Parallel Systems Programming Models: Processes + Threads

Chris Rossbach + Calvin Lin

CS380p

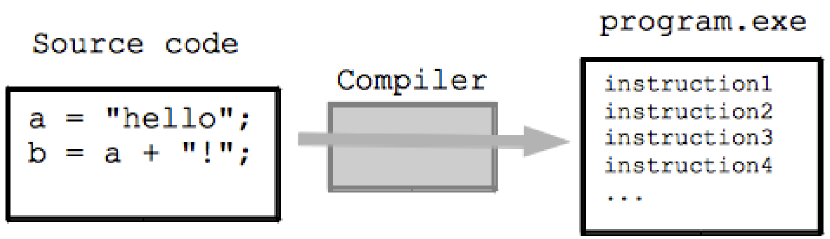
Outline for Today

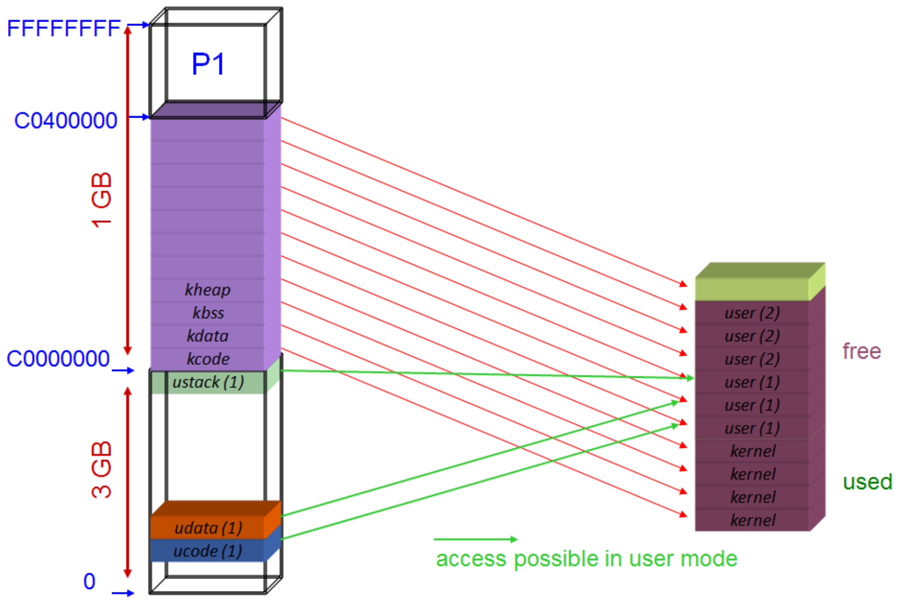
- Parallel programming models
  - Processes
  - Threads
  - Fibers
  - pthreads

Acknowledgments: some materials in this lecture borrowed from or built on materials 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*

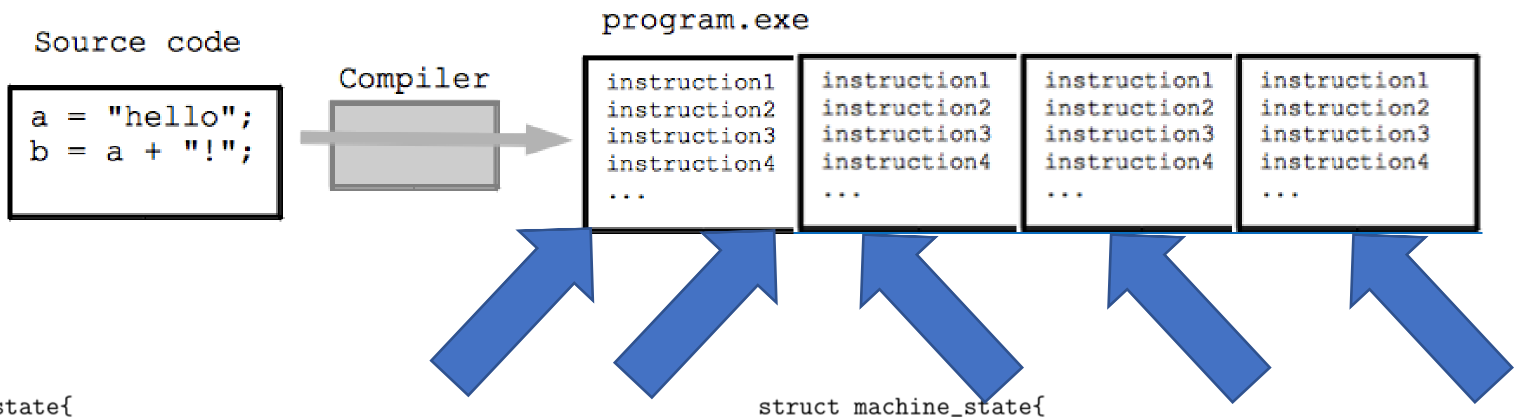
Programming and Machines: a mental model






```
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;
}
```

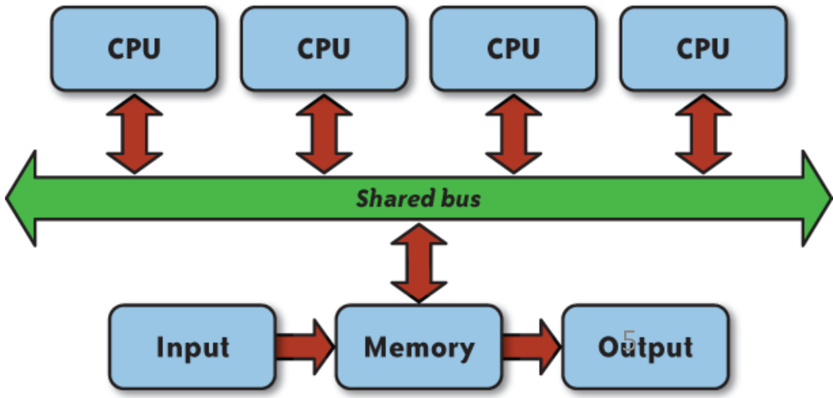
Parallel Machines: a mental model




```
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;
}
```

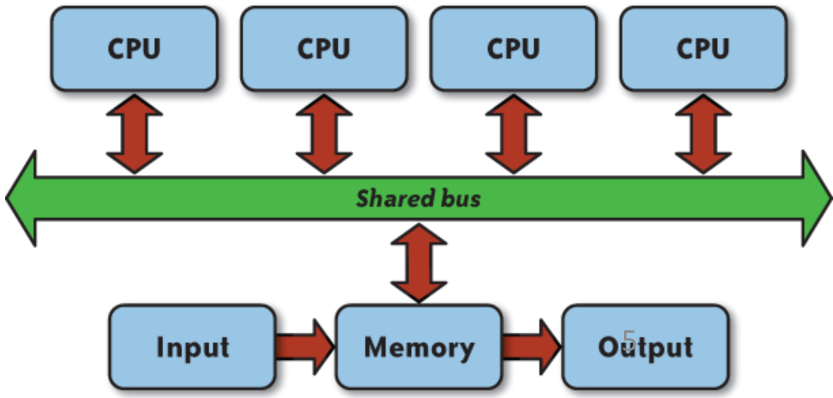
```
