Foundations: Synchronization Execution Abstractions

Chris Rossbach

CS378H Fall 2018

9/10/18

Today

- Questions?
- Administrivia
  - Lab 1 due sooner than you'd like
- Foundations
  - Threads/Processes/Fibers
  - Cache coherence (maybe)

- 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)
  - Andy Tannenbaum

Faux Quiz (answer any 2, 5 min)

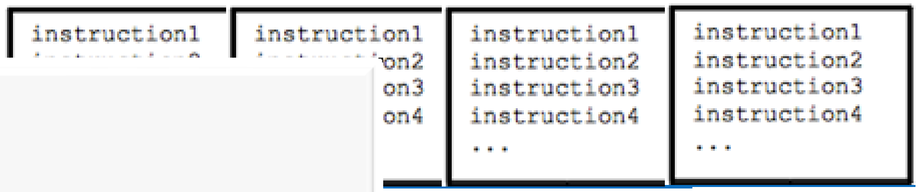
- What is the maximum possible speedup of a 75% parallelizable program on 8 CPUs
- What is super-linear speedup? List two ways in which super-linear speedup can occur.
- What is the difference between strong and weak scaling?
- Define Safety, Liveness, Bounded Waiting, Failure Atomicity
- What is the difference between processes and threads?
- What’s a fiber? When and why might fibers be a better abstraction than threads?

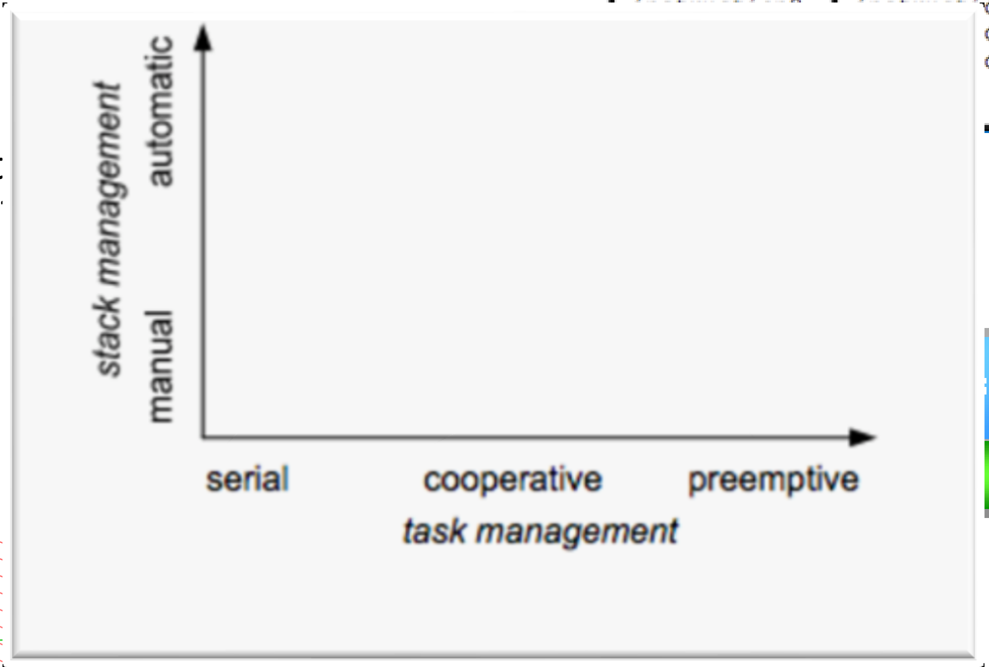
Faux Quiz (answer any 2, 5 min)

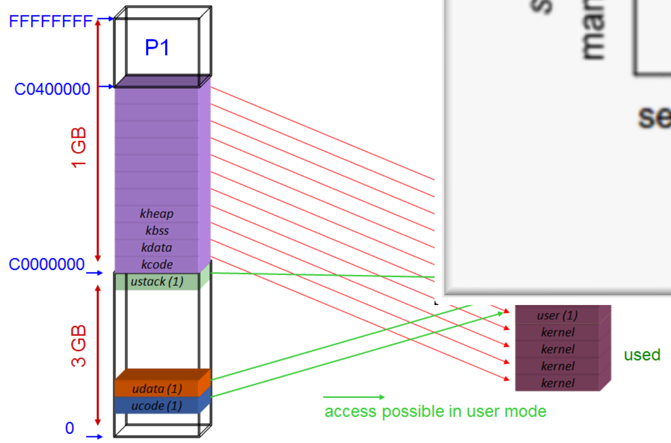
- What is the maximum possible speedup of a 75% parallelizable program on 8 CPUs
- What is super-linear speedup? List two ways in which super-linear speedup can occur.
- What is the difference between strong and weak scaling?
- Define Safety, Liveness, Bounded Waiting, Failure Atomicity
- **What is the difference between processes and threads?**
- **What's a fiber? When and why might fibers be a better abstraction than threads?**

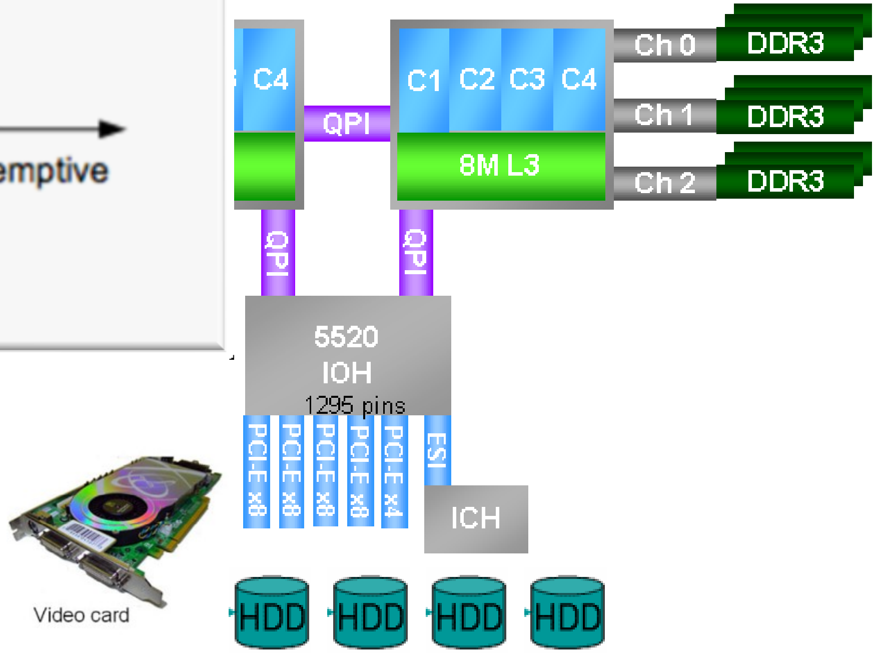
Processes and Threads and Fibers…

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

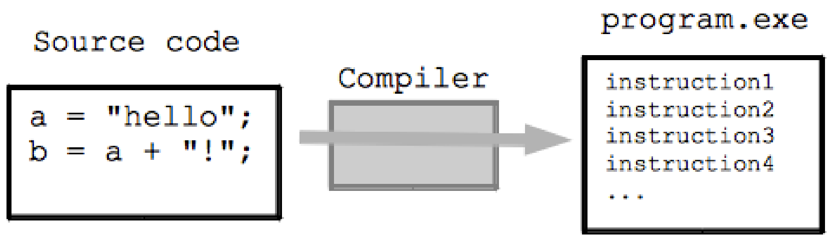








Programming and Machines: a mental model

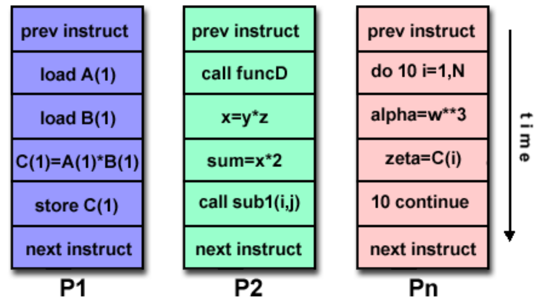


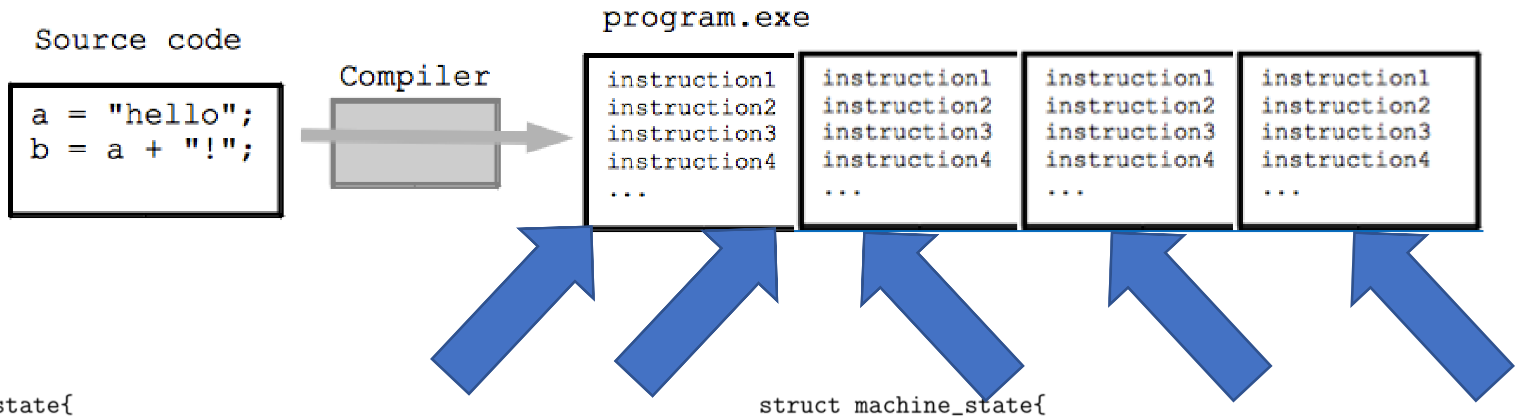
```
a = "hello";
b = a + "!";
```

```
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;
}
```

Parallel Machines: a mental model





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

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

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

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

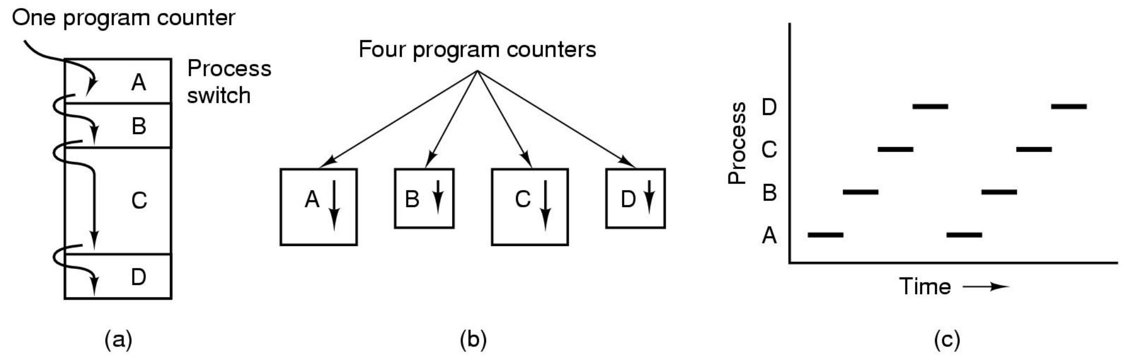
```
instruction1
instruction2
instruction3
instruction4
...
```

```
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;
}
```

