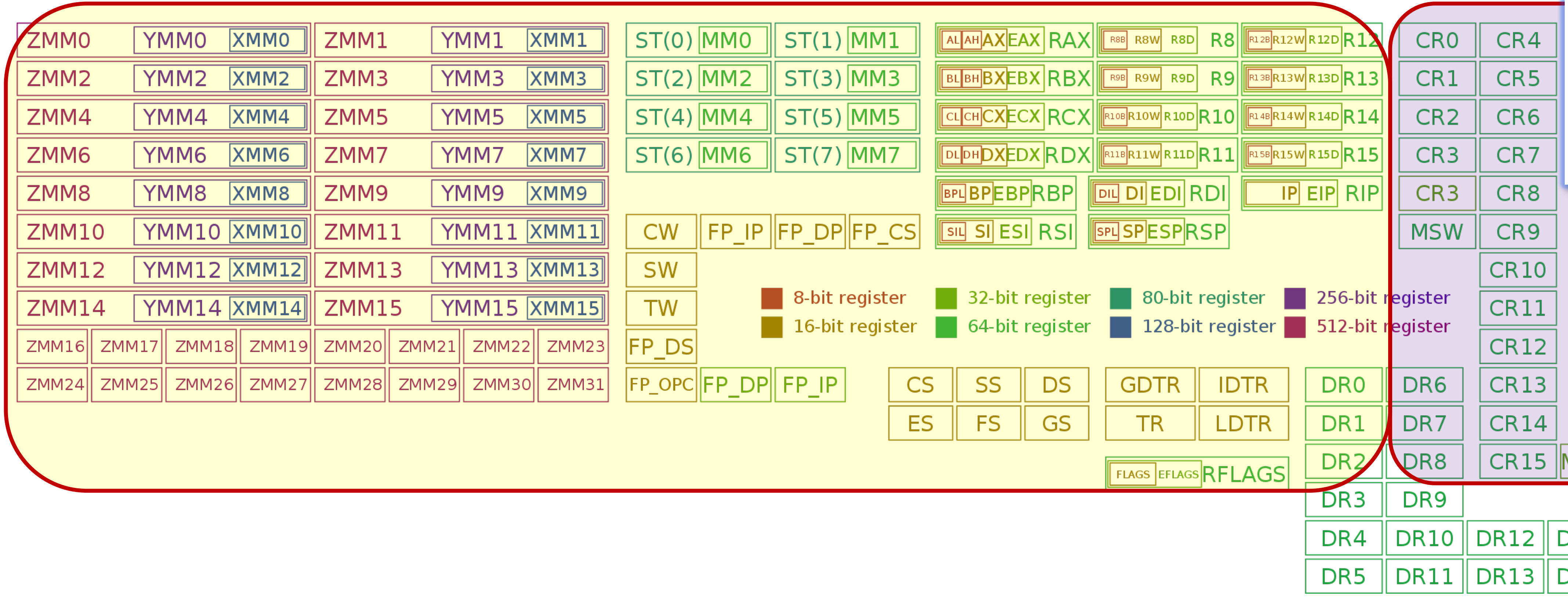
CR0 CR4
CR1 CR5
CR2 CR6
CR3 CR7
CR3 CR8
MSW CR9
CR10
CR11
CR12
DR6 CR13
DR7 CR14
DR8 CR15 MXCSR
DR9
DR4 DR10 DR12 DR14
DR5 DR11 DR13 DR15

- *Reg* *wn work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525*

x86_64 Registers and Threads



```
/*
 *      switch_to(x,y) should switch tasks from x to y.
 *
 * This could still be optimized:
 * - fold all the options into a flag word and test it with a single test.
 * - could test fs/gs bitsliced
 *
 * Kprobes not supported here. Set the probe on schedule instead.
 * Function graph tracer not supported too.
 */
__visible __notrace_funcgraph struct task_struct *
__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
        struct thread_struct *prev = &prev_p->thread;
        struct thread_struct *next = &next_p->thread;
        struct fpu *prev_fpu = &prev->fpu;
        struct fpu *next_fpu = &next->fpu;
        int cpu = smp_processor_id();
        struct tss_struct *tss = &per_cpu(cpu_tss_rw, cpu);

        WARN_ON_ONCE(IS_ENABLED(CONFIG_DEBUG_ENTRY) &&
                     this_cpu_read(irq_count) != -1);

        switch_fpu_prepare(prev_fpu, cpu);

        /* We must save %fs and %gs before load_TLS() because
         * %fs and %gs may be cleared by load_TLS().
         *
         * (e.g. xen_load_tls())
         */
        save_fsgs(prev_p);

        /*
         * Load TLS before restoring any segments so that segment loads
         * reference the correct GDT entries.
         */
        load_TLS(next, cpu);

        /*
         * Leave lazy mode, flushing any hypercalls made here.  This
         * must be done after loading TLS entries in the GDT but before
         * loading segments that might reference them, and and it must
         * be done before fpu__restore(), so the TS bit is up to
         * date.
         */
        arch_end_context_switch(next_p);

        /* Switch DS and ES.
         *
         * Reading them only returns the selectors, but writing them (if
         * nonzero) loads the full descriptor from the GDT or LDT.  The
         * LDT for next is loaded in switch_mm, and the GDT is loaded
         * above.
         *
         * We therefore need to write new values to the segment
         * registers on every context switch unless both the new and old
         * values are zero.
         *
         * Note that we don't need to do anything for CS and SS, as
         * those are saved and restored as part of pt_regs.
         */
        savesegment(es, prev->es);
        if (unlikely(next->es | prev->es))
                loadsegment(es, next->es);

        savesegment(ds, prev->ds);
        if (unlikely(next->ds | prev->ds))
                loadsegment(ds, next->ds);

        load_seg_legacy(prev->fsindex, prev->fsbase,
                        next->fsindex, next->fsbase, FS);
        load_seg_legacy(prev->gsindex, prev->gsbase,
                        next->gsindex, next->gsbase, GS);
```



ZMM0 YMM0 XMM0 | ZMM1 YMM1 XMM1 | ST(0) MM0 | ST(1) MM1 | AL AH AX EAX RAX | R8B R8W R8D R8 | R12B R12W R12D R12 | CR0 | CR4

ZMM2 YMM2 XMM2 | ZMM3 YMM3 XMM3 | ST(2) MM2 | ST(3) MM3 | BL BH BX EBX RBX | R9B R9W R9D R9 | R13B R13W R13D R13 | CR1 | CR5

ZMM4 YMM4 XMM4 | ZMM5 YMM5 XMM5 | ST(4) MM4 | ST(5) MM5 | CL CH CX ECX RCX | R10B R10W R10D R10 | R14B R14W R14D R14 | CR2 | CR6

ZMM6 YMM6 XMM6 | ZMM7 YMM7 XMM7 | ST(6) MM6 | ST(7) MM7 | DL DH DX EDX RDX | R11B R11W R11D R11 | R15B R15W R15D R15 | CR3 | CR7