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode)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

Programming Models for Concurrency



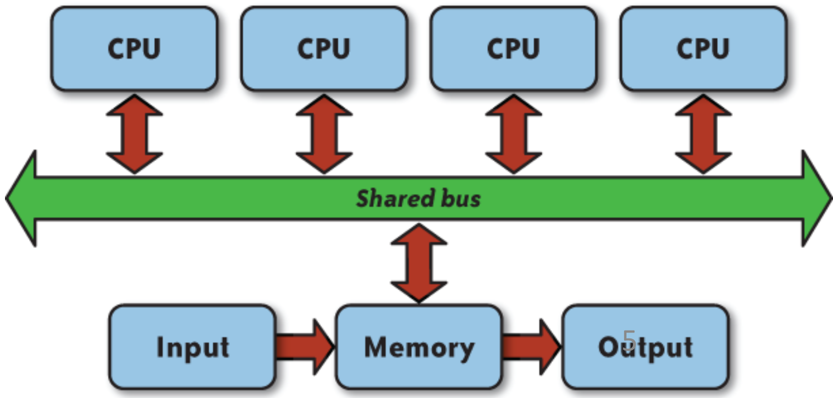
Programming Models for Concurrency

- Concrete model:
  - CPU(s) execute instructions sequentially



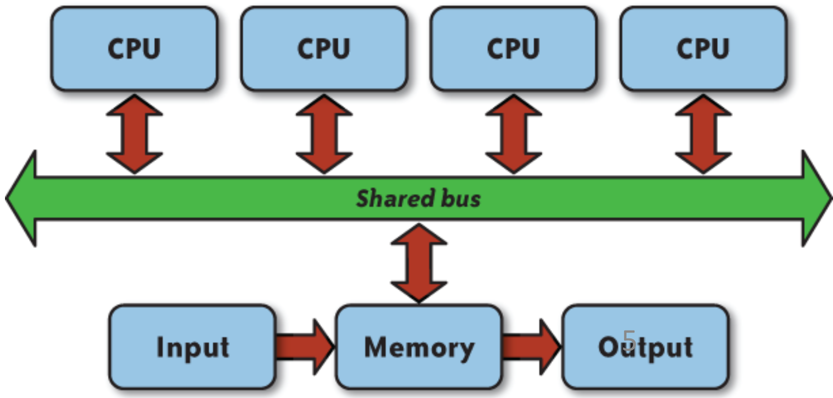
Programming Models for Concurrency

- Concrete model:
  - CPU(s) execute instructions sequentially
- Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer



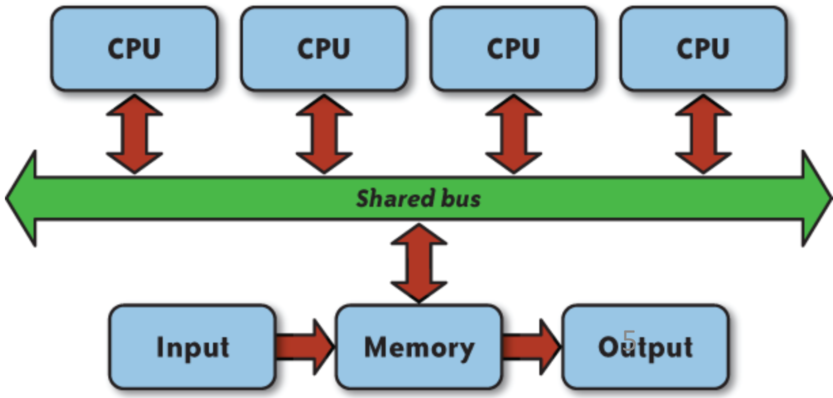
Programming Models for Concurrency

- Concrete model:
  - CPU(s) execute instructions sequentially
- Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer
- Techniques/primitives
  - Threads/Processes
  - Message passing vs shared memory
  - Preemption vs Non-preemption



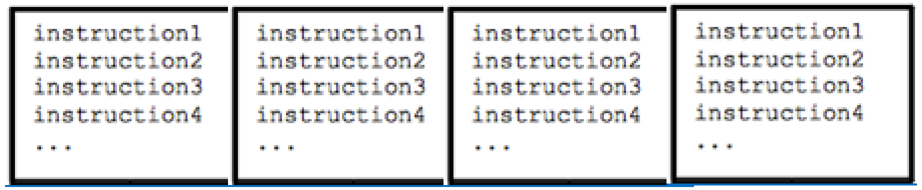