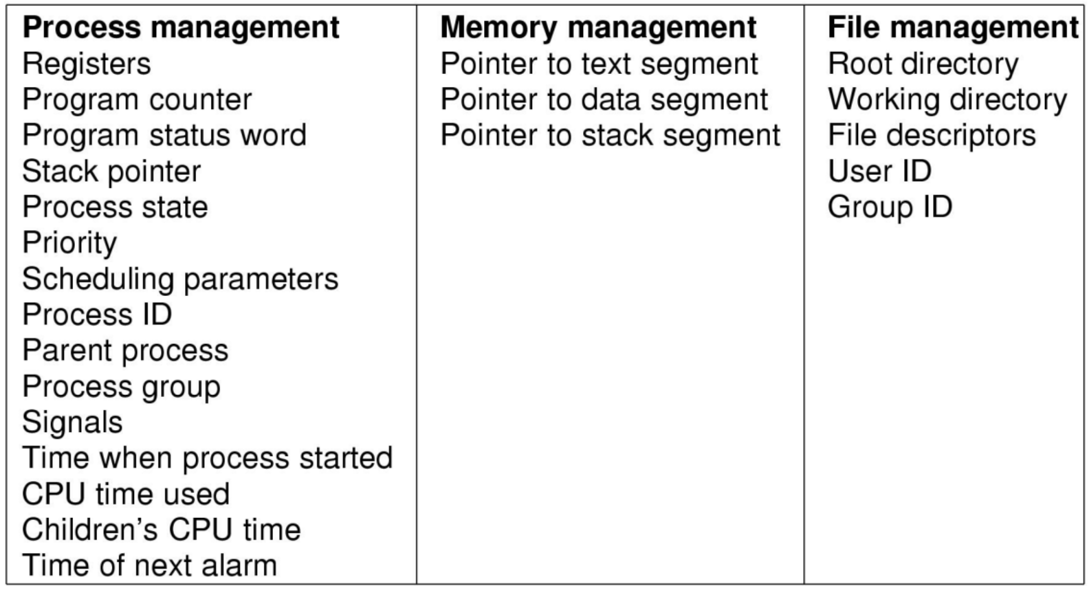
Processes

Model



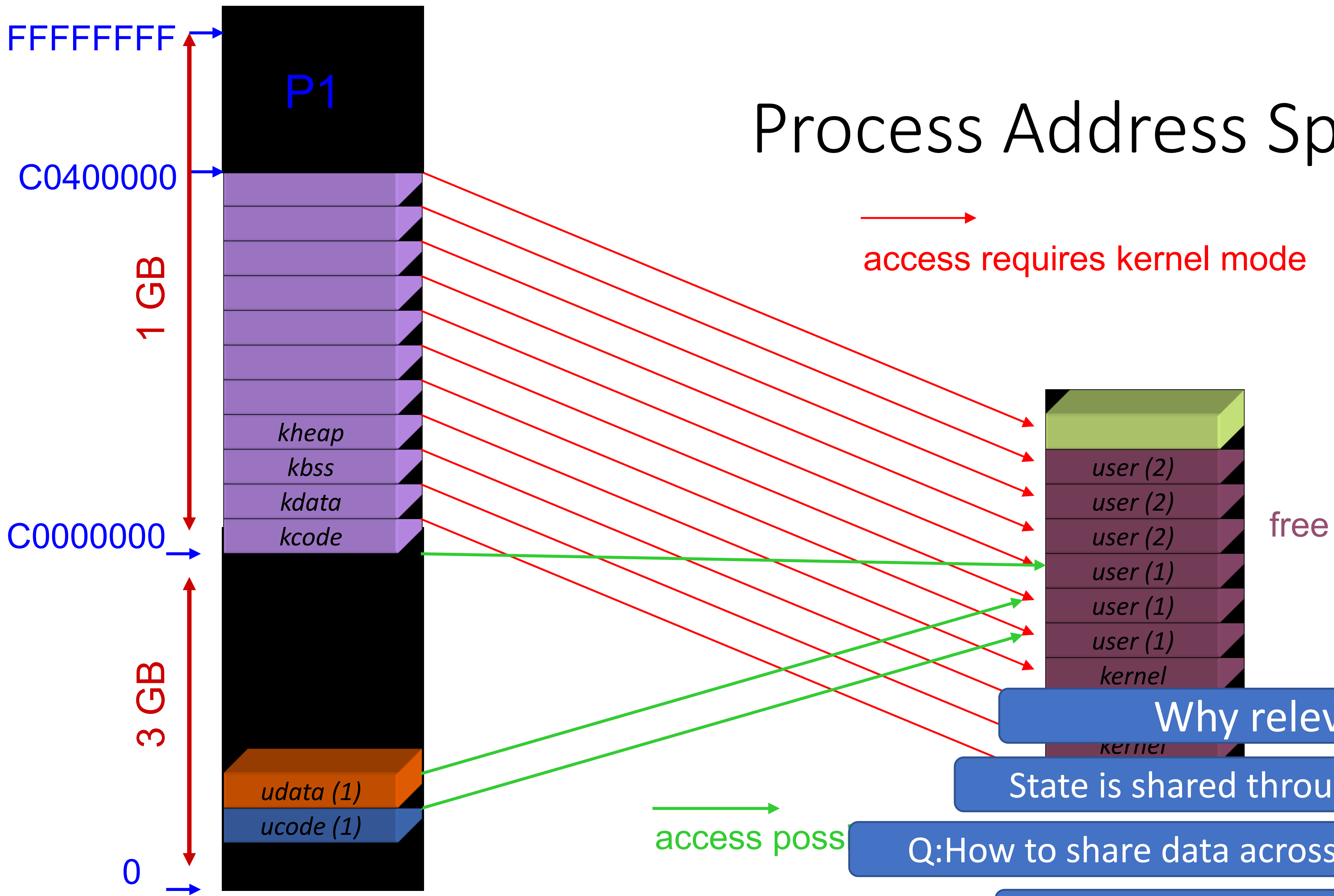
- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

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		

Process Address Space



access requires kernel mode

access poss

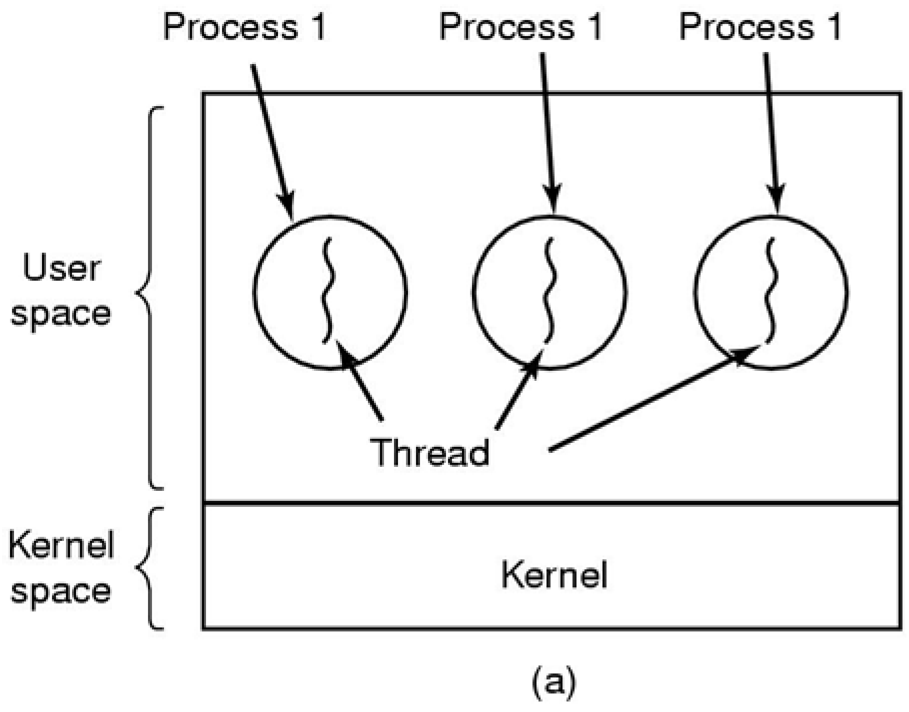
Why relevant?

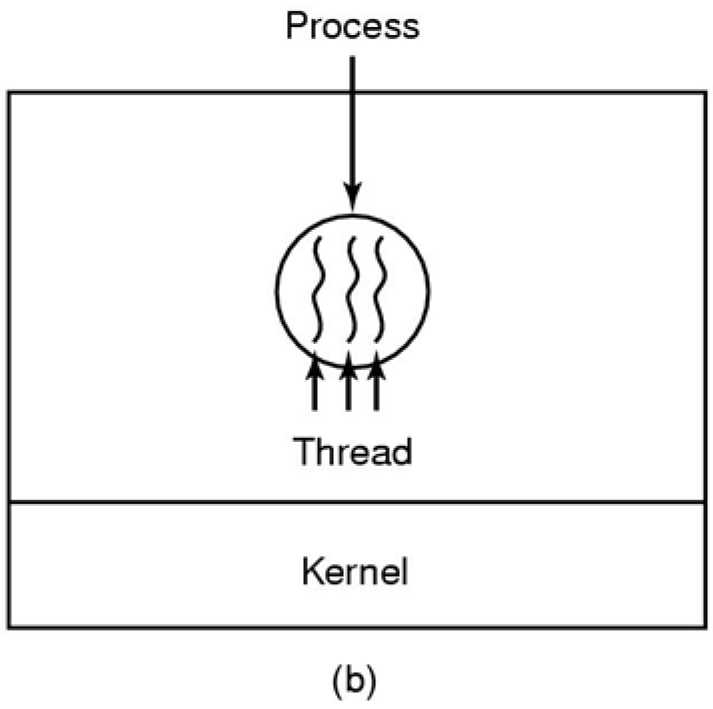
State is shared through memory!

Q:How to share data across processes?

Anyone see another issue?

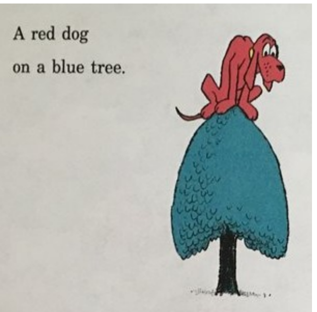
Abstractions for Concurrency





(a) Three processes each with one thread

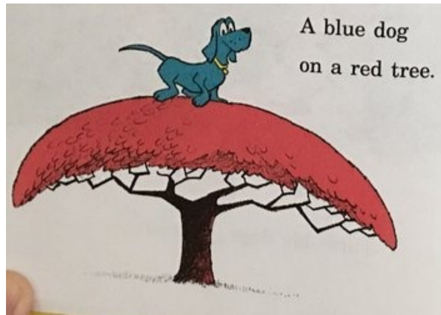
(b) One process with three threads



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

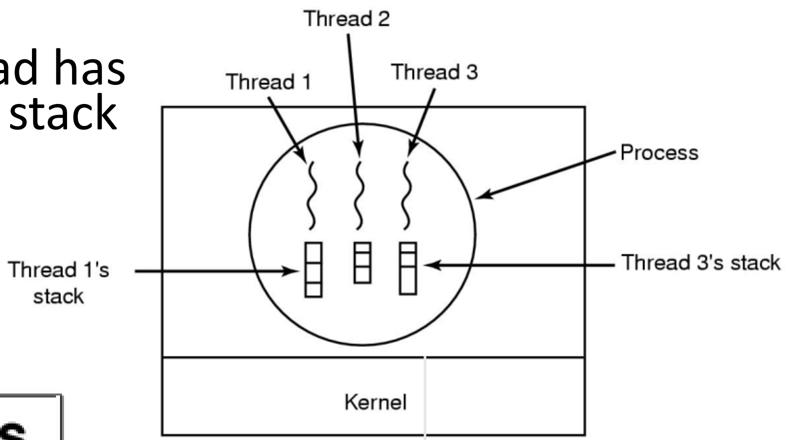
Could you do lab 1 with processes instead of threads?