ZMM8 YMM8 XMM8 | ZMM9 YMM9 XMM9 | BPL BP EBP RBP | DIL DI EDI RDI | IP EIP RIP | CR3 | CR8

ZMM10 YMM10 XMM10 | ZMM11 YMM11 XMM11 | CW | FP_IP | FP_DP | FP_CS | SIL SI ESI RSI | SPL SP ESP RSP | MSW | CR9

ZMM12 YMM12 XMM12 | ZMM13 YMM13 XMM13 | SW | CR10

ZMM14 YMM14 XMM14 | ZMM15 YMM15 XMM15 | TW | CR11

ZMM16 ZMM17 ZMM18 ZMM19 ZMM20 ZMM21 ZMM22 ZMM23 | FP_DS | CR12

ZMM24 ZMM25 ZMM26 ZMM27 ZMM28 ZMM29 ZMM30 ZMM31 | FP_OPC | FP_DP | FP_IP | CS | SS | DS | GDTR | IDTR | DR0 | DR6 | CR13

ES | FS | GS | TR | LDTR | DR1 | DR7 | CR14

FLAGS EFLAGS RFLAGS | DR2 | DR8 | CR15 | MXCSR

DR3 | DR9

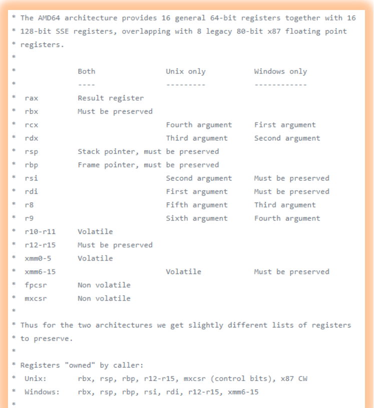
DR4 | DR10 | DR12 | DR14

DR5 | DR11 | DR13 | DR15

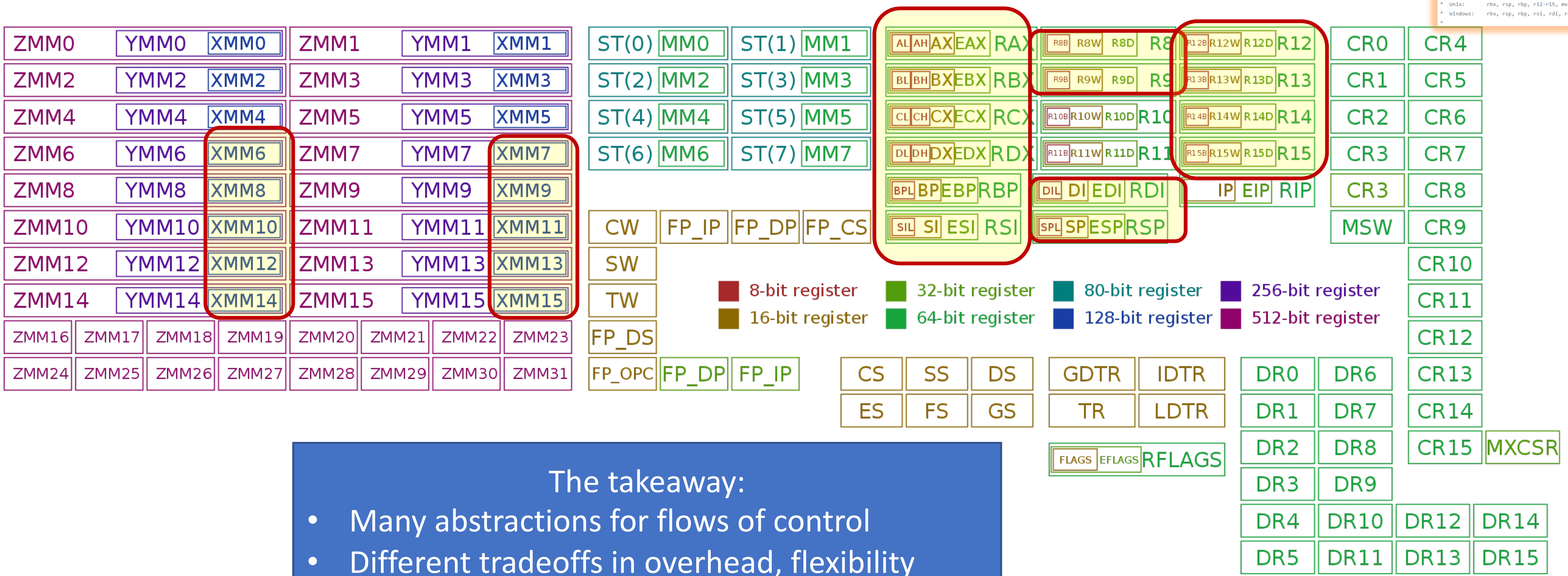
🟥 8-bit register
🟫 16-bit register
🟩 32-bit register
🟩 64-bit register
🟩 80-bit register
🟦 128-bit register
🟪 256-bit register
🟥 512-bit register

- *Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525*

x86_64 Registers and Fibers



```
* The AMD64 architecture provides 16 general 64-bit registers together with 16
* 128-bit SSE registers, overlapping with 8 legacy 80-bit x87 floating point
* registers.
*
*              Both                Unix only           Windows only
*              ----                ---------           ------------
*  rax         Result register
*  rbx         Must be preserved
*  rcx                             Fourth argument     First argument
*  rdx                             Third argument      Second argument
*  rsp         Stack pointer, must be preserved
*  rbp         Frame pointer, must be preserved
*  rsi                             Second argument     Must be preserved
*  rdi                             First argument      Must be preserved
*  r8                              Fifth argument      Third argument
*  r9                              Sixth argument      Fourth argument
*  r10-r11     Volatile
*  r12-r15     Must be preserved
*  xmm0-5      Volatile
*  xmm6-15                         Volatile            Must be preserved
*  fpcsr       Non volatile
*  mxcsr       Non volatile
*
* Thus for the two architectures we get slightly different lists of registers
* to preserve.
*
* Registers "owned" by caller:
*  Unix:       rbx, rsp, rbp, r12-r15, mxcsr (control bits), x87 CW
*  Windows:    rbx, rsp, rbp, rsi, rdi, r12-r15, xmm6-15
*
```



ZMM0 YMM0 XMM0 | ZMM1 YMM1 XMM1 | ST(0) MM0 | ST(1) MM1 | AL AH AX EAX RAX | R8B R8W R8D R8 | R12B R12W R12D R12 | CR0 | CR4

ZMM2 YMM2 XMM2 | ZMM3 YMM3 XMM3 | ST(2) MM2 | ST(3) MM3 | BL BH BX EBX RBX | R9B R9W R9D R9 | R13B R13W R13D R13 | CR1 | CR5

ZMM4 YMM4 XMM4 | ZMM5 YMM5 XMM5 | ST(4) MM4 | ST(5) MM5 | CL CH CX ECX RCX | R10B R10W R10D R10 | R14B R14W R14D R14 | CR2 | CR6

ZMM6 YMM6 XMM6 | ZMM7 YMM7 XMM7 | ST(6) MM6 | ST(7) MM7 | DL DH DX EDX RDX | R11B R11W R11D R11 | R15B R15W R15D R15 | CR3 | CR7

ZMM8 YMM8 XMM8 | ZMM9 YMM9 XMM9 | BPL BP EBP RBP | DIL DI EDI RDI | IP EIP RIP | CR3 | CR8

ZMM10 YMM10 XMM10 | ZMM11 YMM11 XMM11 | CW | FP_IP | FP_DP | FP_CS | SIL SI ESI RSI | SPL SP ESP RSP | MSW | CR9

ZMM12 YMM12 XMM12 | ZMM13 YMM13 XMM13 | SW | CR10

ZMM14 YMM14 XMM14 | ZMM15 YMM15 XMM15 | TW | CR11

ZMM16 ZMM17 ZMM18 ZMM19 ZMM20 ZMM21 ZMM22 ZMM23 | FP_DS | CR12

ZMM24 ZMM25 ZMM26 ZMM27 ZMM28 ZMM29 ZMM30 ZMM31 | FP_OPC | FP_DP | FP_IP | CS SS DS | GDTR IDTR | DR0 DR6 | CR13

ES FS GS | TR LDTR | DR1 DR7 | CR14

FLAGS EFLAGS RFLAGS | DR2 DR8 | CR15 | MXCSR

DR3 DR9

DR4 DR10 DR12 DR14

DR5 DR11 DR13 DR15

🟥 8-bit register	🟩 32-bit register	🟦 80-bit register	🟪 256-bit register
🟫 16-bit register	🟩 64-bit register	🟦 128-bit register	🟪 512-bit register

The takeaway:

- Many abstractions for flows of control
- Different tradeoffs in overhead, flexibility
- Matters for concurrency: exercised heavily

- *Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525*

Pthreads

- POSIX standard thread model,
- Specifies the API and call semantics.
- Popular – most thread libraries are Pthreads-compatible

Preliminaries

- Include `pthread.h` in the main file
- Compile program with `–lpthread`
  - `gcc –o test test.c –lpthread`
  - may not report compilation errors otherwise but calls will fail
- Good idea to check return values on common functions

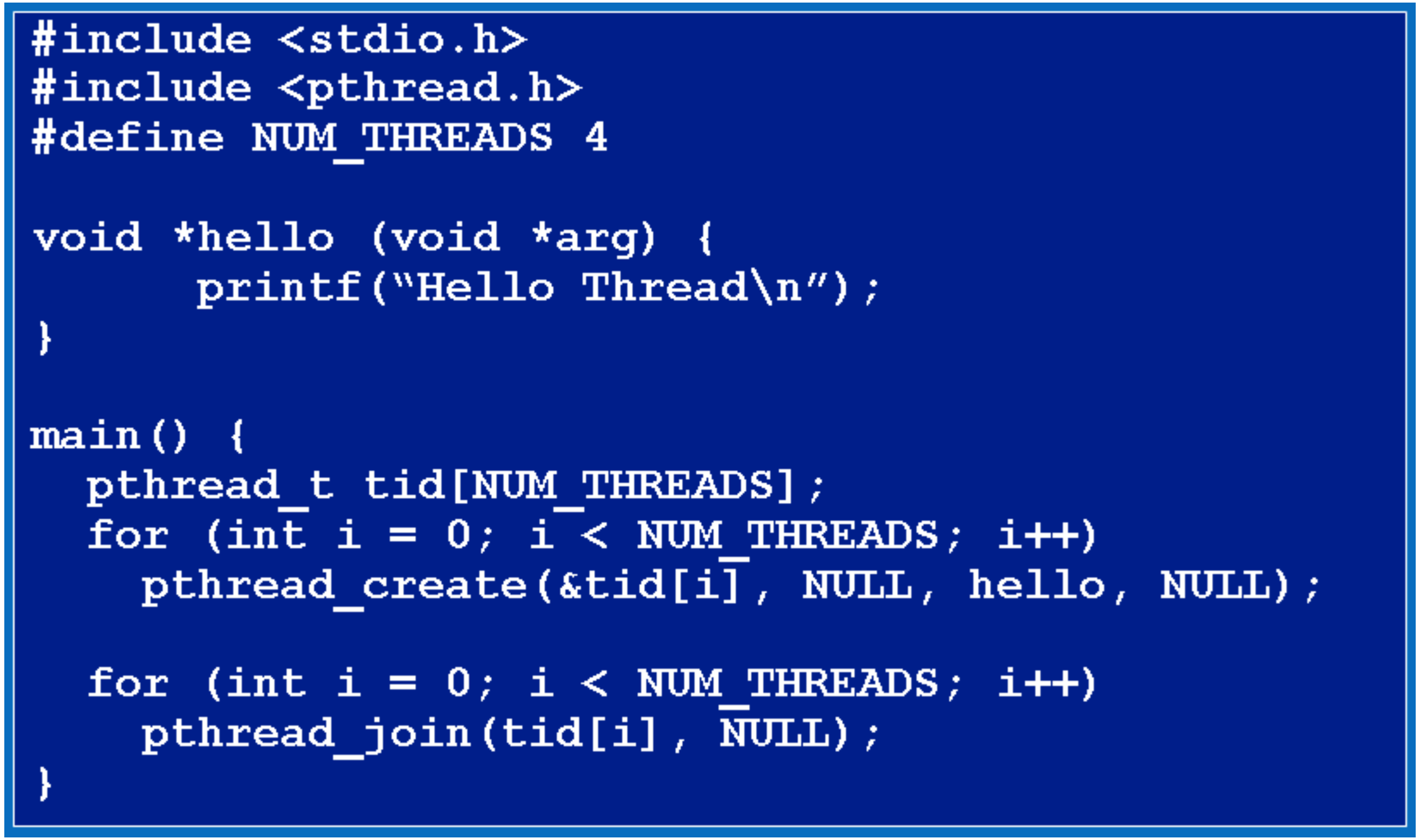
Thread creation

- Types: `pthread_t` – type of a thread
- Some calls:

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void *),
                   void *arg);
int pthread_join(pthread_t thread, void **status);
int pthread_detach();
void pthread_exit();
```

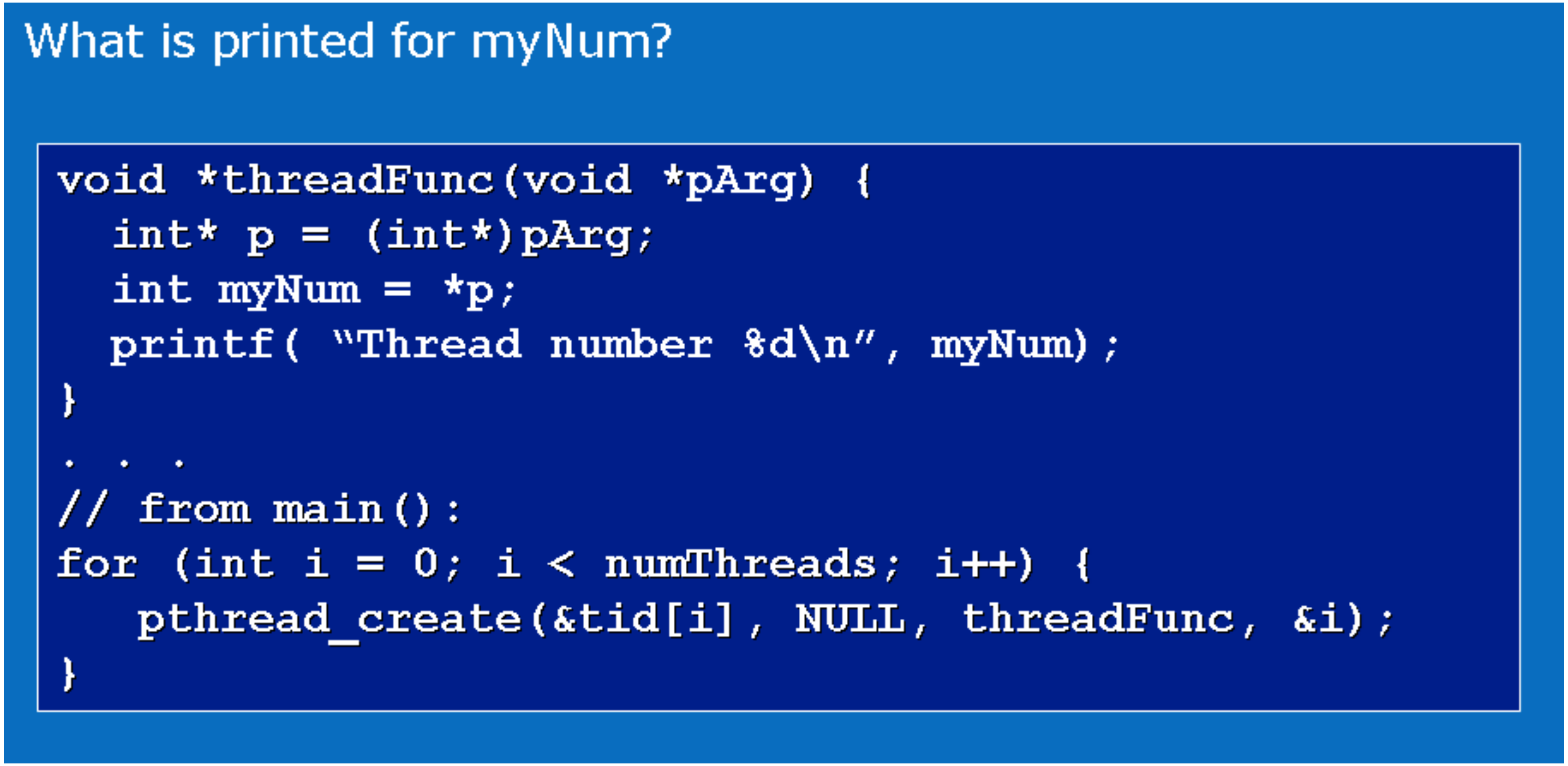
  - No explicit parent/child model, except main thread holds process info
  - Call `pthread_exit` in main, don't just fall through;
  - When do you need `pthread_join ?`
    - `status` = exit value returned by joinable thread
  - Detached threads are those which cannot be joined (can also set this at creation)

Creating multiple threads



```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
      printf(“Hello Thread\n”);
}

main() {
  pthread_t tid[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, hello, NULL);

  for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
```

Can you find the bug here?



What is printed for myNum?

```
void *threadFunc(void *pArg) {
  int* p = (int*)pArg;
  int myNum = *p;
  printf( “Thread number %d\n”, myNum);
}
. . .
// from main():
for (int i = 0; i < numThreads; i++) {
   pthread_create(&tid[i], NULL, threadFunc, &i);
}
```

Pthread Mutexes

- Type: `pthread_mutex_t`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Attributes: for shared mutexes/condition vars among processes, for priority inheritance, etc.
    - use defaults
- Important: Mutex scope must be visible to all threads!

Pthread Spinlock

- Type: `pthread_spinlock_t`

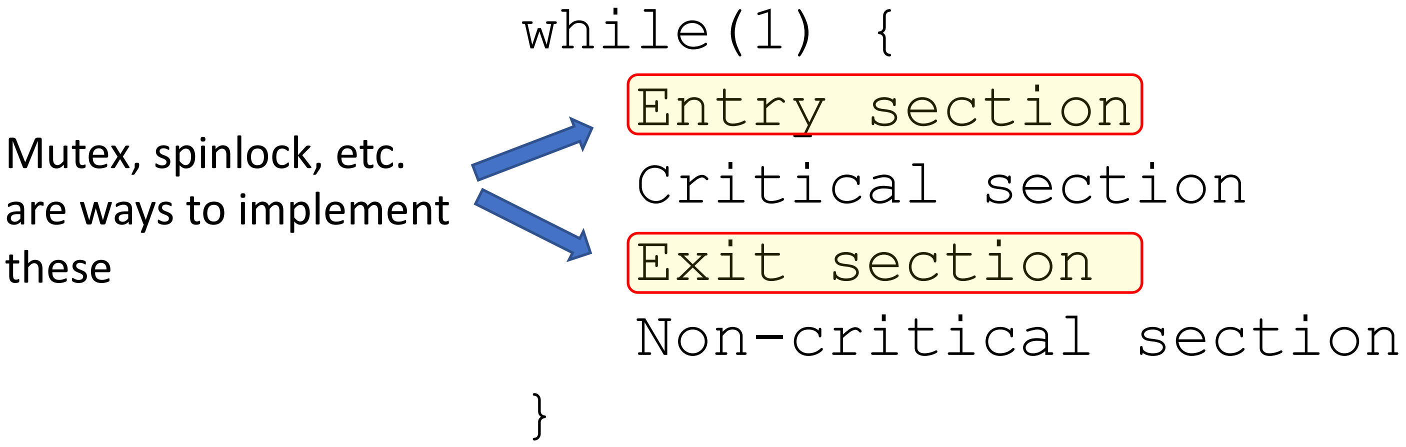
```
int pthread_spinlock_init(pthread_spinlock_t *lock);
int pthread_spinlock_destroy(pthread_spinlock_t *lock);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

Wait…what's the difference?

```
int pthread_mutex_init(pthread_mutex_t *mutex,…);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Review: mutual exclusion model

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region



Mutex, spinlock, etc. are ways to implement these

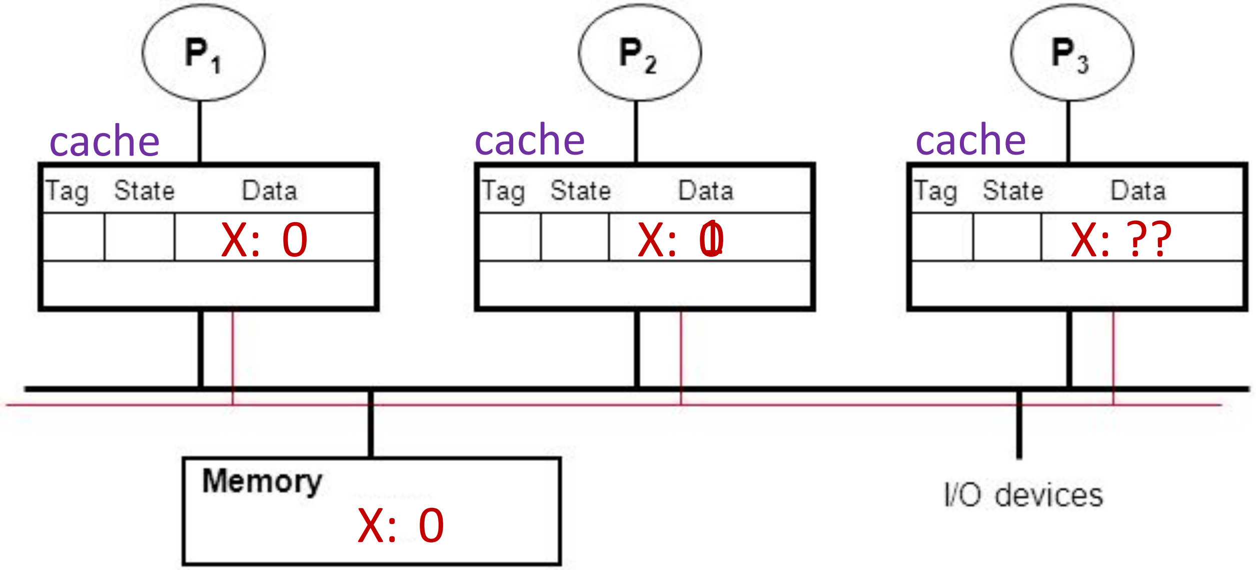
```
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