Programming Models for Concurrency

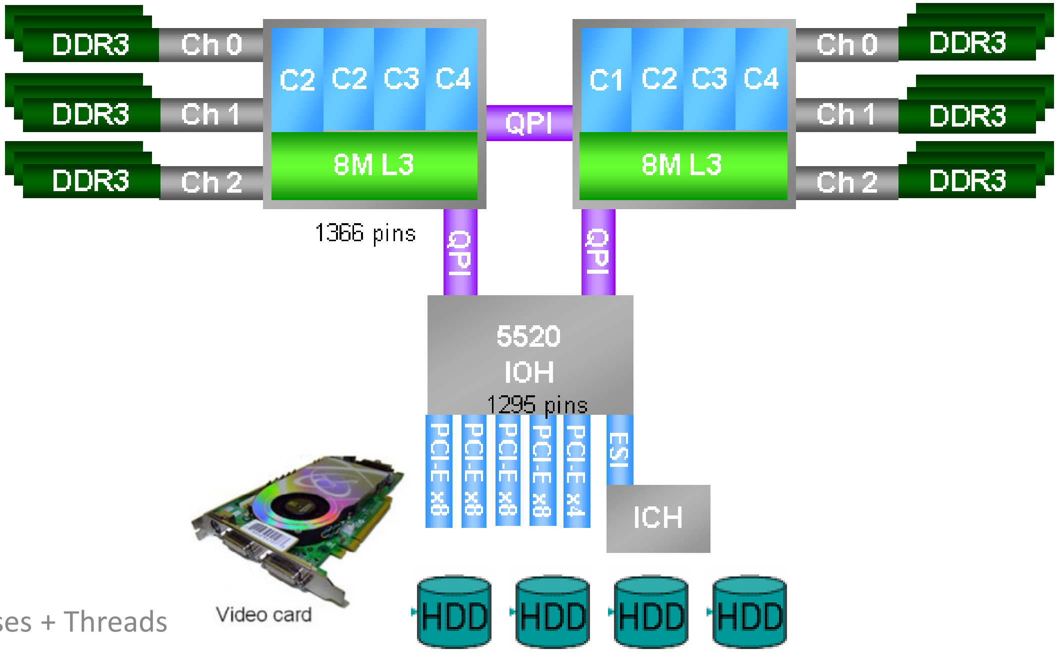
- Concrete model:
  - CPU(s) execute instructions sequentially
- Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer
- Techniques/primitives
  - Threads/Processes
  - Message passing vs shared memory
  - Preemption vs Non-preemption
- Dimensions/techniques not always orthogonal



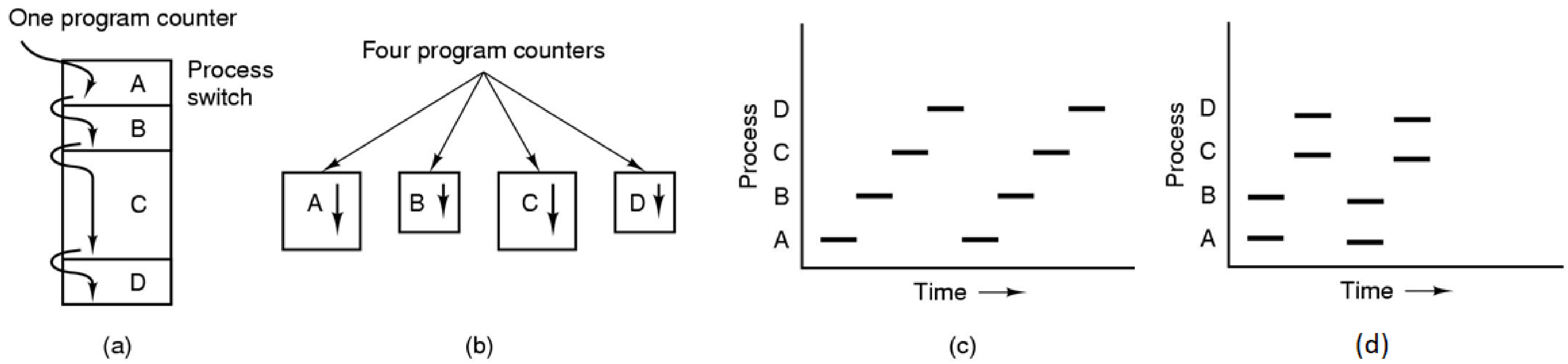
Processes and Threads

- Abstractions
- Unit of Allocation/Containment
- State
  - Where is shared state?
  - How is it accessed?
  - Is it mutable?



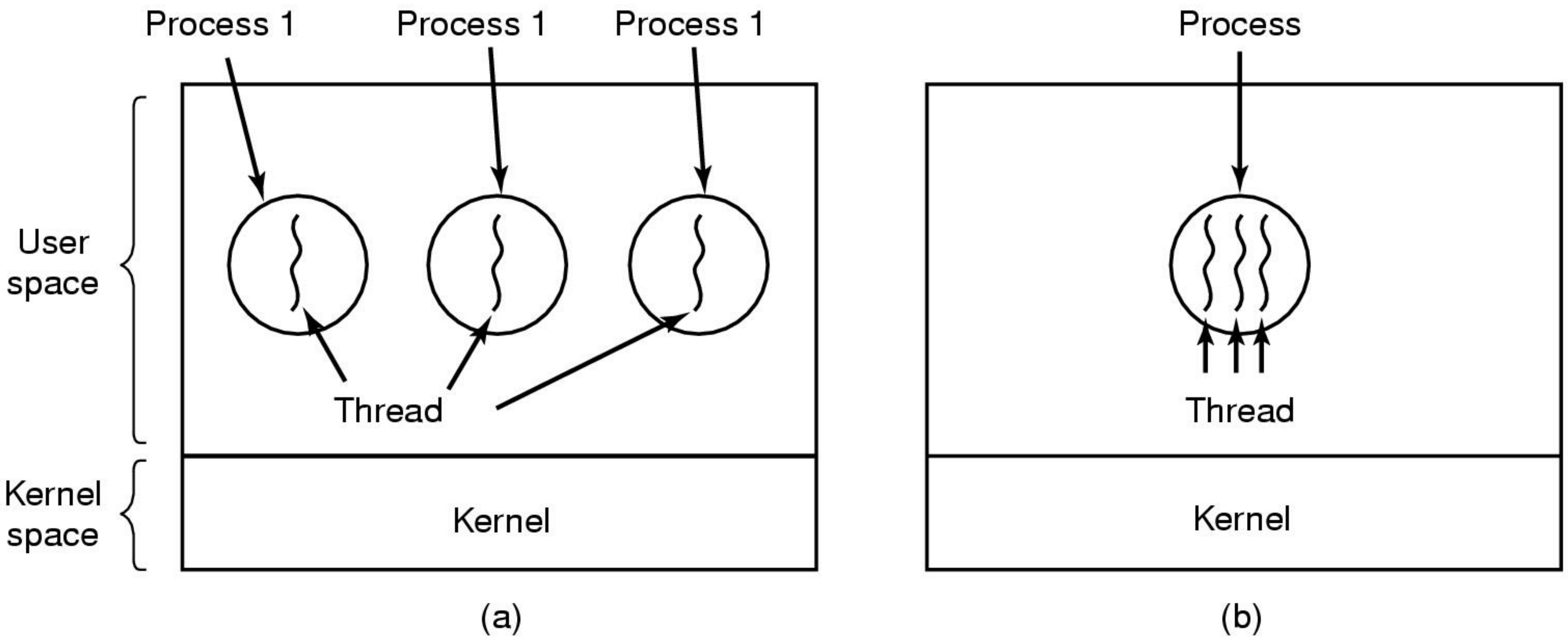


Processes
The Process Model



- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Uniprocessor: Only one program active at any instant
- Multiprocessor: two run in parallel per quantum

Threads
The Thread Model