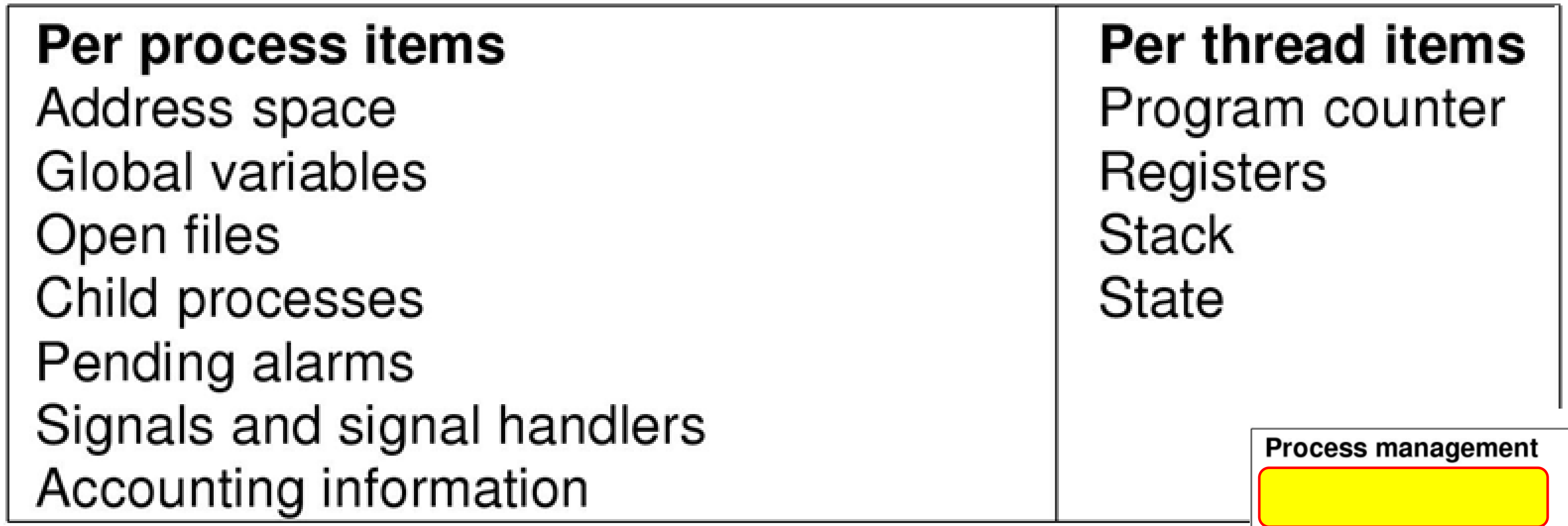
Threads simplify sharing and reduce context overheads



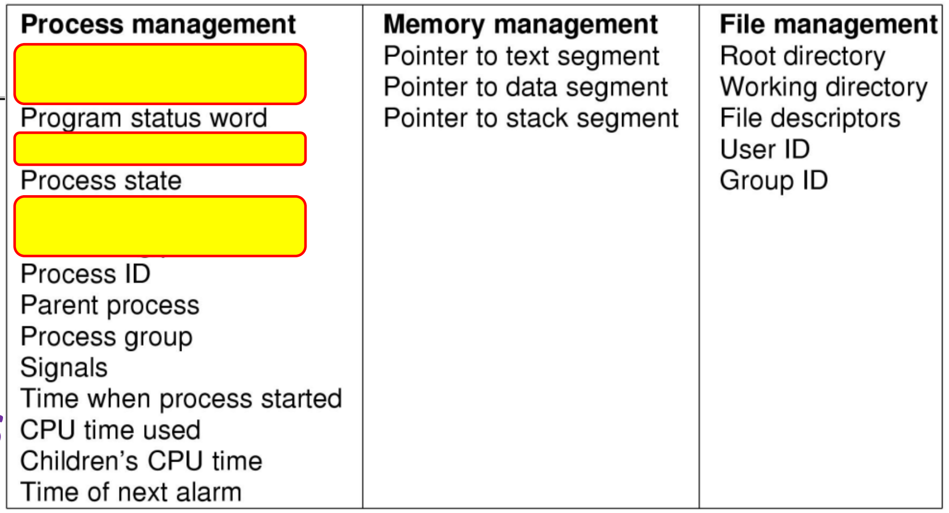
The Thread Model

Each thread has its own stack





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	

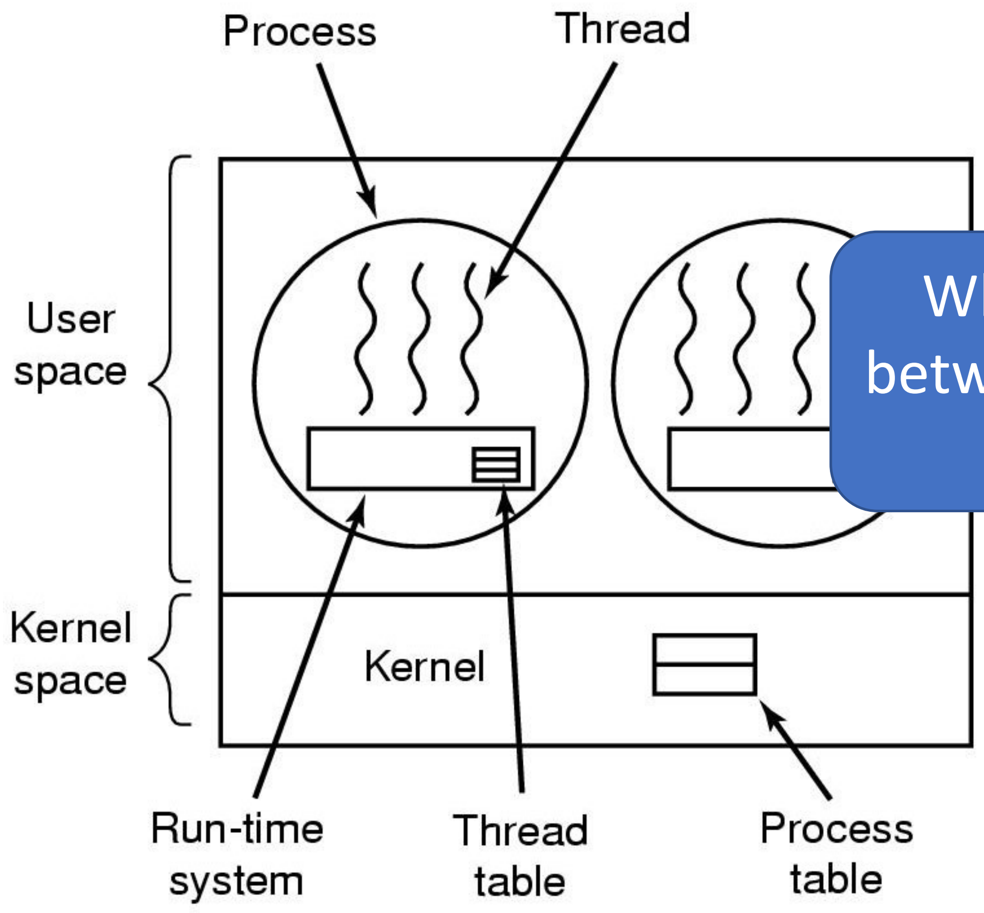


Process management	**Memory management**	**File management**
[redacted]	Pointer to text segment	Root directory
Program status word	Pointer to data segment	Working directory
[redacted]	Pointer to stack segment	File descriptors
Process state		User ID
[redacted]		Group ID
Process ID		
Parent process		
Process 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
- ***Decouples memory and control abstractions***
- ***What is the advantage of that?***

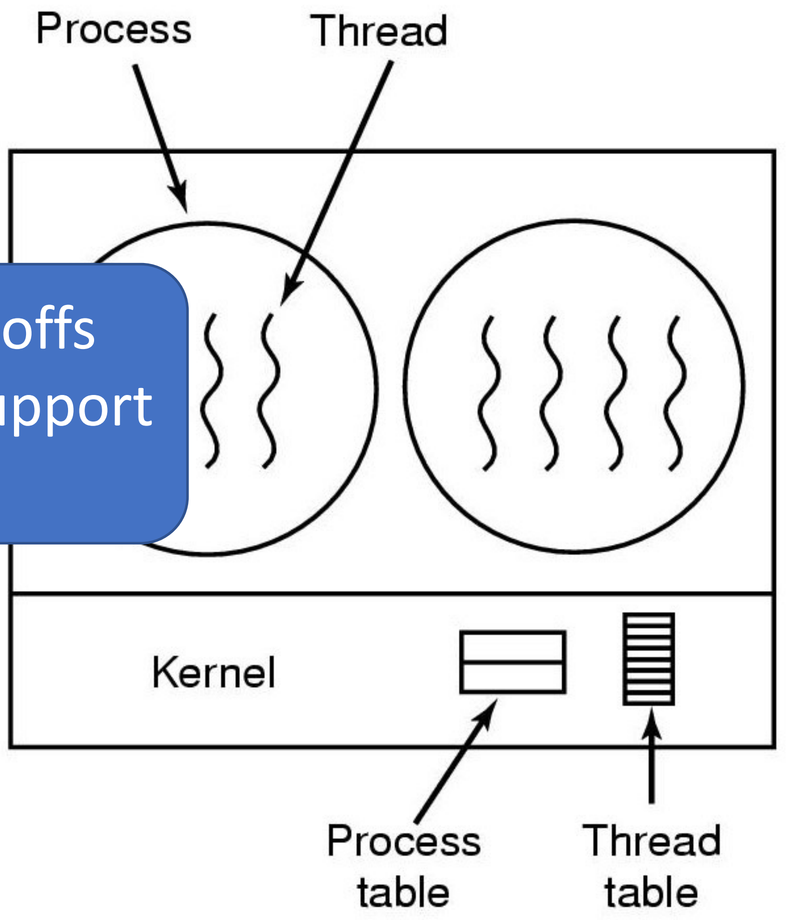
Where to Implement Threads:

User Space



A user-level threads package

Kernel Space

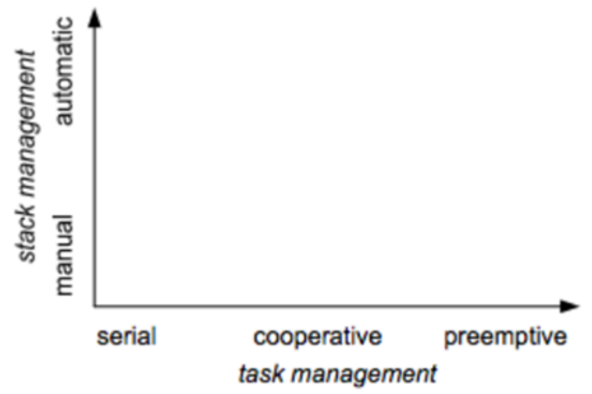


A threads package managed by the kernel

What are some tradeoffs between user/kernel support for threads?

Execution Context Management

"Task" == "Flow of Control", but with less typing
"Stack" == Task State



Task Management

- Preemptive
  - Interleave on uniprocessor
  - Overlap on multiprocessor
- Serial
  - One at a time, no conflict
- Cooperative
  - Yields at well-defined points
  - E.g. wait for long-running I/O

Stack Management

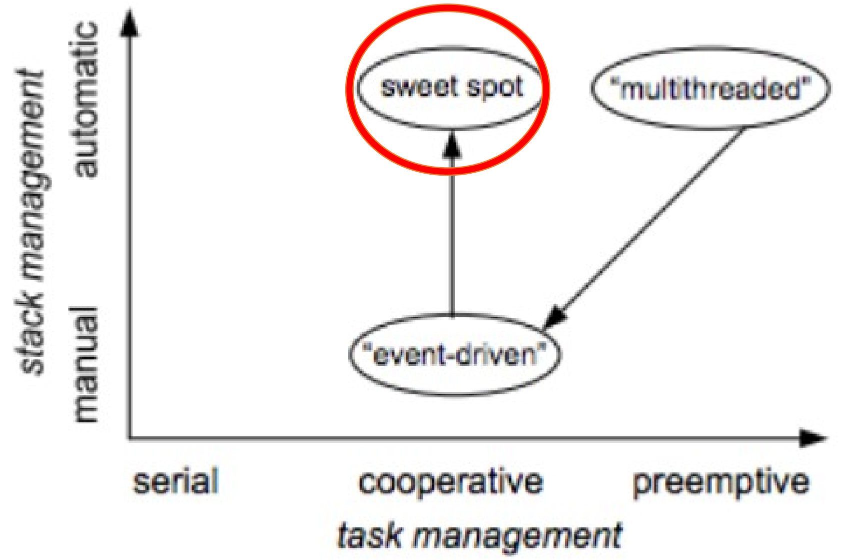
- Manual
  - Inherent in Cooperative
  - Changing at quiescent points
- Automatic
  - Inherent in pre-emptive
  - Downside: Hidden concurrency assumptions

These dimensions can be orthogonal

Fibers: the Sweet Spot?