Multiprocessor Cache Coherence

Physics | Concurrency

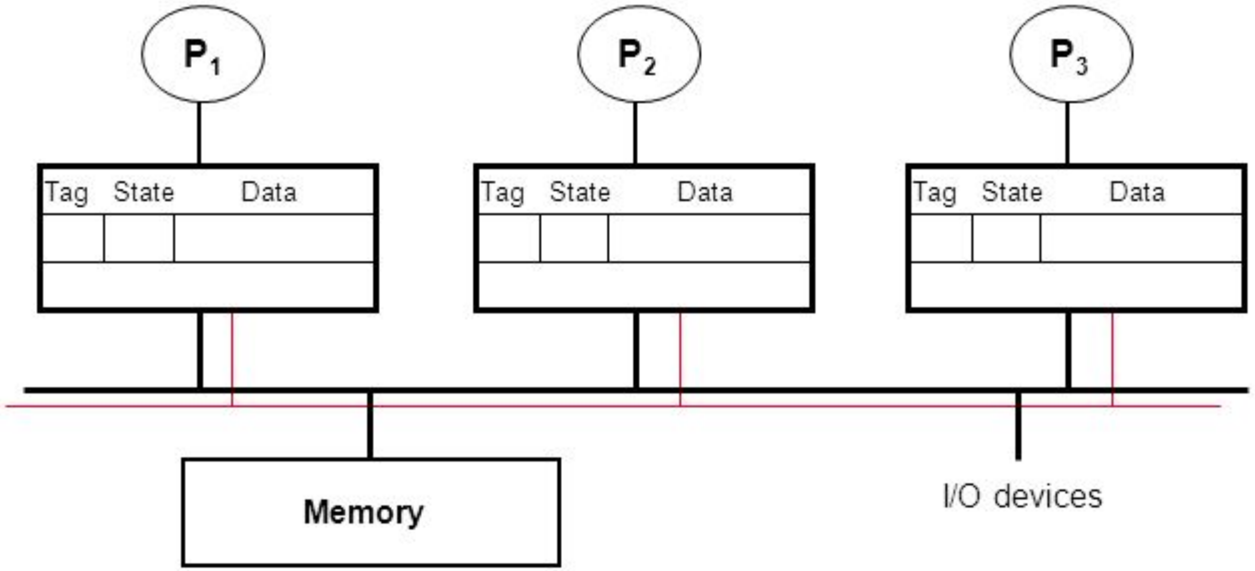
$F = ma$ ~ *coherence*

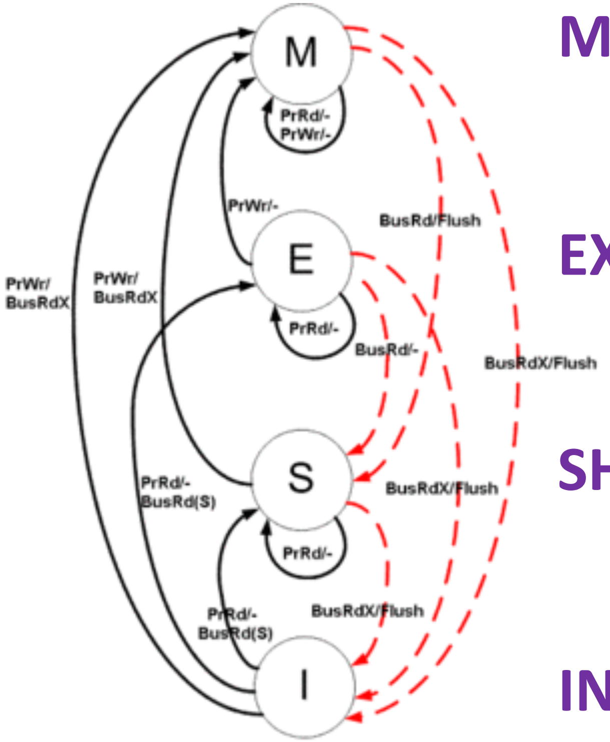
Multiprocessor Cache Coherence



- P1: read X
- P2: read X
- P2: X++
- P3: read X

Multiprocessor Cache Coherence





MODIFIED

EXCLUSIVE

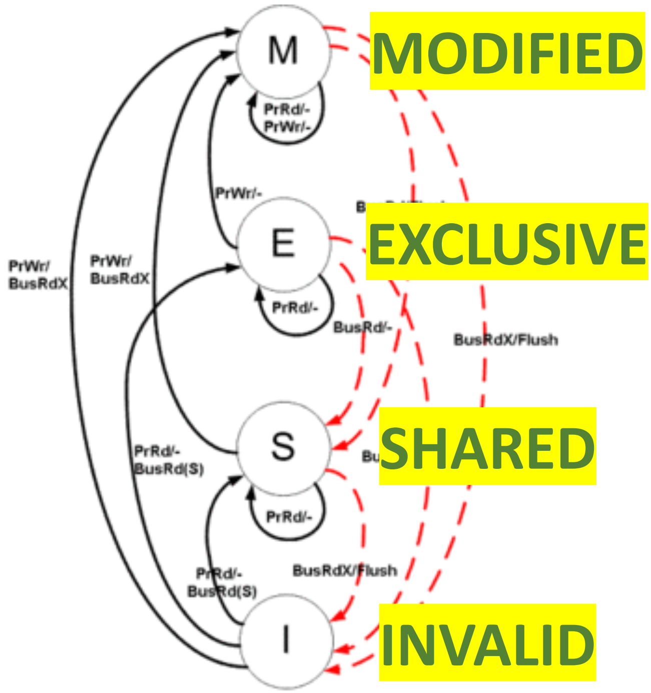
SHARED

INVALID

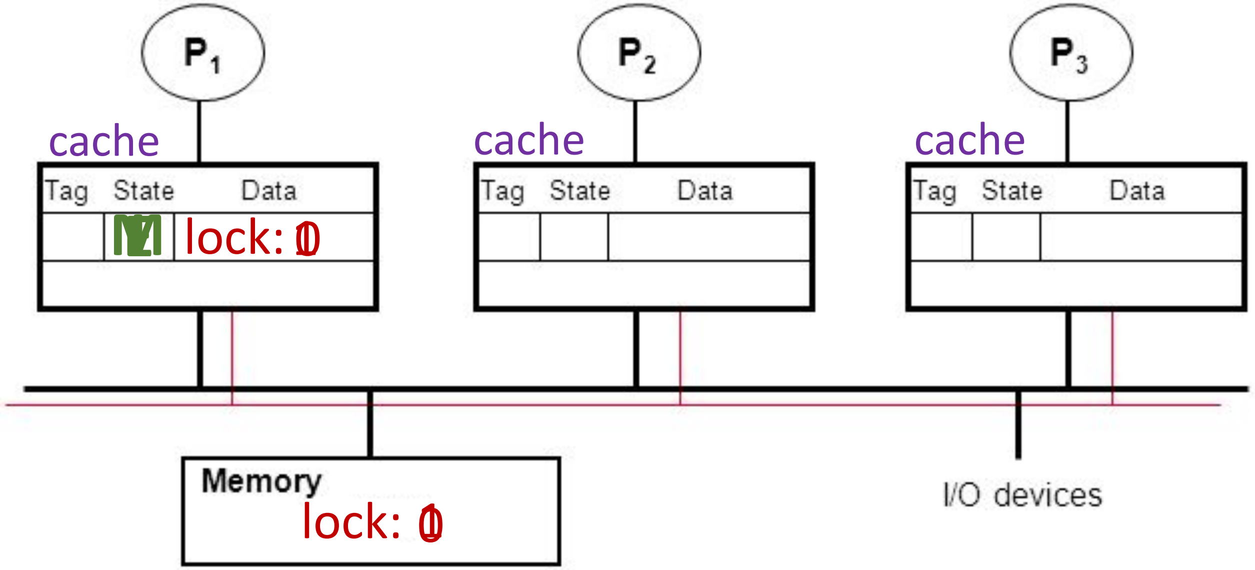
Each cache line has a state (M, E, S, I)

- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
- Reads → ‘S’ → multiple copies possible
- Write → ‘M’ → single copy → lots of cache coherence traffic