(a) Three processes each with one thread
(b) One process with three threads

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	

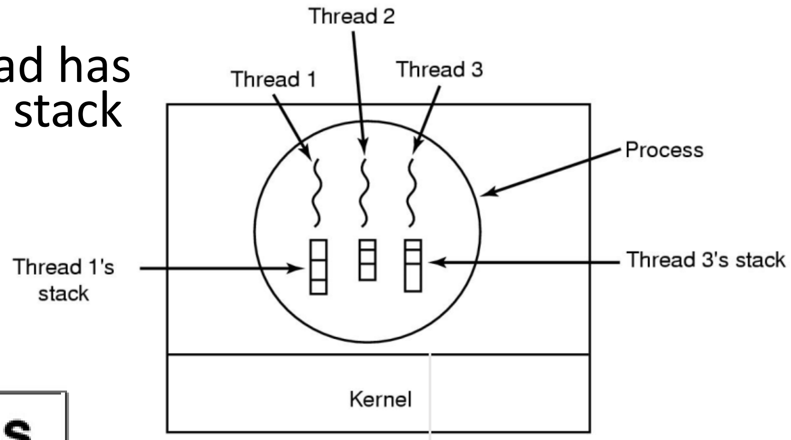
The Thread Model

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	

- Items shared by all threads in a process

The Thread Model

Each thread has its own stack

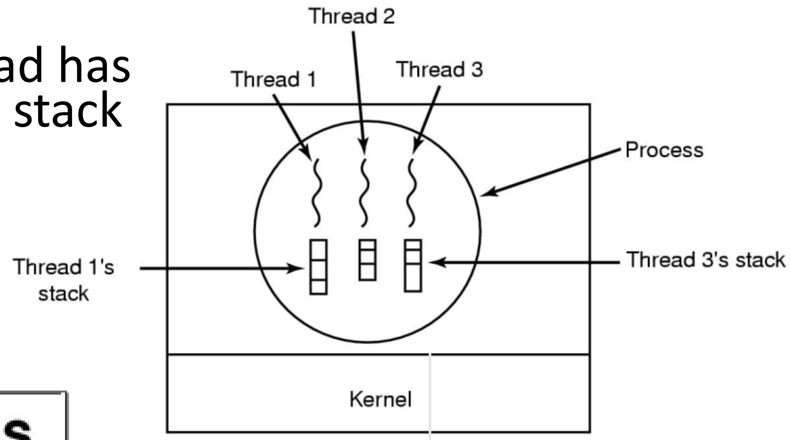


Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

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

The Thread Model

Each thread has its own stack

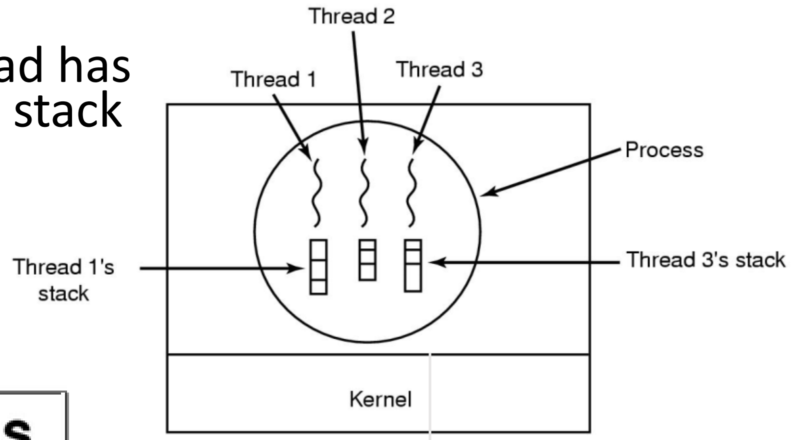


Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Items shared by all threads in a process
- Items private to each thread
- ***Decouples memory and control abstractions!***

The Thread Model

Each thread has its own stack