- Cooperative tasks
  - most desirable when reasoning about concurrency
  - usually associated with event-driven programming
- Automatic stack management
  - most desirable when reading/maintaining code
  - Usually associated with threaded (or serial) programming

Fibers: cooperative threading with automatic stack management

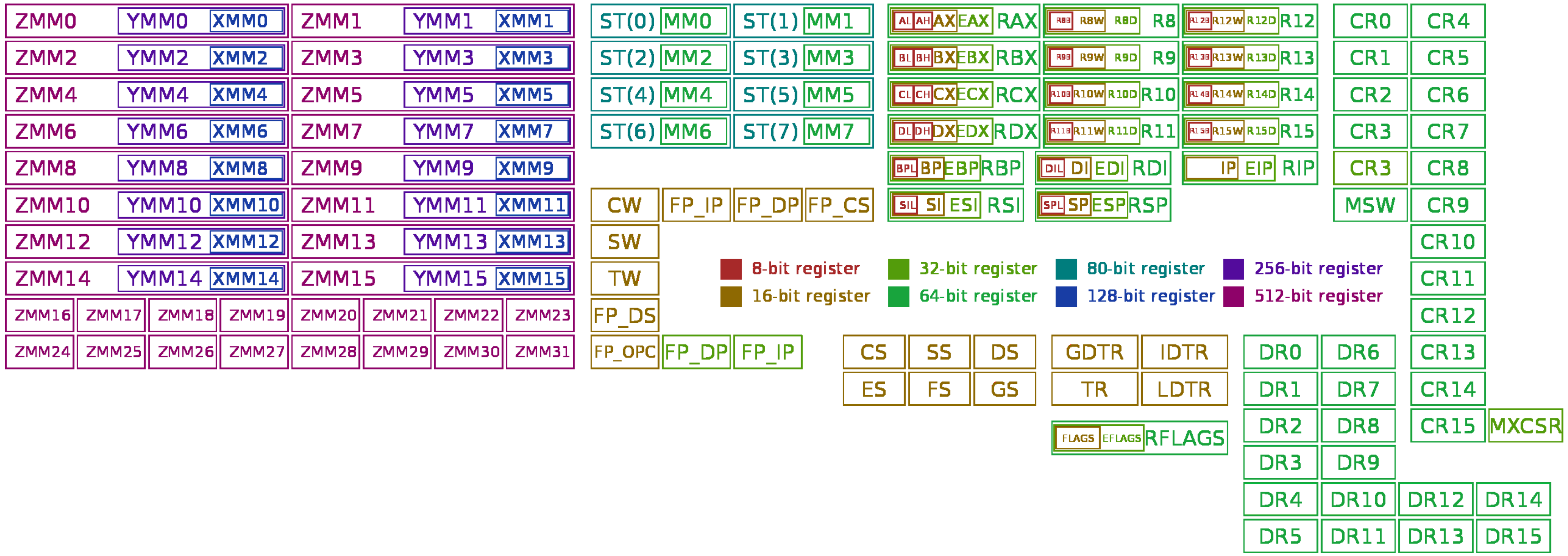


Threads vs Fibers

Blah blah **fibers** blah **thread** blah…

- 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

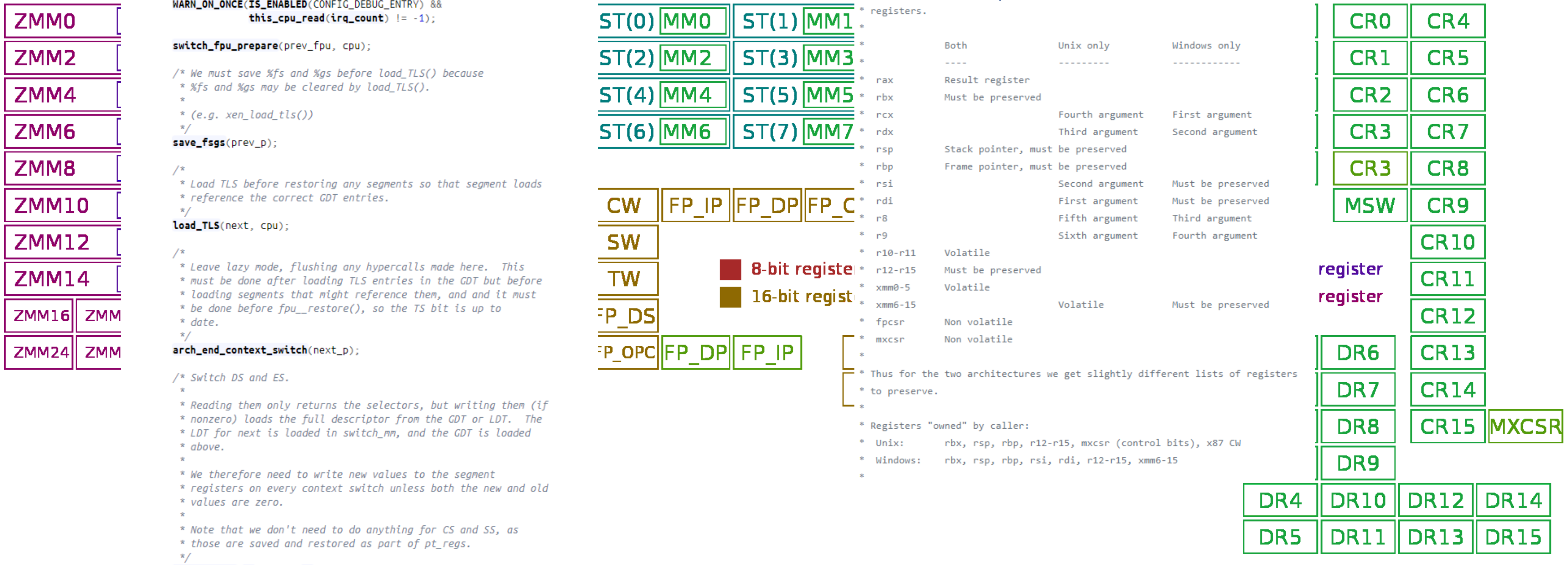
x86_64 Architectural Registers

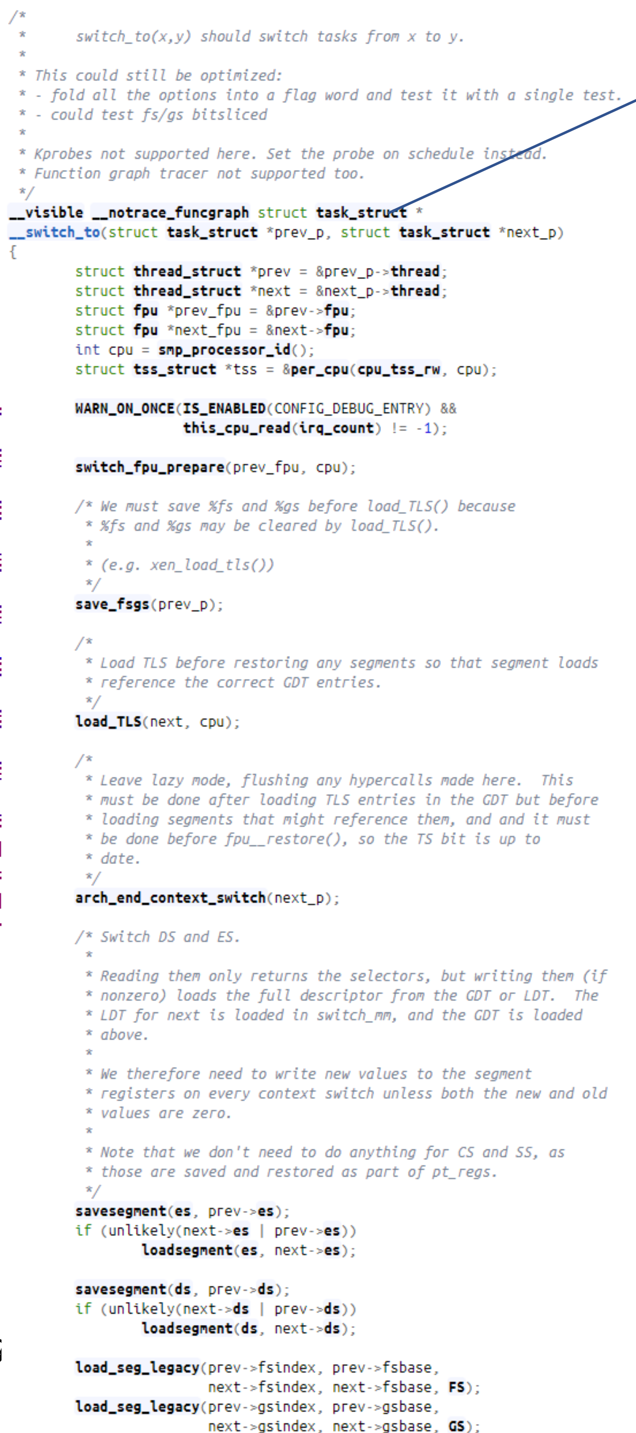


- *Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525*

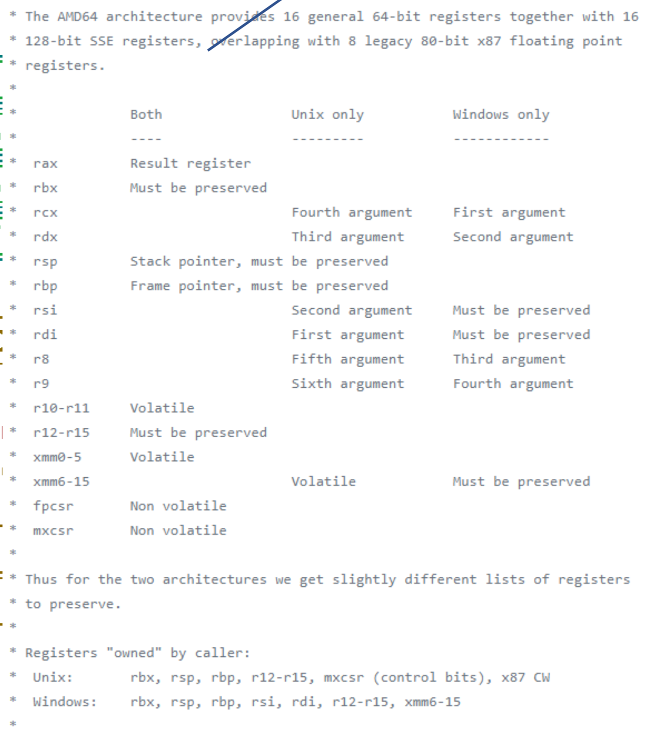
Linux x86_64 context switch *excerpt*

Complete fiber context switch on Unix and Windows





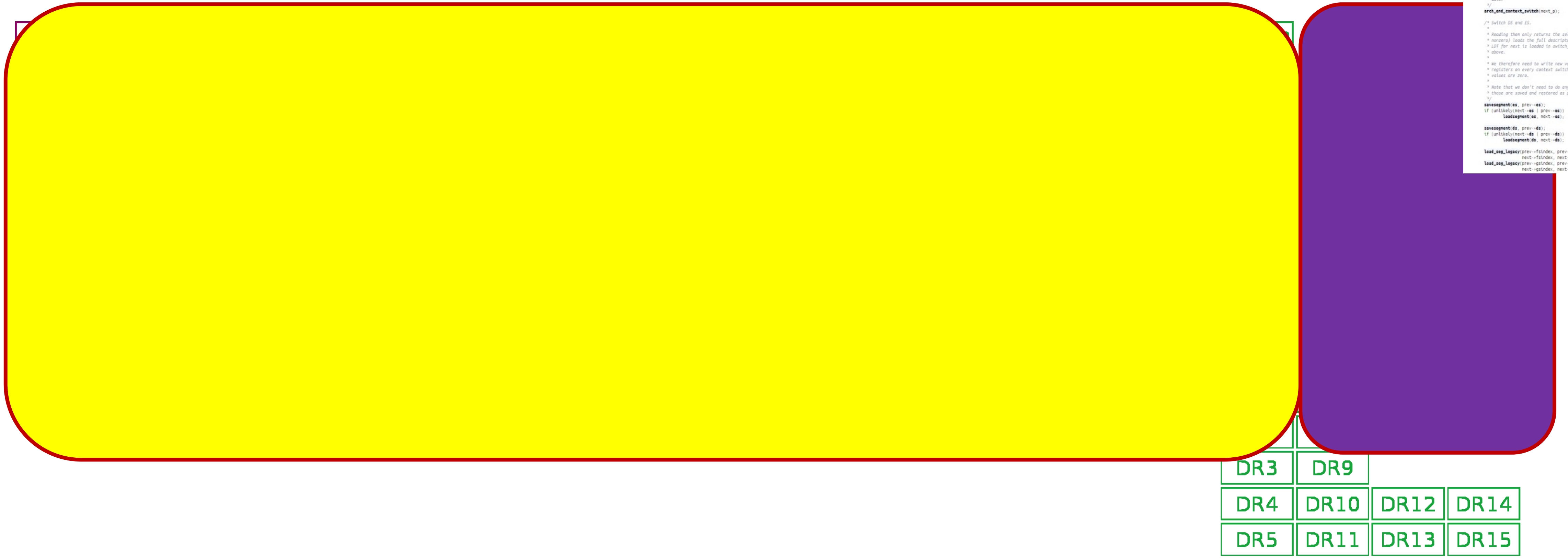
```
/*
 *      switch_to(x,y) should switch tasks from x to y.
 *
 * This could still be optimized:
 * - fold all the options into a flag word and test it with a single test.
 * - could test fs/gs bitsliced
 *
 * Kprobes not supported here. Set the probe on schedule instead.
 * Function graph tracer 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);
```