Cache Coherence: single-thread



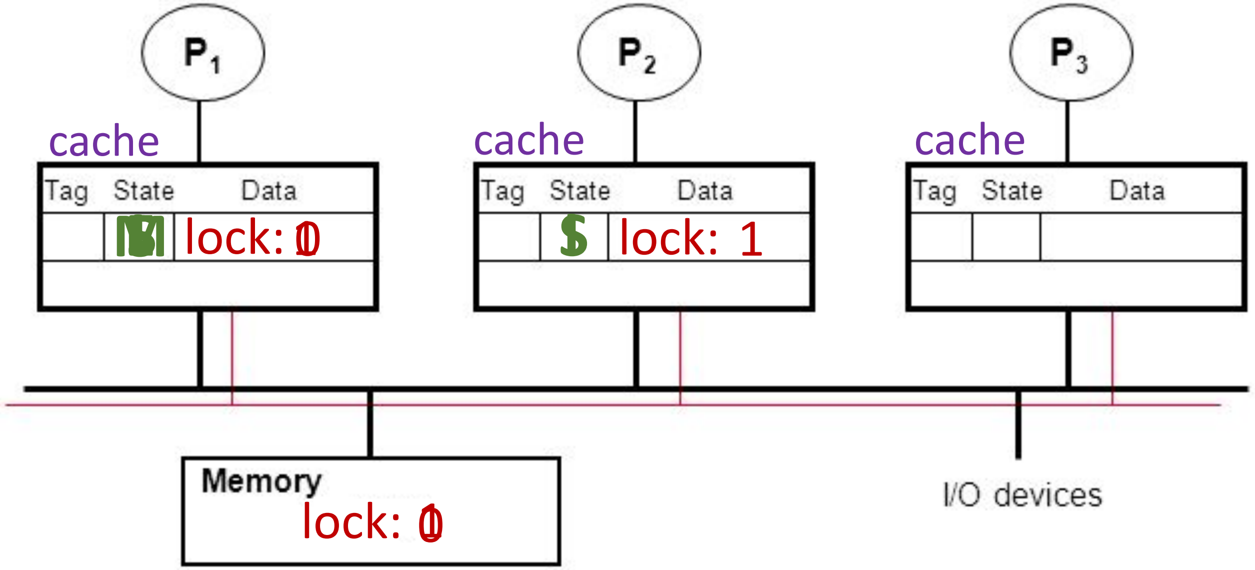
[cache eviction]

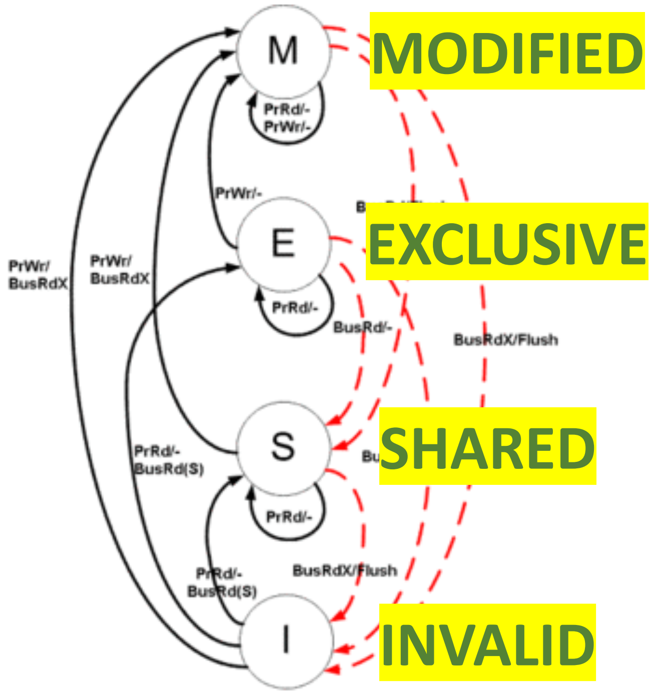


P1

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





P1

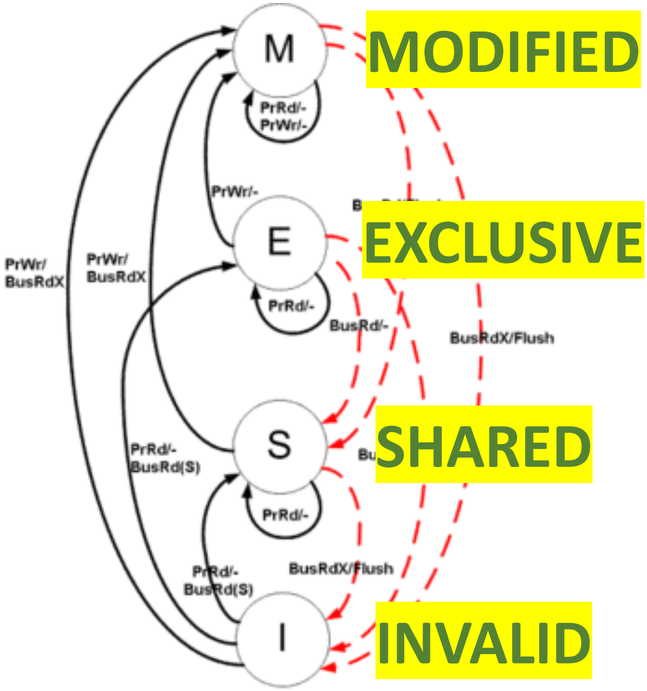
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

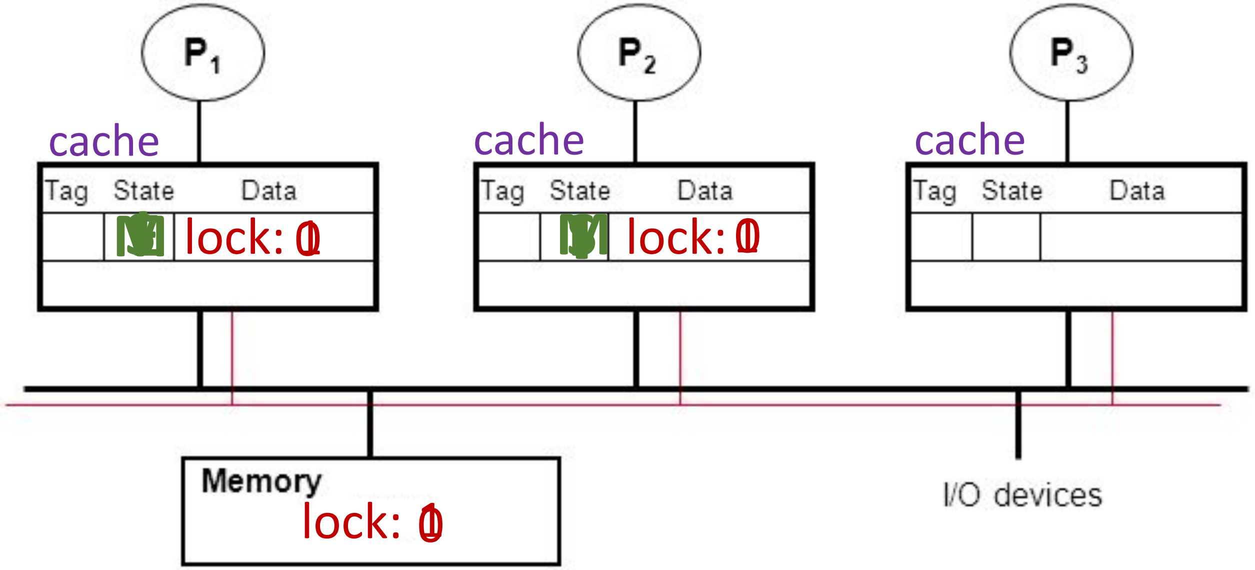
SAFE!

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





NOT SAFE!

P1

P2

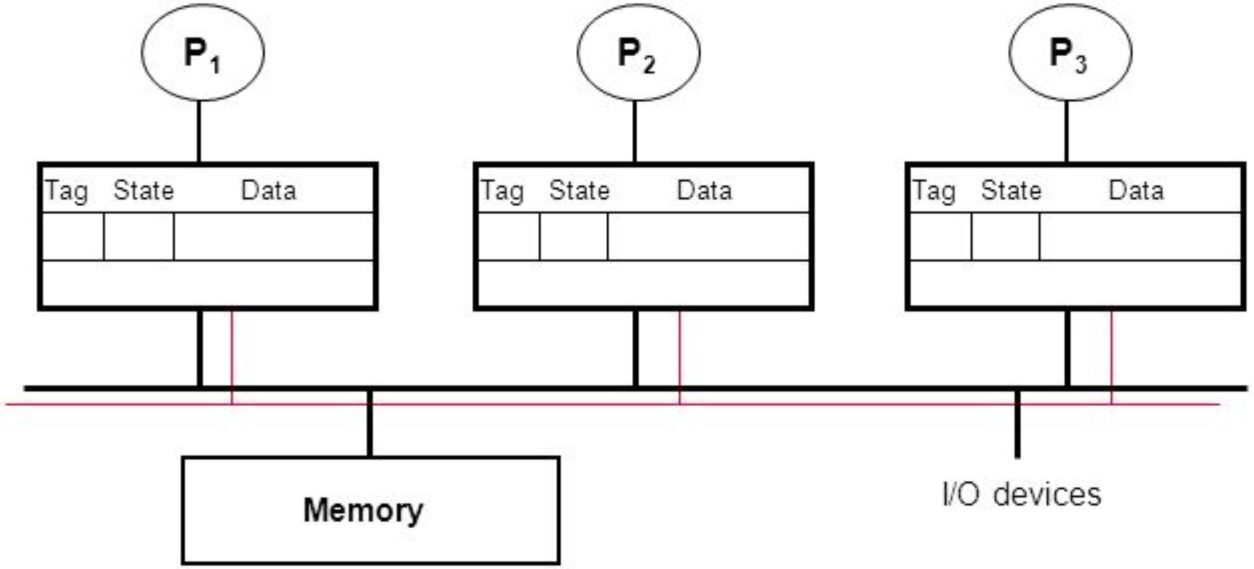
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Read-Modify-Write (RMW)

- Implementing locks requires read-modify-write operations
- Required effect is:
  - An atomic and isolated action
    1. read memory location **AND**
    2. write a new value to the location
  - RMW is *very tricky* in multi-processors
  - Cache coherence alone doesn't solve it