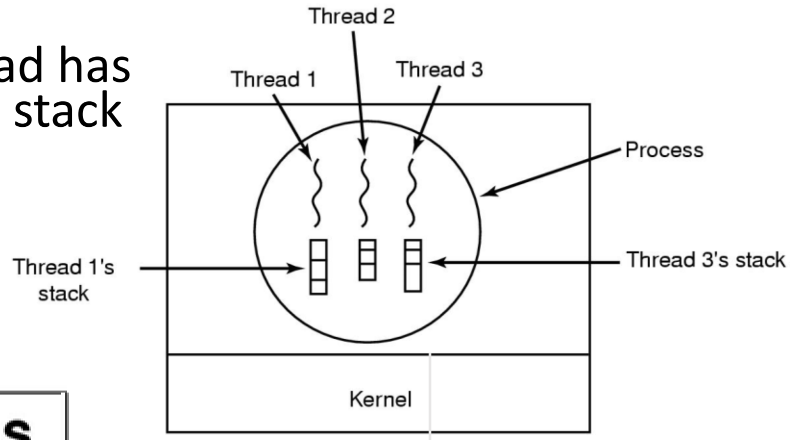


Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Items shared by all threads in a process
- Items private to each thread
- ***Decouples memory and control abstractions!***
- ***What is the advantage of that?***

The Thread Model

Each thread has its own stack



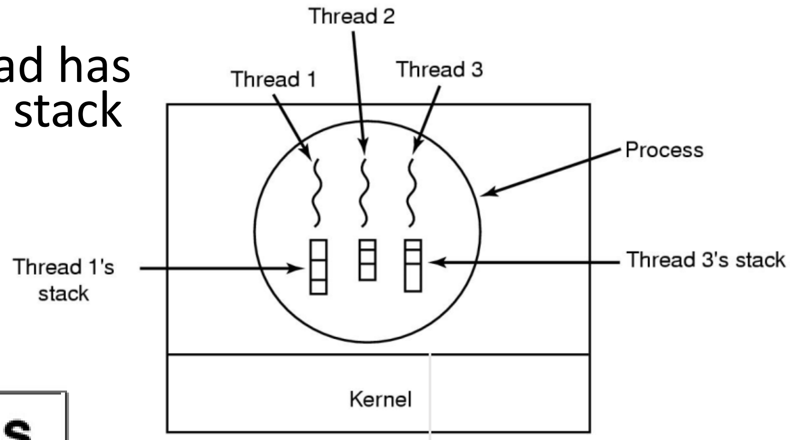
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Process management	Memory management	File management
	Pointer to text segment	Root directory
Program status word	Pointer to data segment	Working directory
Process state	Pointer to stack segment	File descriptors
Process ID		User ID
Parent process		Group ID
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?***

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	

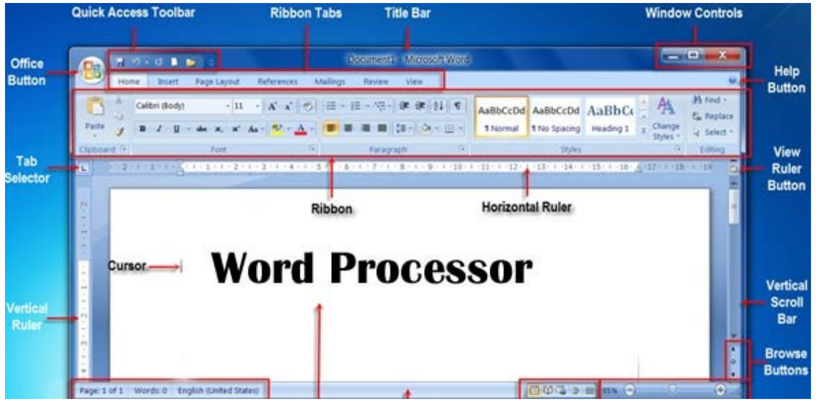