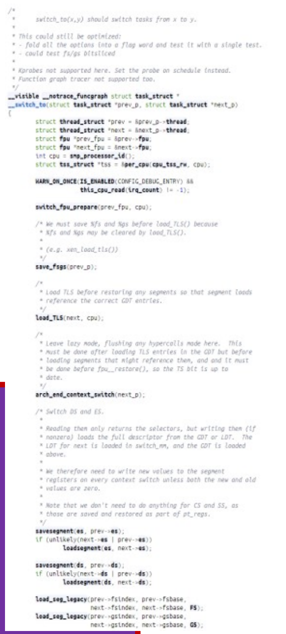


```
* The AMD64 architecture provides 16 general 64-bit registers together with 16
* 128-bit SSE registers, overlapping with 8 legacy 80-bit x87 floating point
* registers.
*
*             Both                 Unix only            Windows only
*             ----                 ---------            ------------
*  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
*
```

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

x86_64 Registers and Threads



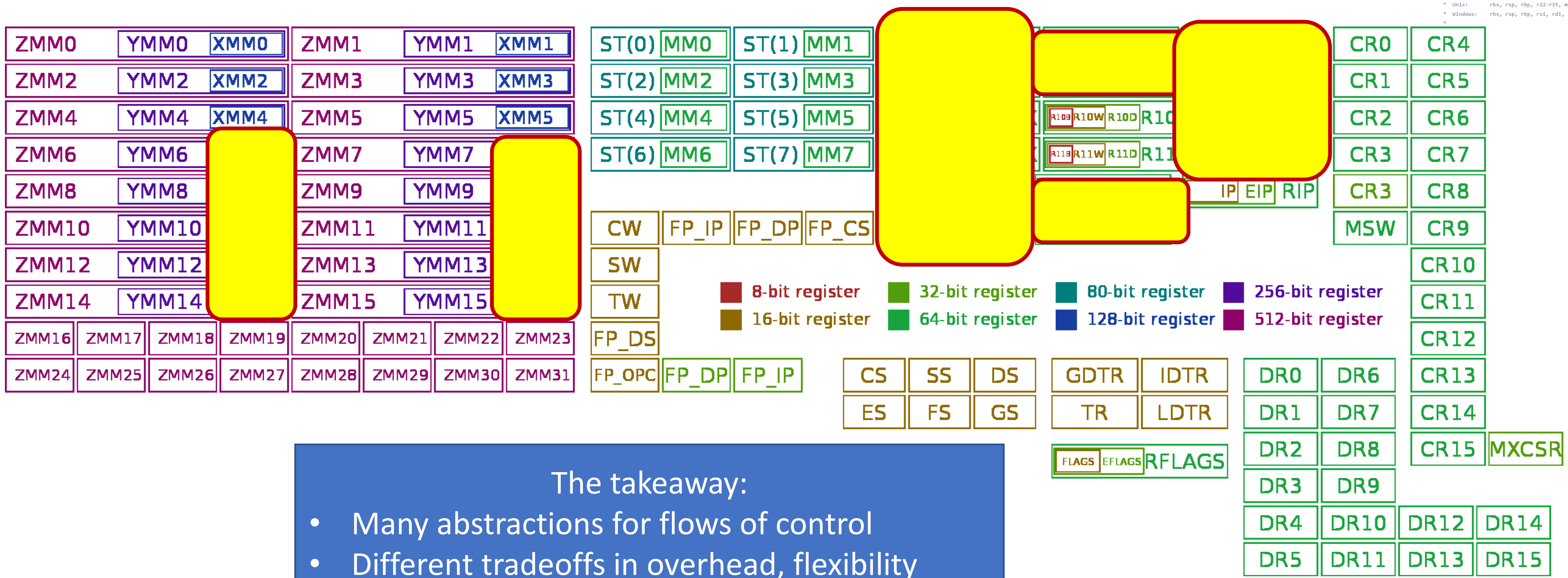


```
/*
 *	switch_to(x,y) should switch tasks from x to y.
 *
 * This could still be optimized:
 * - fold all the options into a flag word and test it with a single test.
 * - could test fs/gs bitsliced
 *
 * Kprobes not supported here. Set the probe on schedule instead.
 * Function graph tracer not supported 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);
```

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

```
* 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
*
```

x86_64 Registers and Fibers



The takeaway:

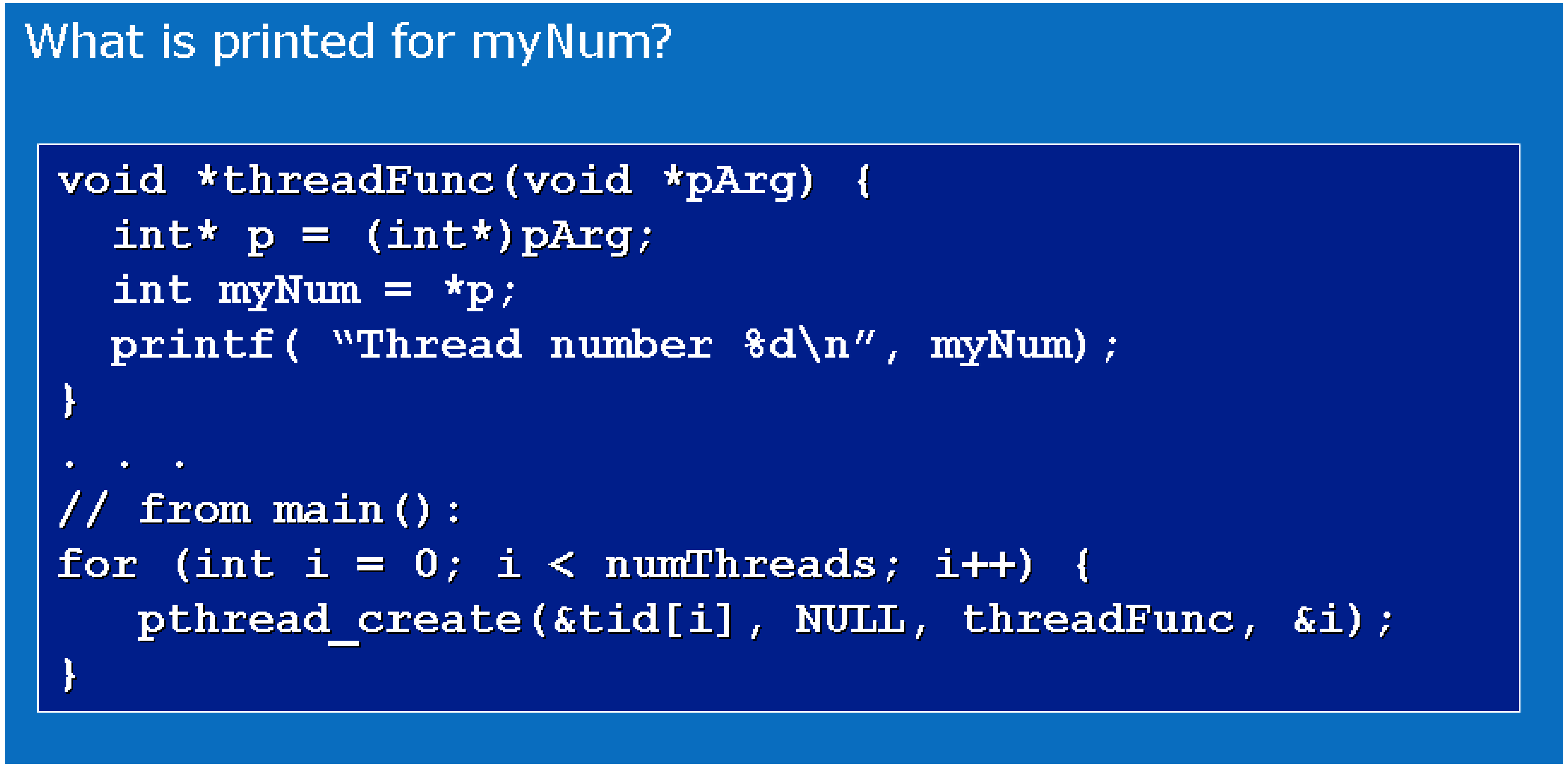
- Many abstractions for flows of control
- Different tradeoffs in overhead, flexibility
- Matters for concurrency: exercised heavily

- *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

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);
```

```
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: 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) {