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
      test R0
      bnz try
      store lock, 1
}
```



Essence of HW-supported RMW

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
      test R0
      bnz try
      store lock, 1
}
```

Make this into a single
(atomic hardware instruction)

HW Support for Read-Modify-Write (RMW)

Test & Set	CAS	Exchange, locked increment/decrement,	LLSC: load-linked store-conditional
Most architectures	Many architectures	x86	PPC, Alpha, MIPS

`int TST(addr) {`
`  atomic {`
`    ret = *addr;`
`    if(!*addr)`
`      *addr = 1;`
`    return ret;`
`  }`
`}`
	`bool cas(addr, old, new) {`
	`  atomic {`
	`    if(*addr == old) {`
	`      *addr = new;`
	`      return true;`
	`    }`
	`    return false;`
	`  }`
	`}`
		`int XCHG(addr, val) {`
		`  atomic {`
		`    ret = *addr;`
		`    *addr = val;`
		`    return ret;`
		`  }`
		`}`
			`bool LLSC(addr, val) {`
			` ret = *addr;`
			` atomic {`
			`   if(*addr == ret) {`
			`     *addr = val;`
			`     return true;`
			`  }`
			`  return false;`
			`}`

```
void CAS_lock(lock) {
  while(CAS(&lock, 0, 1) != true);
}
```

HW Support for RMW: LL-SC

LLSC: load-linked store-conditional

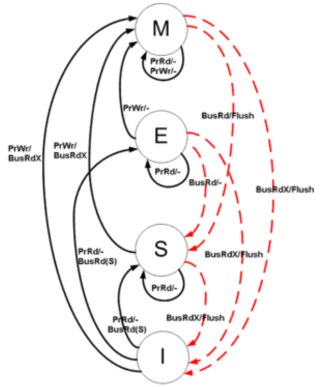
PPC, Alpha, MIPS

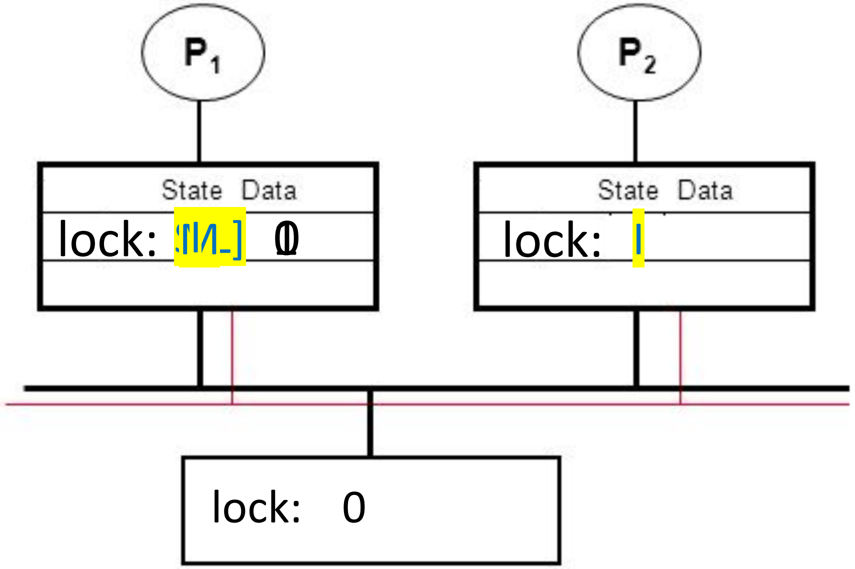
```
bool LLSC(addr, val) {
 ret = *addr;
 atomic {
   if(*addr == ret) {
     *addr = val;
     return true;
  }
  return false;
}
```

```
void LLSC_lock(lock) {
  while(1) {
    old = load-linked(lock);
    if(old == 0 && store-cond(lock, 1))
      return;
  }
}
```

- load-linked is a load that is “linked” to a subsequent store-conditional
- Store-conditional only succeeds if value from linked-load is unchanged

LLSC Lock Action Zone





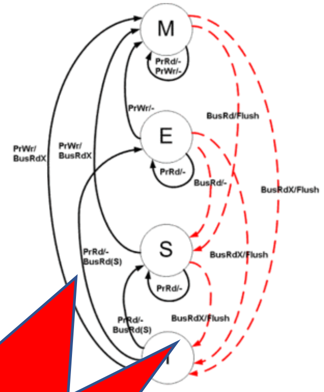
P1

```
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

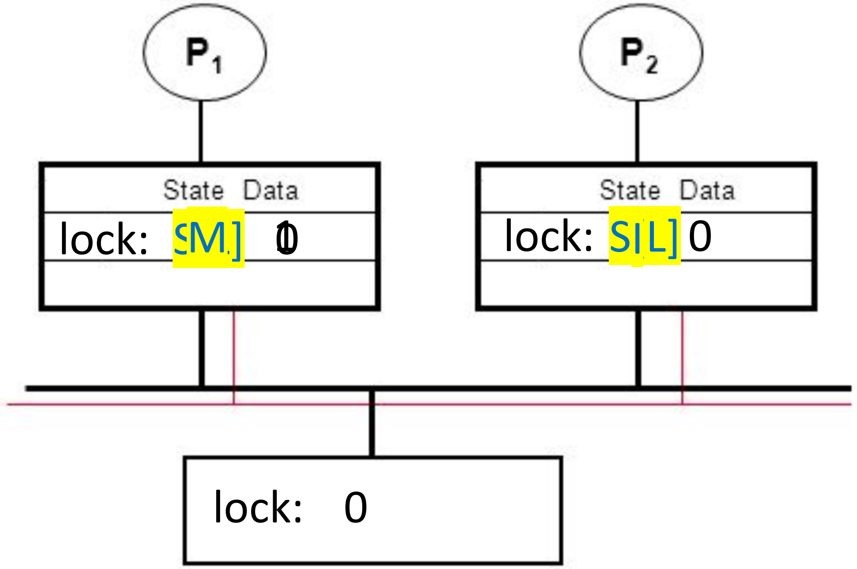
P2

```
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone II



Store conditional fails



P1

```
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

P2

```
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
while (test&set(lock) == 1)
   ; //spin
}
```

(test & set ~ CAS ~ LLSC)

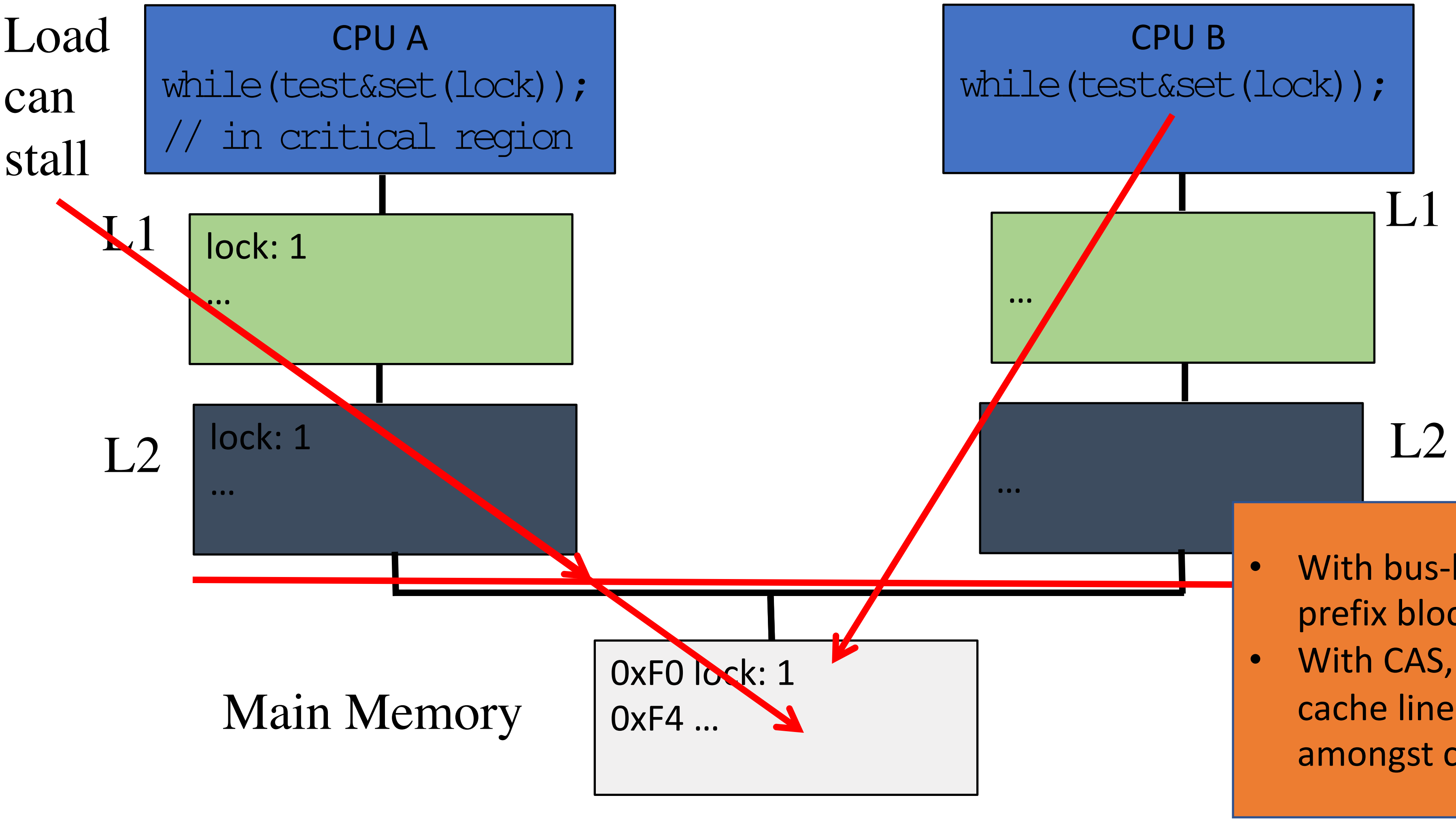
```
Lock::Release() {
    *lock = 0;
}
```

- What is the problem with this?
  - A. CPU usage B. Memory usage C. Lock::Acquire() latency
  - D. Memory bus usage E. Does not work

Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

What happens to lock variable’s cache line when different cpu’s contend?