Process management	Memory management	File management
	Pointer to text segment	Root directory
Program status word	Pointer to data segment	Working directory
	Pointer to stack segment	File descriptors
Process state		User ID
		Group ID
Process ID		
Parent process		
Process group		

- Items shared by all threads in a process
- Items private to each thread
- ***Decouples memory and control abstractio***
- ***What is the advantage of that?***

How can we share mutable state across threads?
How can we share mutable state across processes?

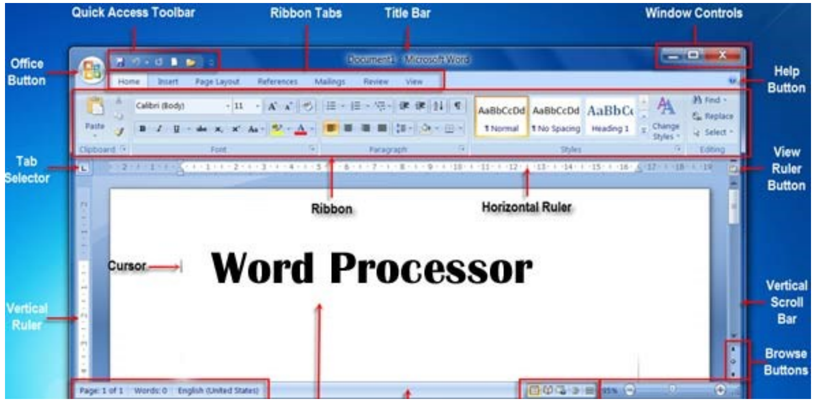
Using threads

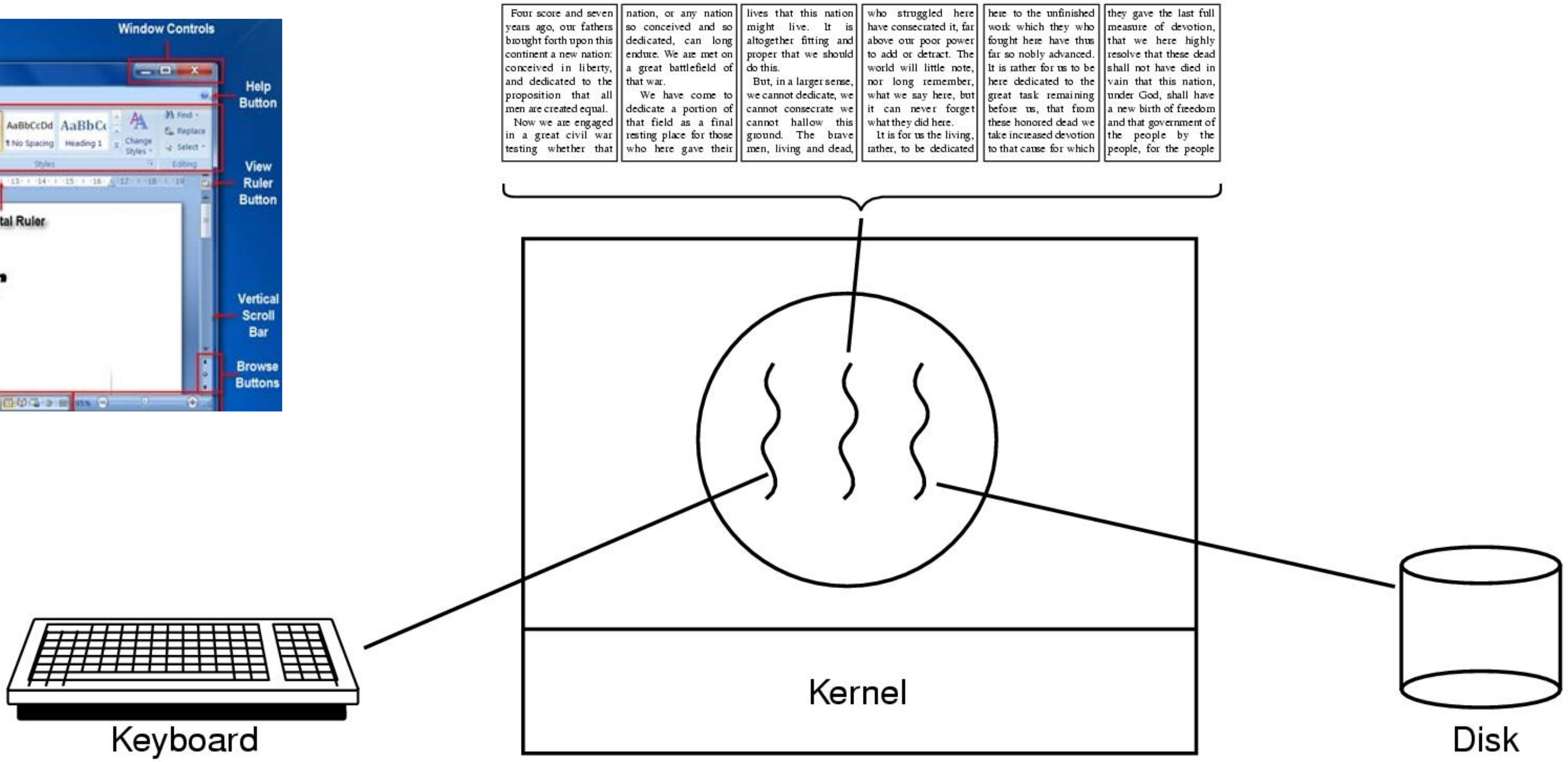
Ex. How might we use threads in a word processor program?



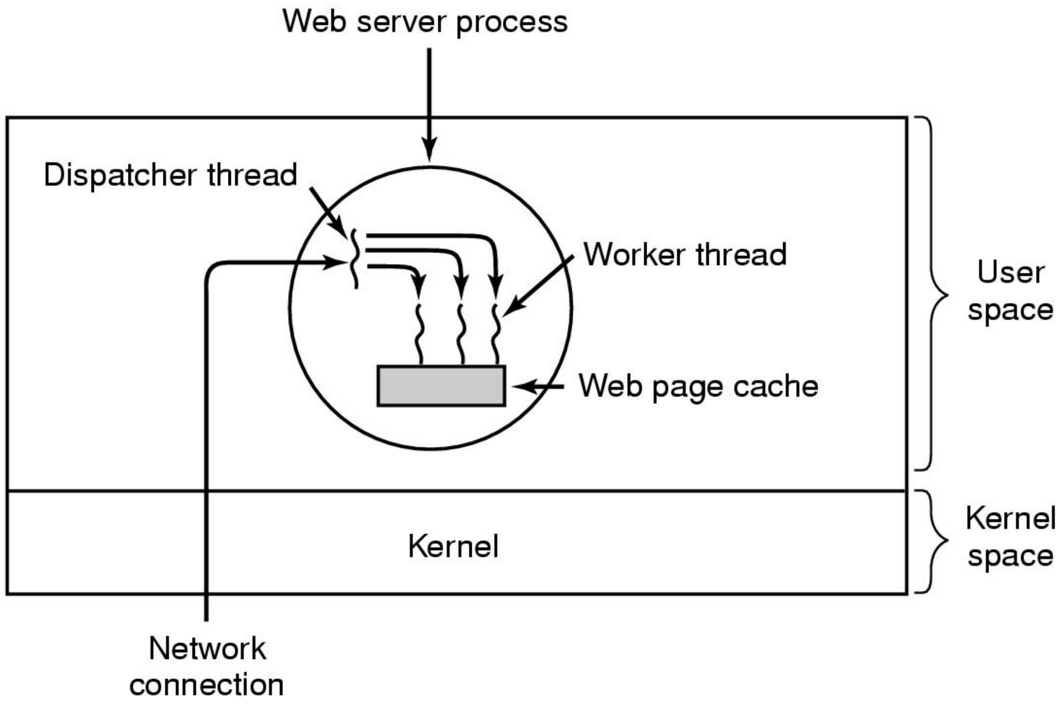
Using threads

Ex. How might we use threads in a word processor program?