    Critical 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
}
```

(test & set ~= CAS ~= LLSC)

TST: ***Test&set***

- Reads a value from memory
- Write "1" back to memory location

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

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…

Implementing Locks

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

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

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

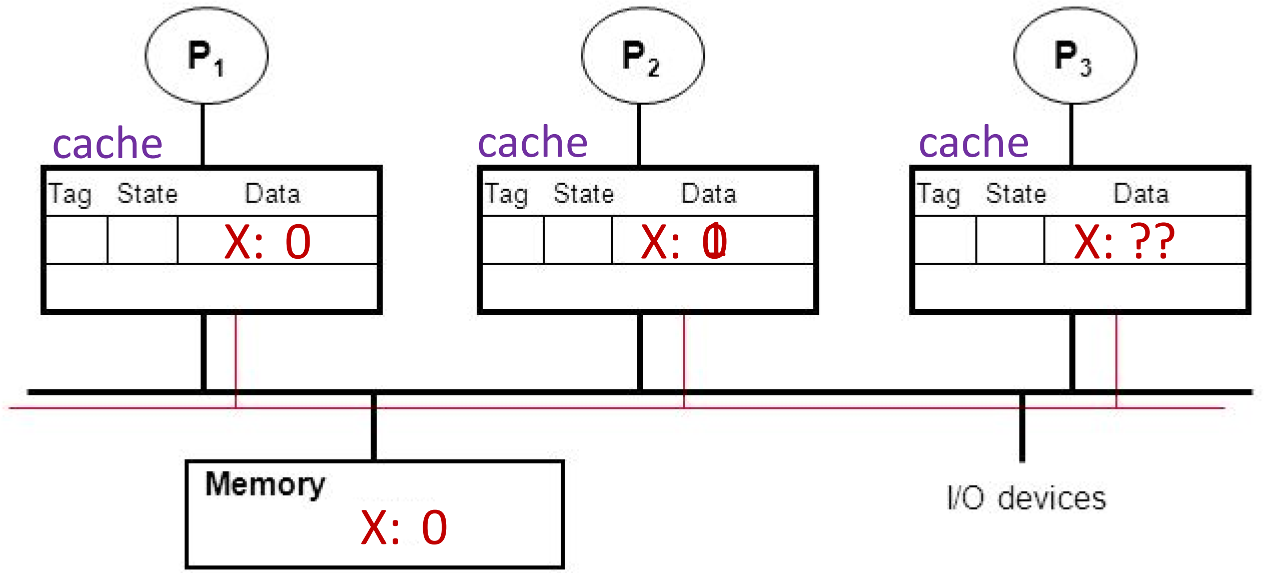
What are the problem(s) with this?

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

Multiprocessor Cache Coherence

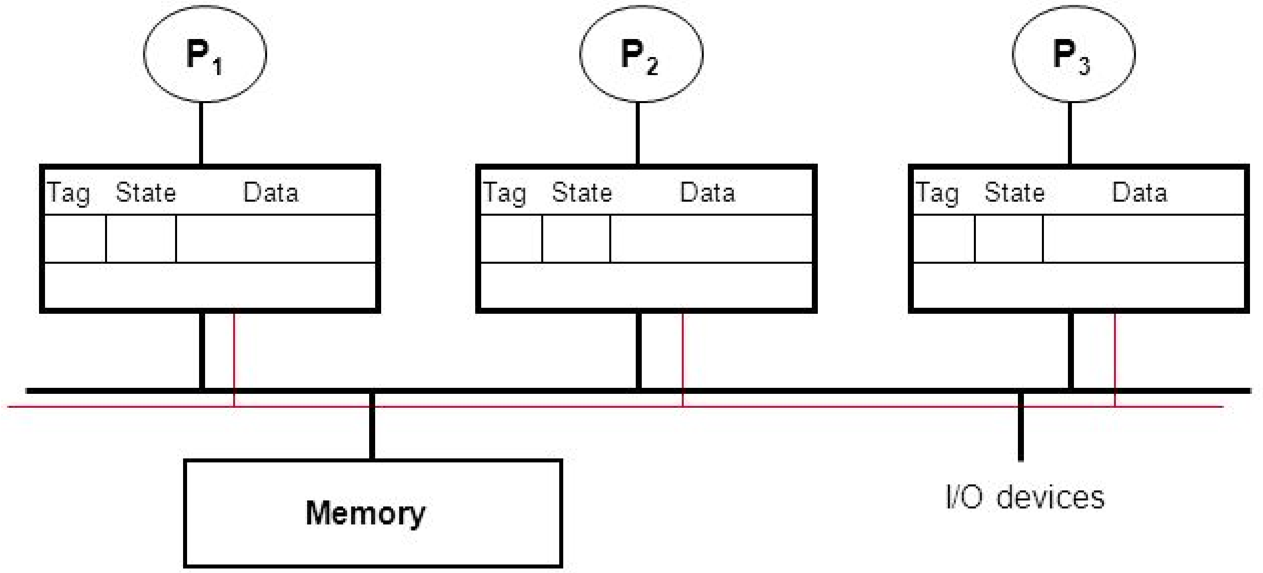
Physics	Concurrency
$F = ma$	~ *coherence*

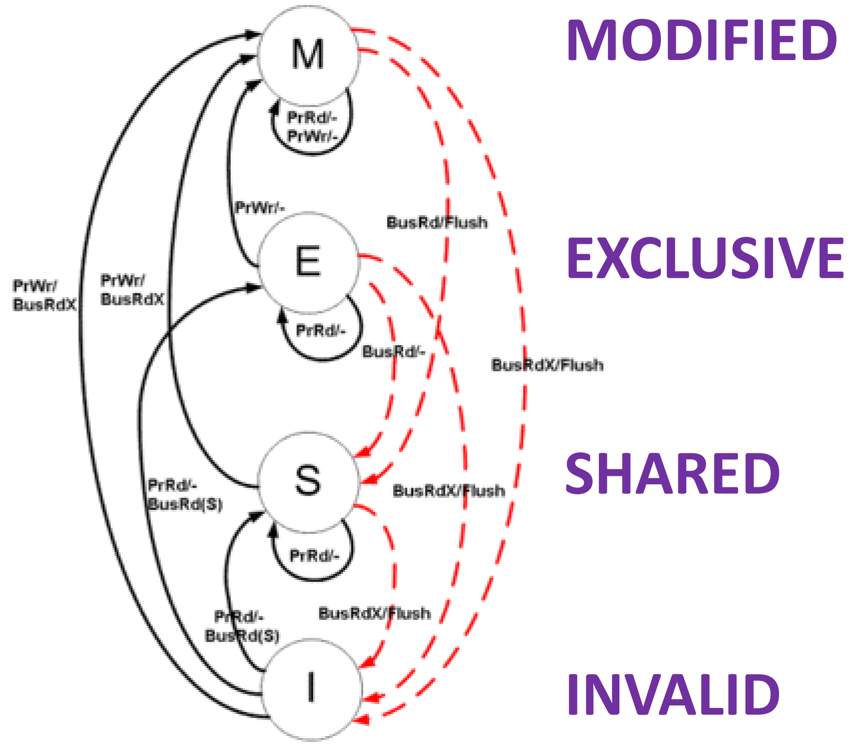
Multiprocessor Cache Coherence



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

Multiprocessor Cache Coherence

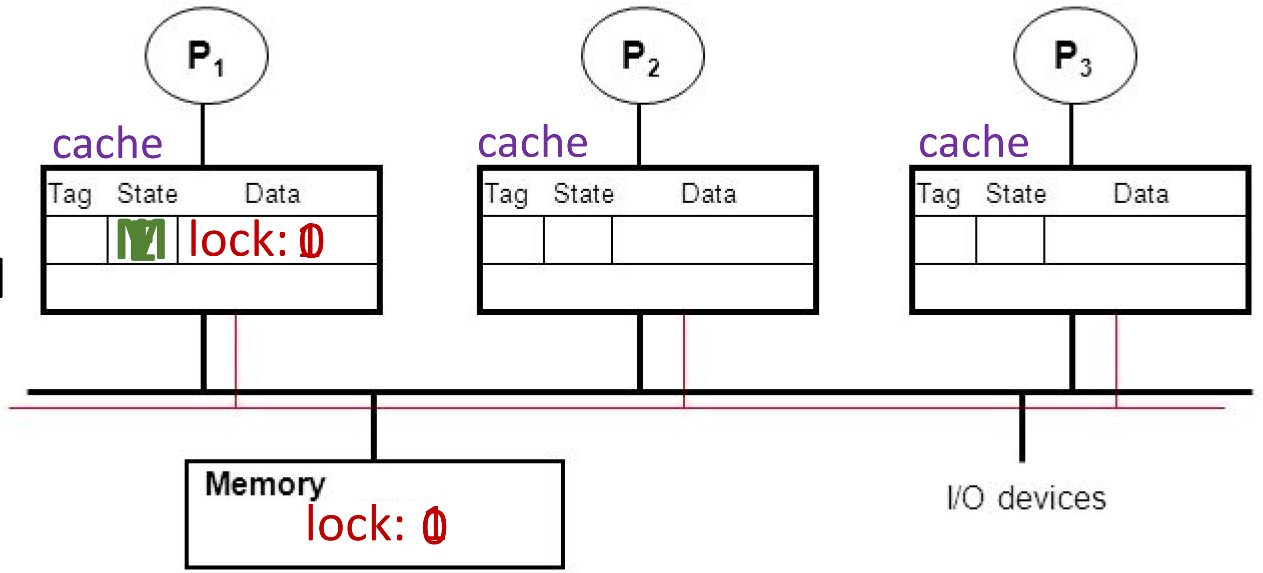




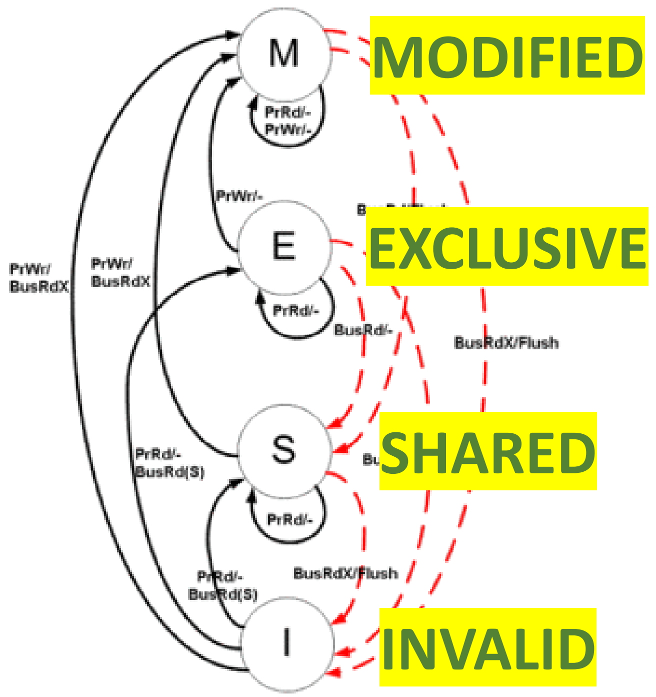
Each cache line has a state (M, E, S, I)

- 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



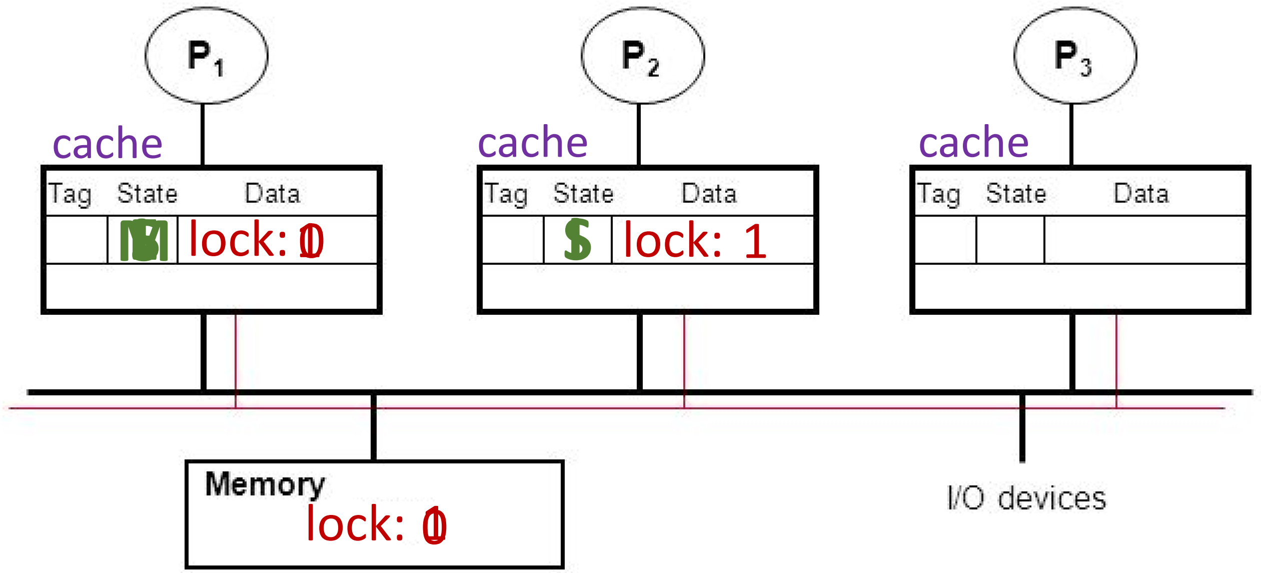
[cache eviction]

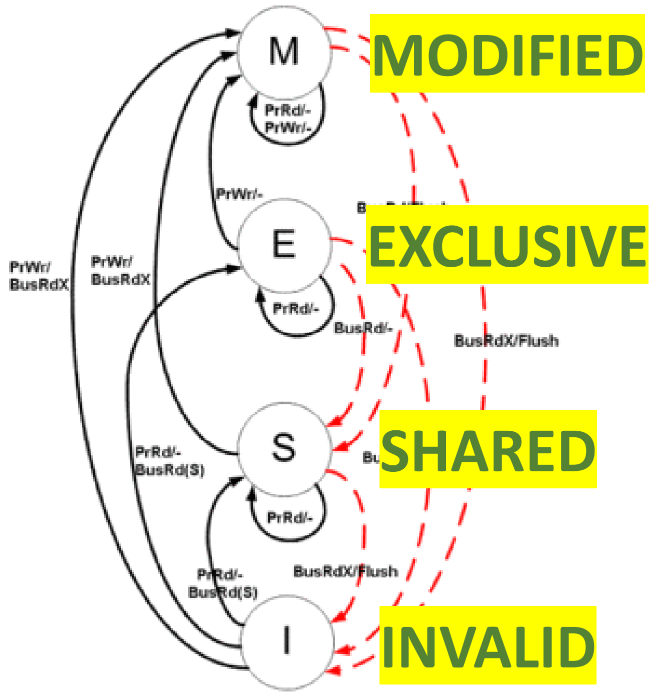


P1