```
while(test&set(lock));
// in critical region
```

```
while(test&set(lock));
```

- With bus-locking, lock prefix blocks *everyone*
- With CAS, LL-SC, cache line cache line “ping pongs” amongst contenders

TTS: Reducing busy wait contention

Test&Set

```
Lock::Acquire() {
while (test&set(lock) == 1);
}
```

Busy-wait on in-memory copy

```
Lock::Release() {
    *lock = 0;
}
```

Test&Test&Set

```
Lock::Acquire() {
while(1) {
  while (*lock == 1) ; // spin just reading
  if (test&set(lock) == 0)  break;
}
```

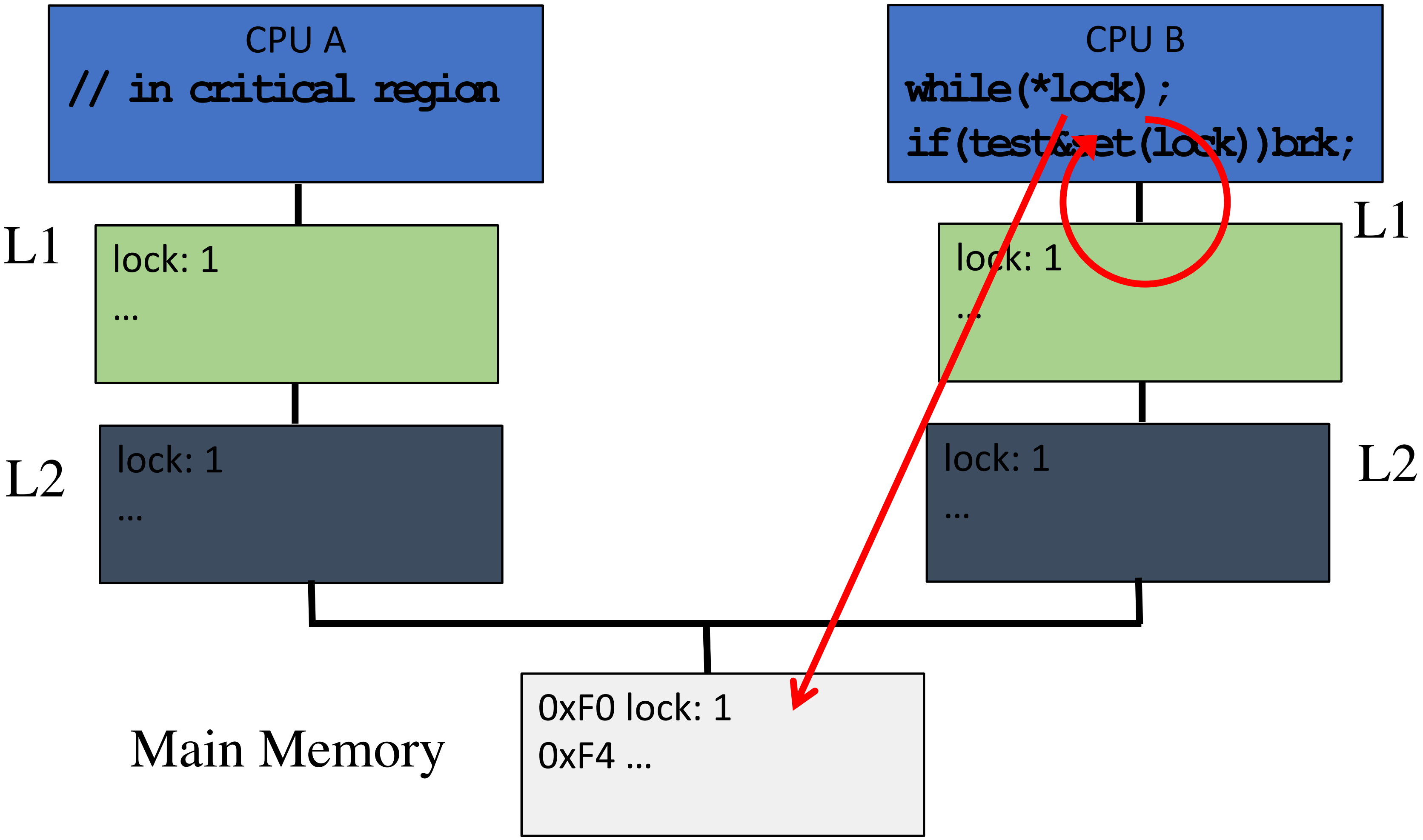
Busy-wait on cached copy

```
Lock::Release() {
*lock = 0;
}
```

- What is the problem with this?
  - A. CPU usage B. Memory usage C. Lock::Acquire() latency
  - D. Memory bus usage E. Does not work

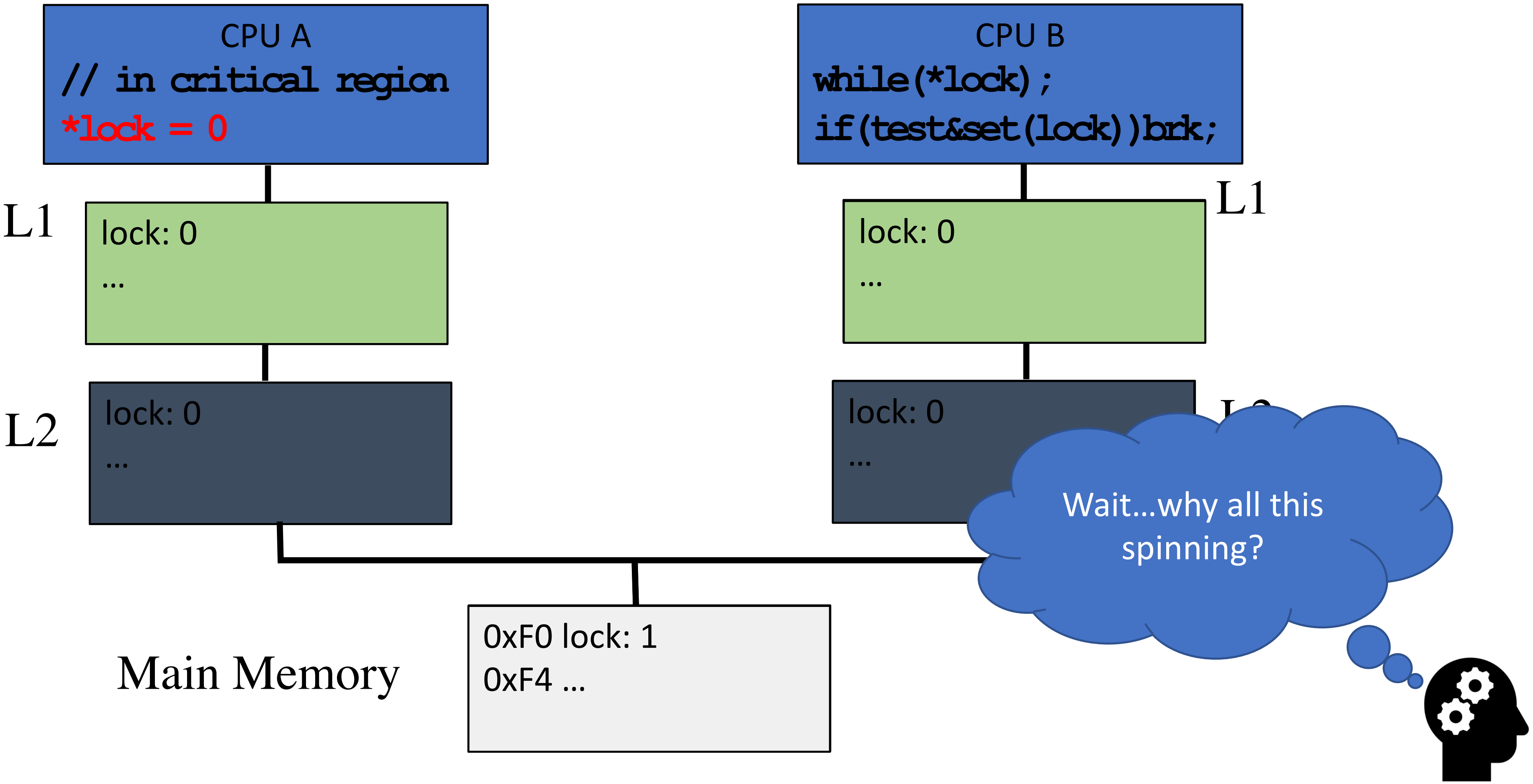
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



Test & Test & Set with Memory Hierarchies

What happens to lock variable’s cache line when different cpu’s contend for the same lock?