Thread Usage



A multithreaded Web server

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
       read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

(a) Dispatcher thread

(b) Worker thread

Where to Implement Threads:

Where to Implement Threads:

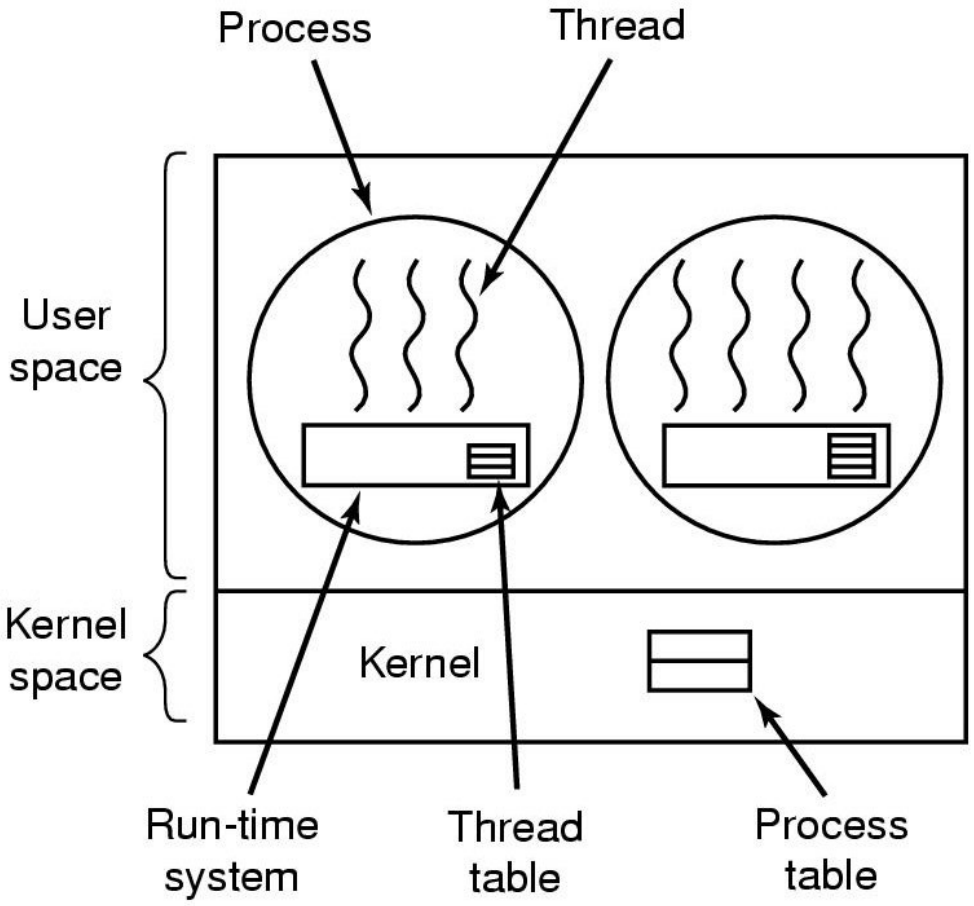
User Space

Kernel Space

Where to Implement Threads:

User Space

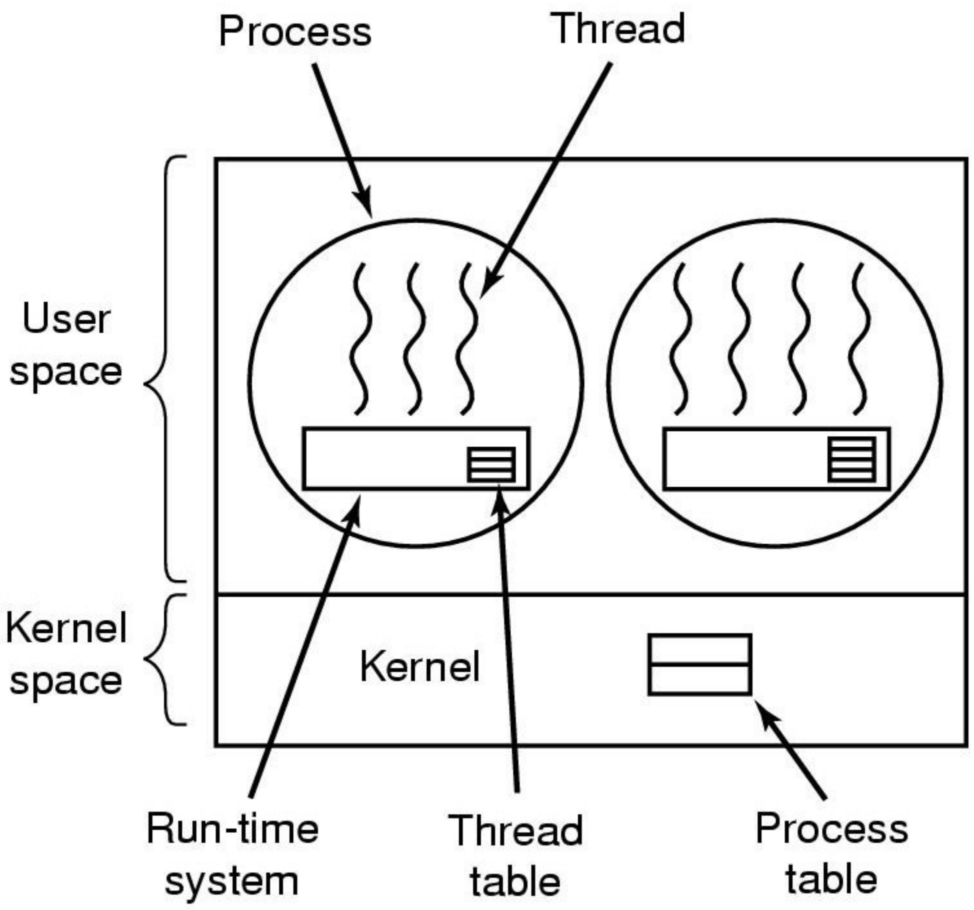
Kernel Space



A user-level threads package

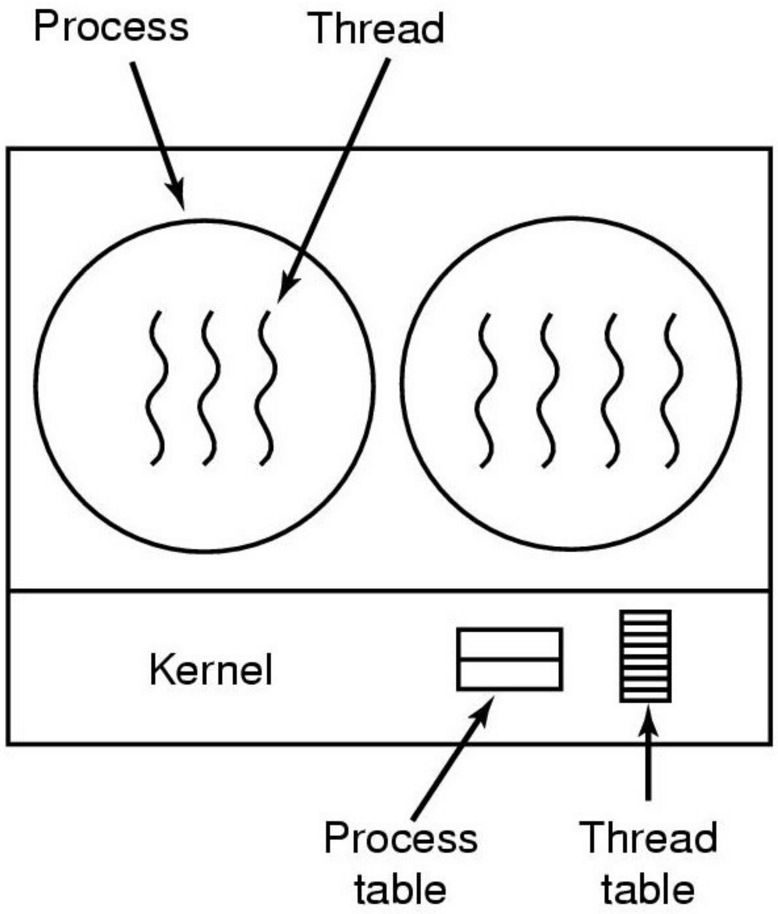
Where to Implement Threads:

User Space



A user-level threads package

Kernel Space