```
// (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

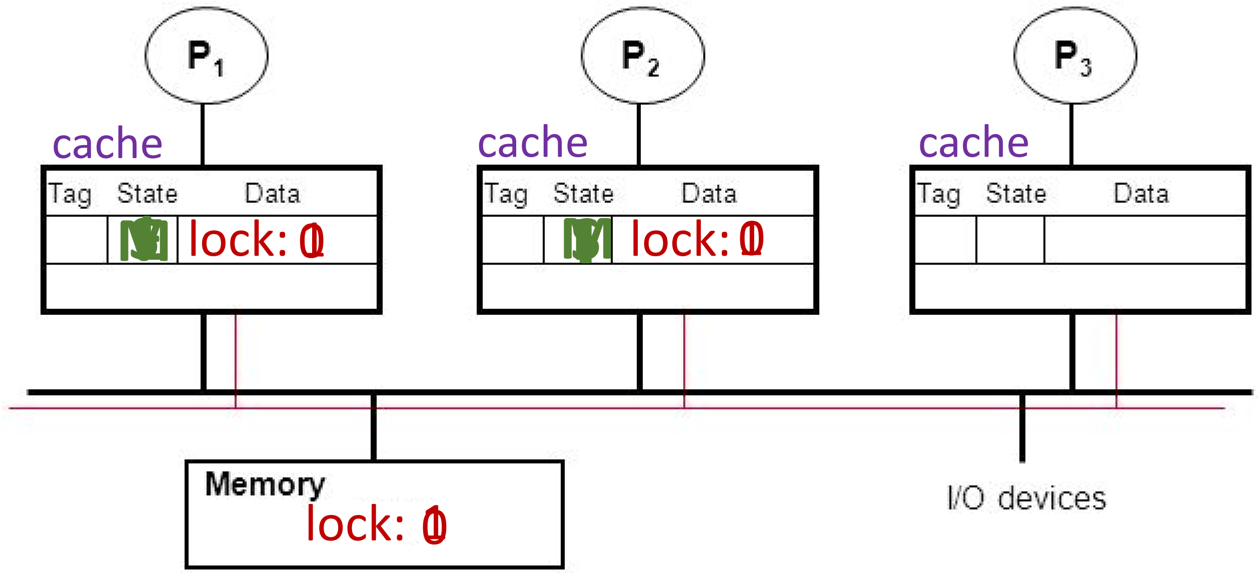
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

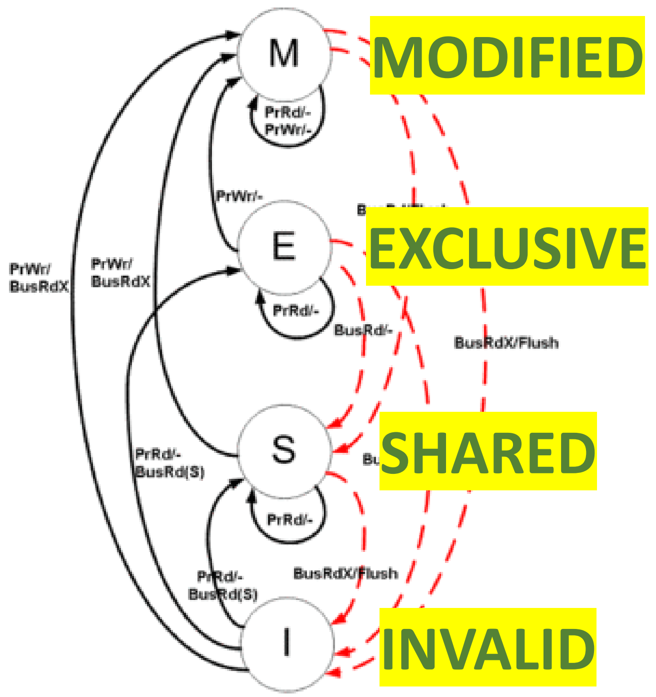
SAFE!

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





P1

P2

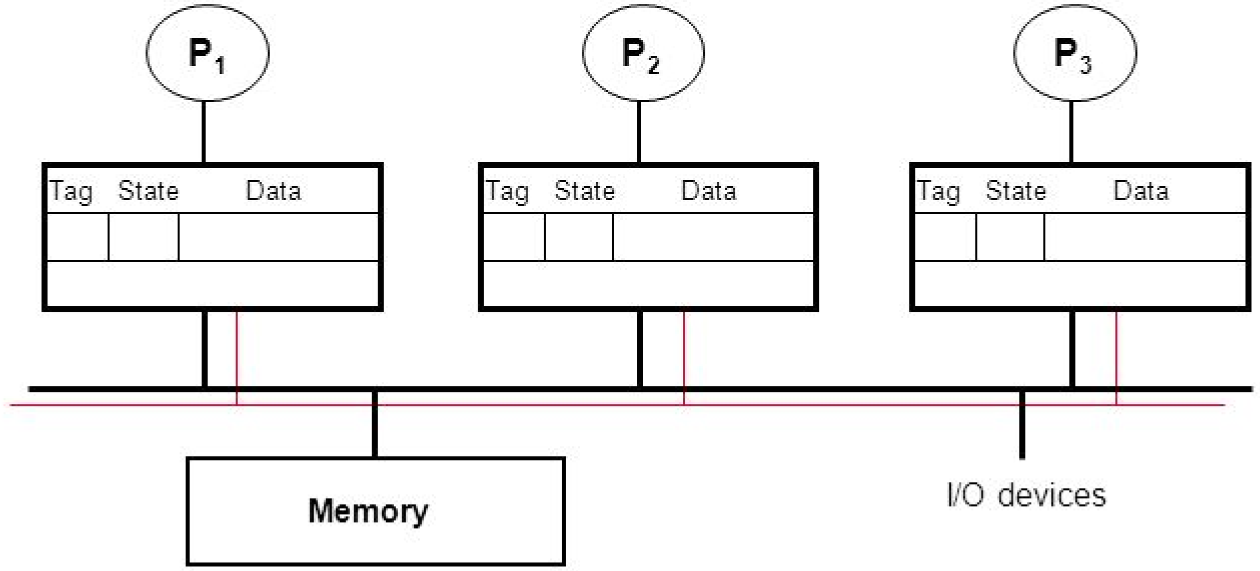
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

```
// (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:



}
```

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

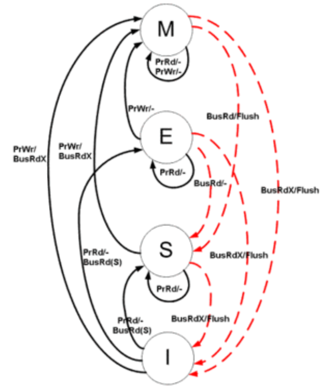
LLSC: load-linked store-conditional

PPC, Alpha, MIPS

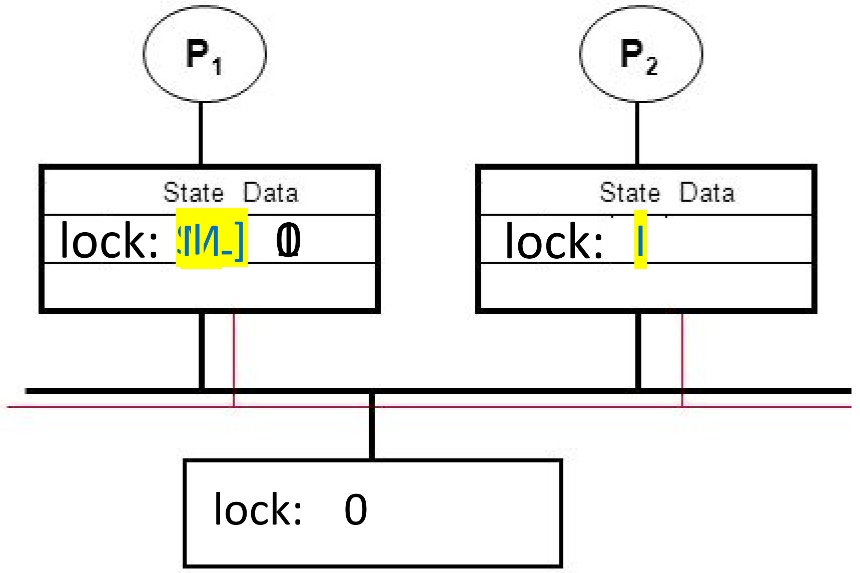
```
bool LLSC(addr, val) {
 ret = *addr;
 atomic {
   if(*addr == ret) {
     *addr = val;
     return true;
  }
  return false;
}
```

```
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



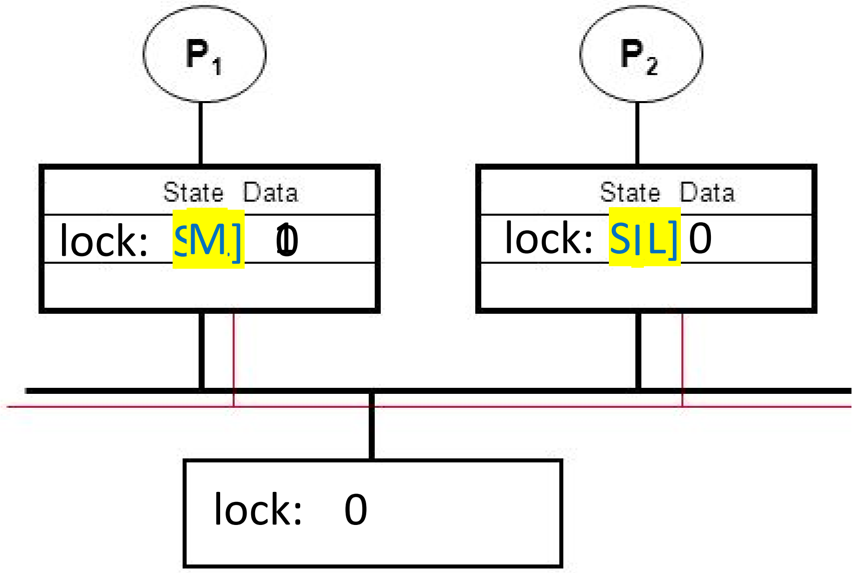
LLSC Lock Action Zone

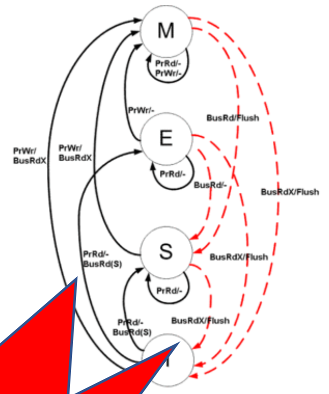


P1
```
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

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)

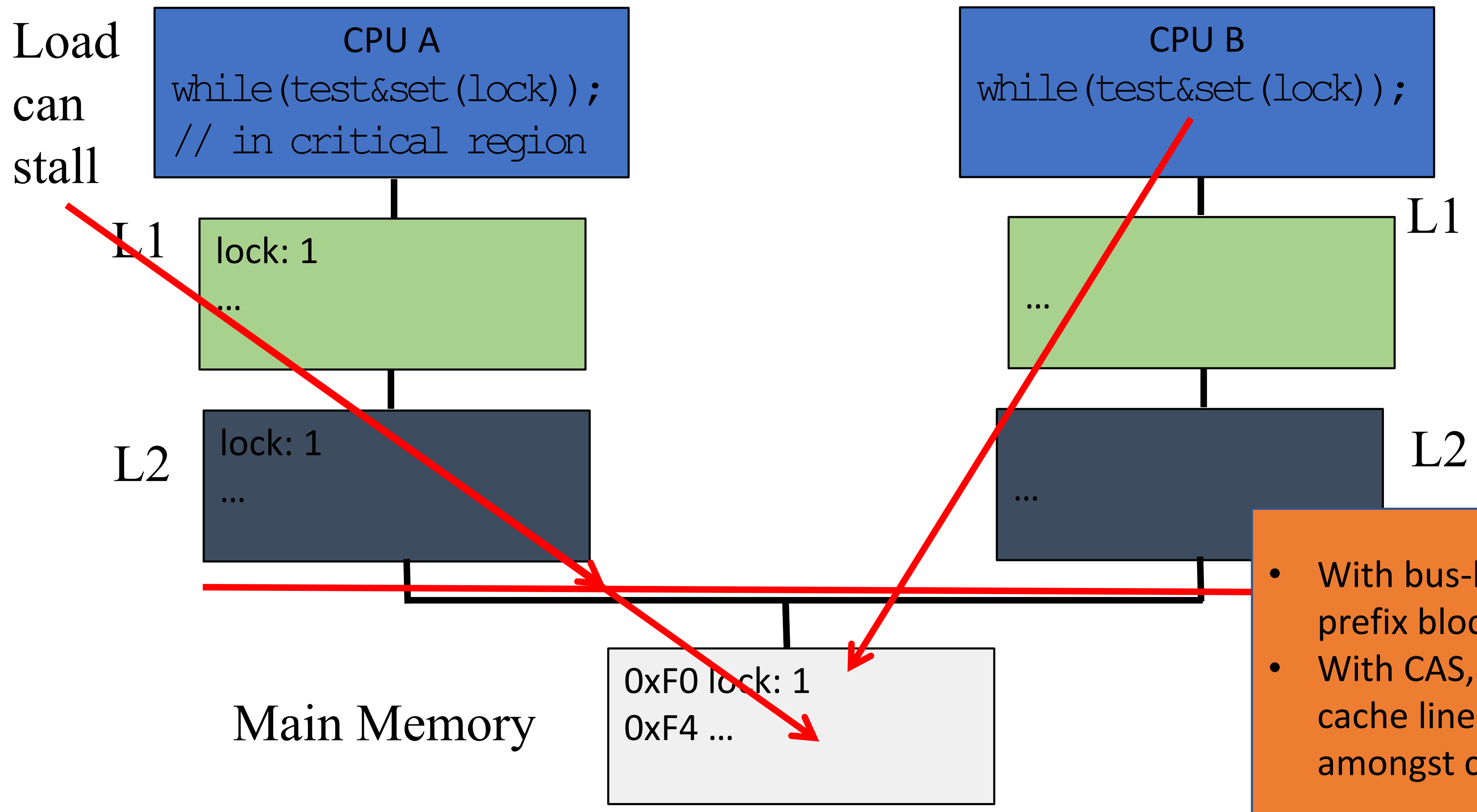
```
Lock::Release() {
    *lock = 0;
}
```

- 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?