```
// in critical region
*lock = 0
```

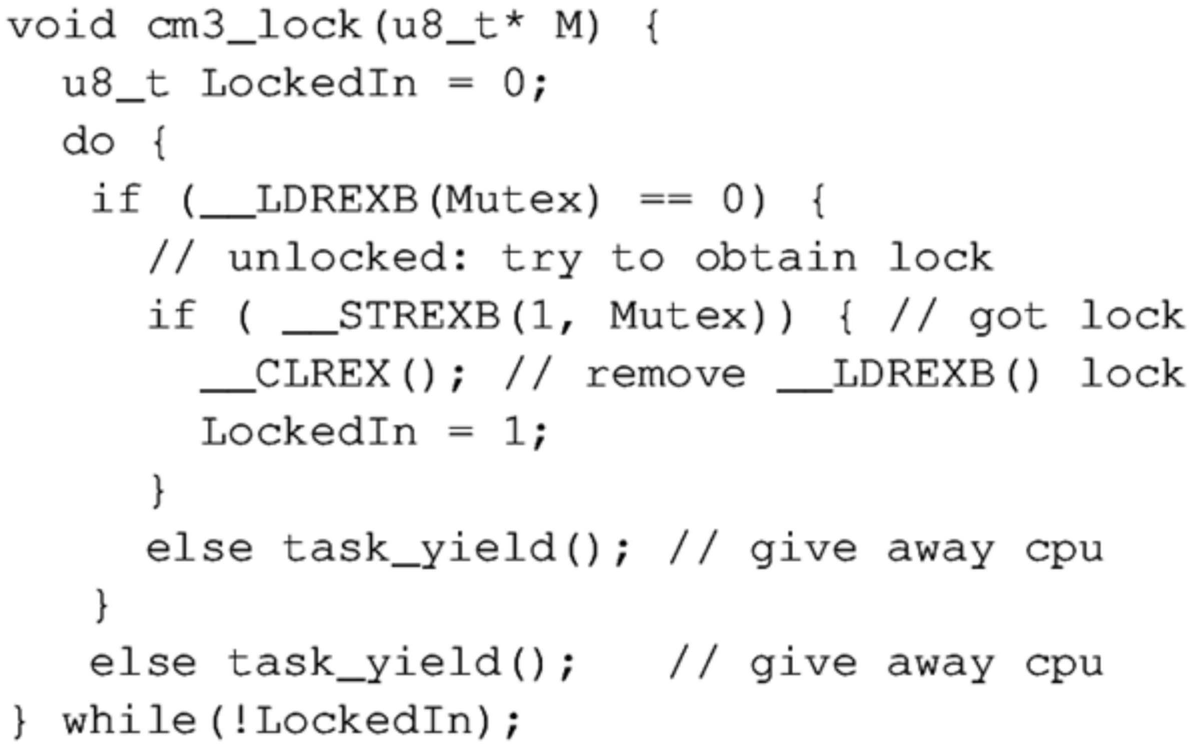
```
while(*lock);
if(test&set(lock))brk;
```

How can we improve over busy-wait?

```
Lock::Acquire() {
while(1) {
  while (*lock == 1) ; // spin just reading
  if (test&set(lock) == 0)  break;
}
```

Mutex

- Same abstraction as spinlock
- But is a “blocking” primitive
  - Lock available → same behavior
  - Lock held → yield/block
- Many ways to yield
- Simplest case of semaphore



```
void cm3_lock(u8_t* M) {
  u8_t LockedIn = 0;
  do {
   if (__LDREXB(Mutex) == 0) {
     // unlocked: try to obtain lock
     if ( __STREXB(1, Mutex)) { // got lock
       __CLREX(); // remove __LDREXB() lock
       LockedIn = 1;
     }
     else task_yield(); // give away cpu
   }
   else task_yield();   // give away cpu
} while(!LockedIn);
```

- Is it better to use a spinlock or mutex on a uni-processor?
- Is it better to use a spinlock or mutex on a multi-processor?
- How do you choose between spinlock/mutex on a multi-processor?

Priority Inversion

A(prio-0) → enter(l);

B(prio-100) → enter(l); → must wait.

Solution?

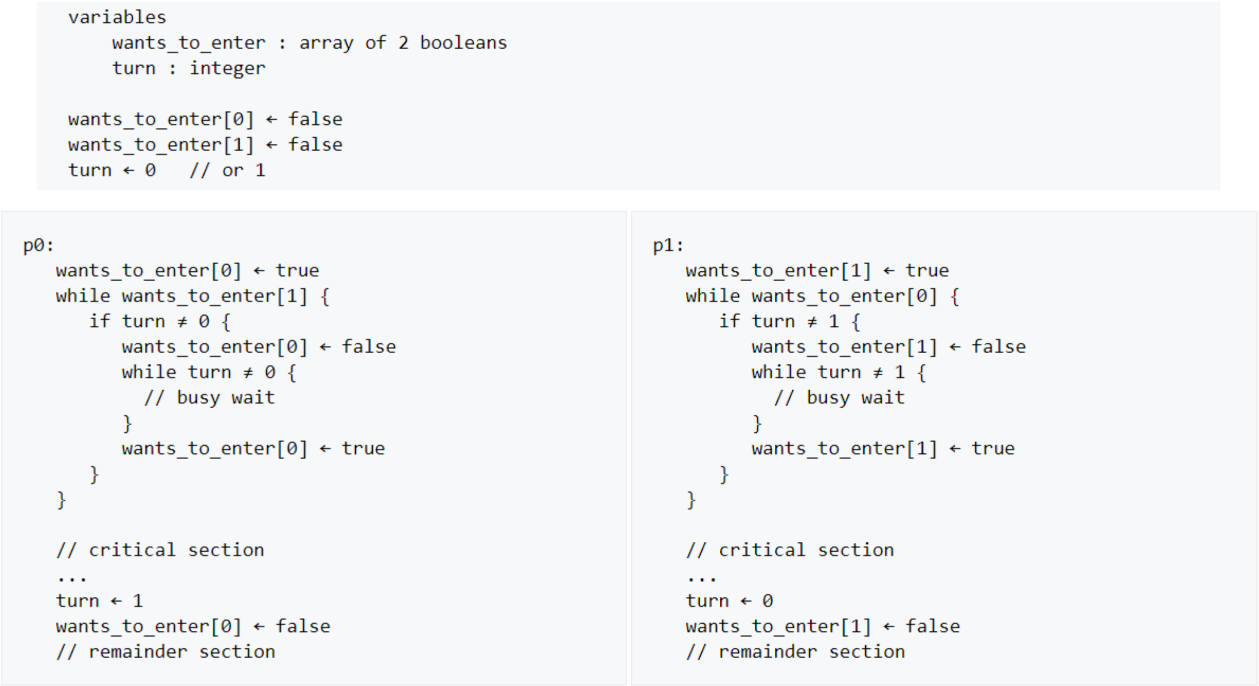
Priority inheritance: A runs at B's priority

MARS pathfinder failure:

http://wiki.csie.ncku.edu.tw/embedded/priority-inversion-on-Mars.pdf

Other ideas?

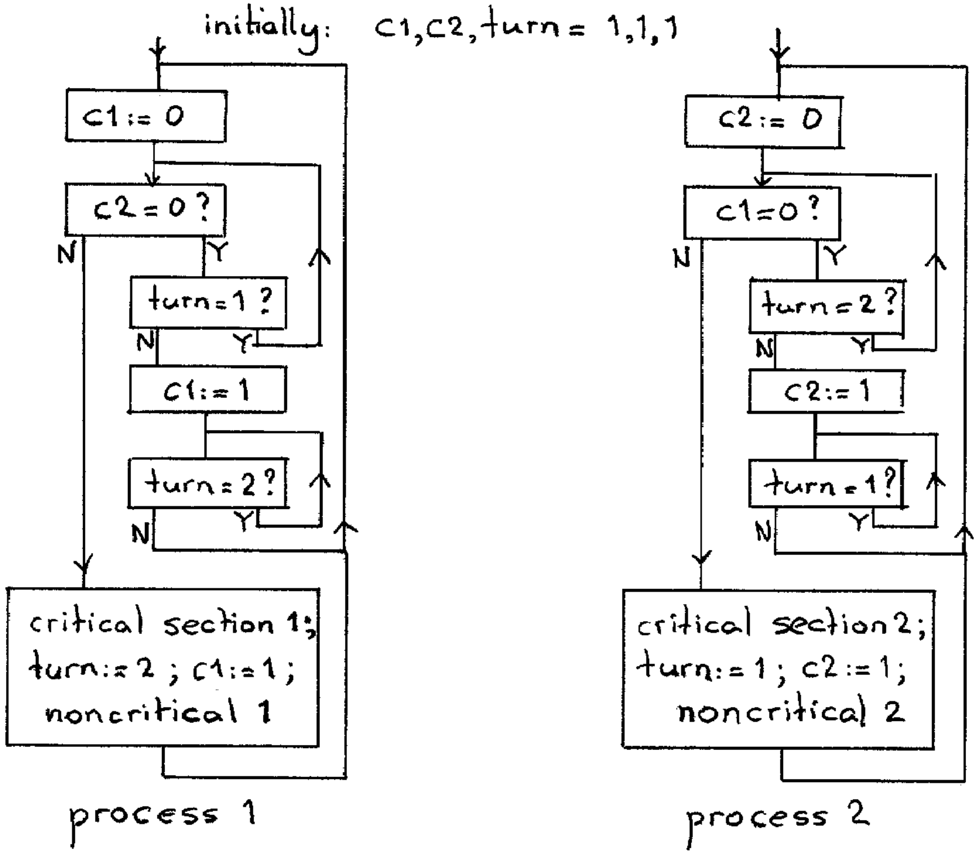
Dekker’s Algorithm



```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```



Th.J. Dekker's Solution

Lab #1

- Basic synchronization
- http://www.cs.utexas.edu/~rossbach/cs378/lab/lab0.html

- ***Start early!!!***

Questions?