- 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);
}
```

Test&Test&Set

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

Busy-wait on in-memory copy

Busy-wait on cached copy

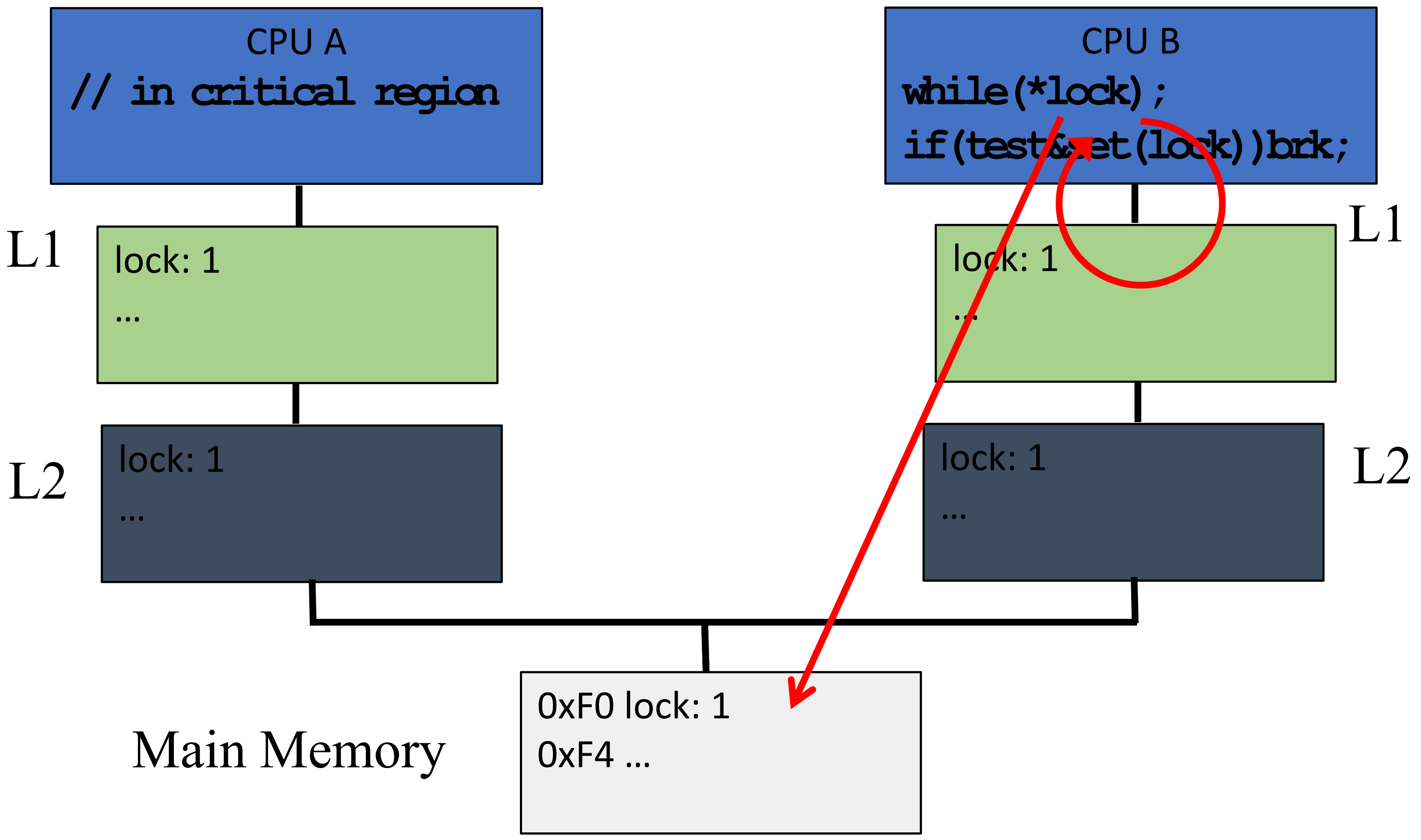
```
Lock::Release() {
    *lock = 0;
}
```

```
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

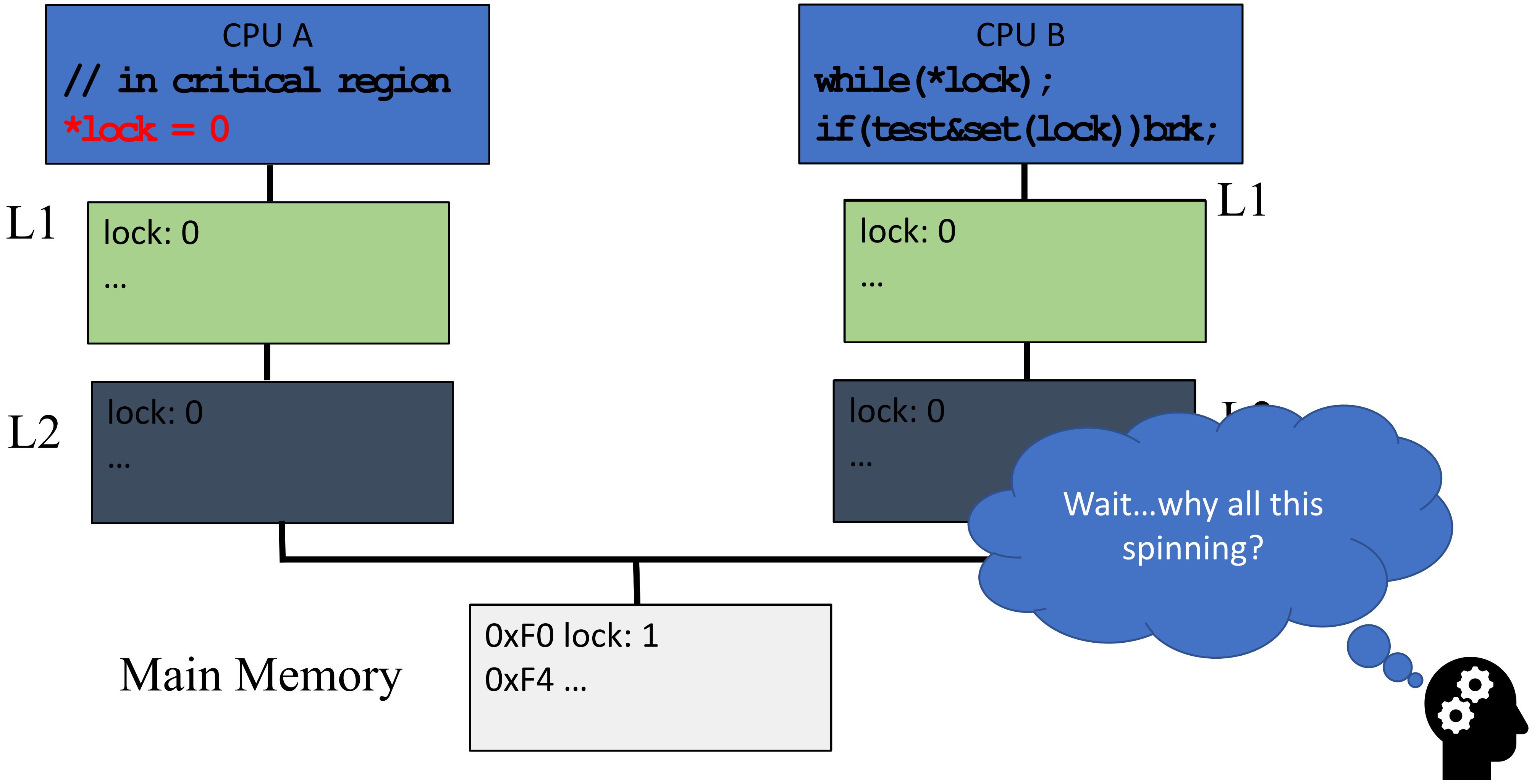
Test & Test & Set with Memory Hierarchies

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?



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

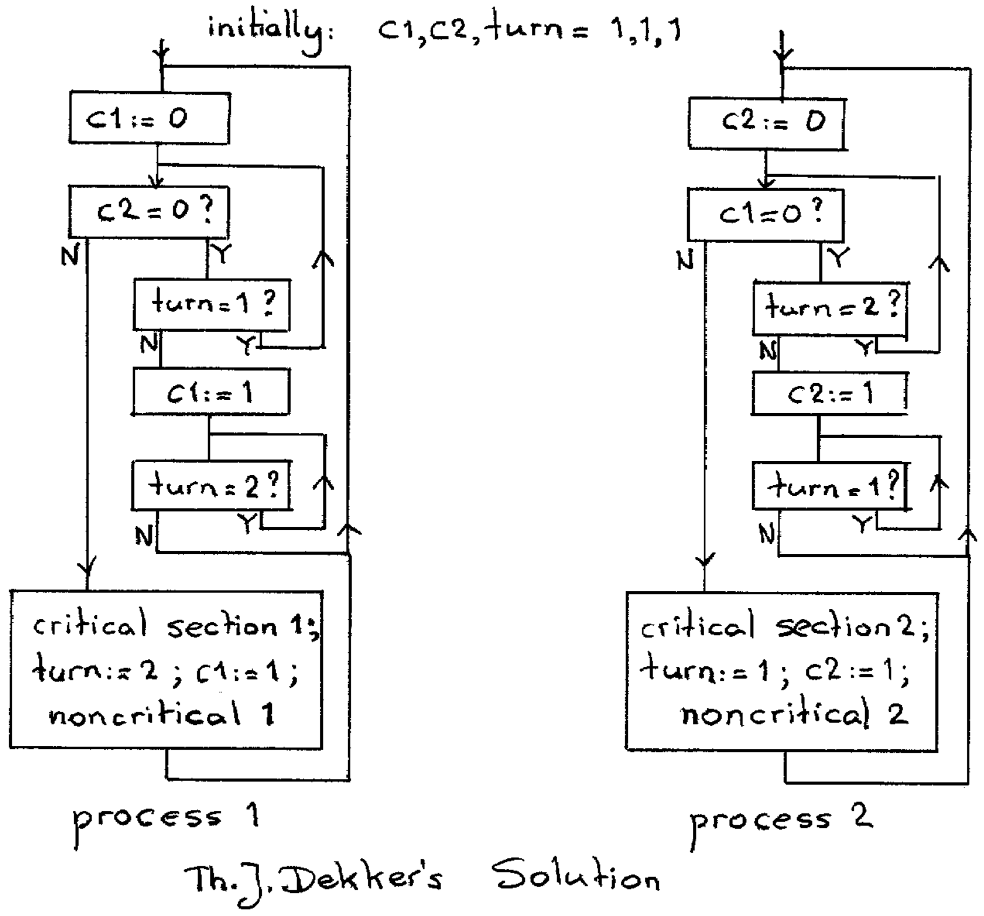
Other ideas?

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
```


initially: c1,c2,turn = 1,1,1
c1 := 0
c2 = 0 ?
N
Y
turn = 1 ?
N
Y
c1 := 1
turn = 2 ?
N
Y
critical section 1; turn := 2 ; c1 := 1; noncritical 1
c2 := 0
c1 = 0 ?
N
Y
turn = 2 ?
N
Y
c2 := 1
turn = 1 ?
N
Y
critical section 2; turn := 1; c2 := 1; noncritical 2
process 1
process 2
Th.J.Dekker's Solution

Questions?