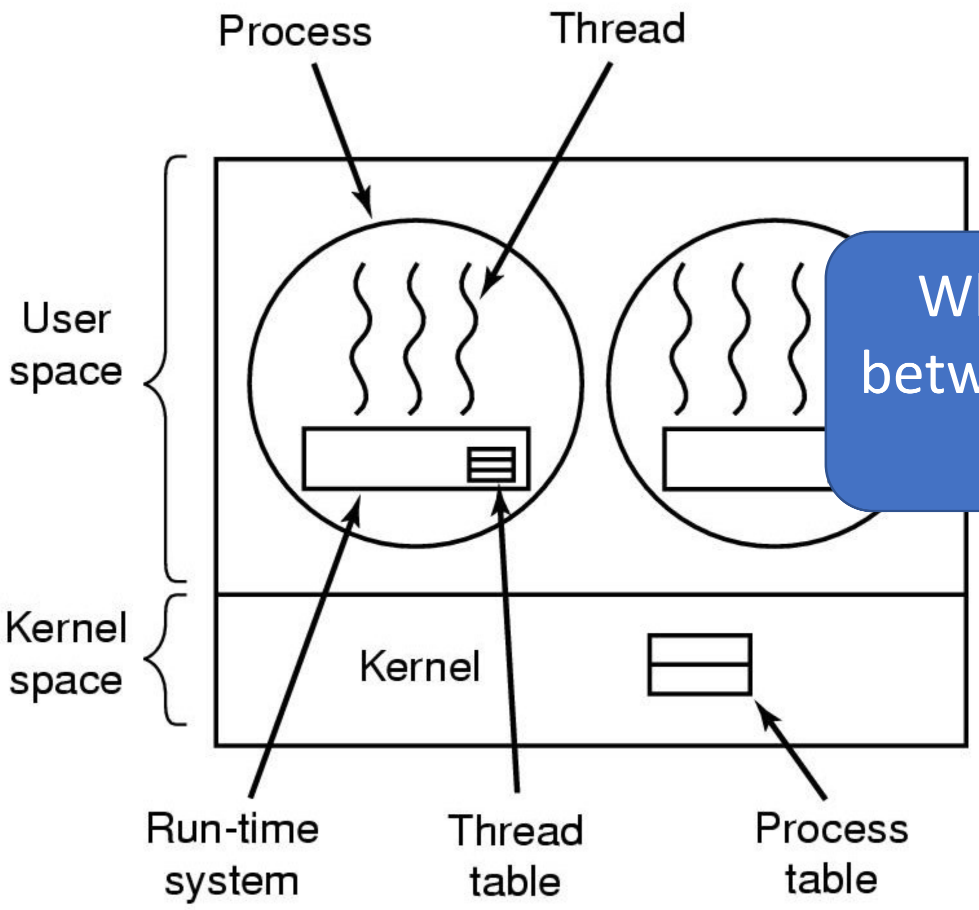


A threads package managed by the kernel

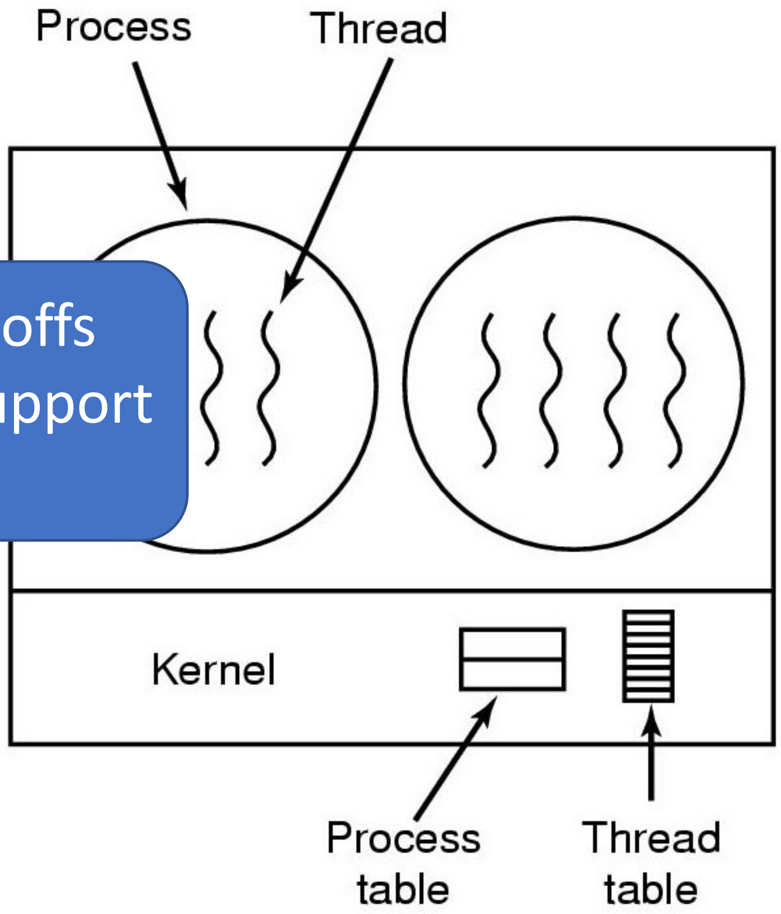
Where to Implement Threads:

User Space

Kernel Space



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



A user-level threads package

A threads package managed by the kernel

Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management

Execution Context Management

"Task" == "Flow of Control"
"Stack" == Task State

Task Management

- Preemptive
  - Interleave on uniprocessor
  - Overlap on multiprocessor

Execution Context Management

"Task" == "Flow of Control"
"Stack" == Task State

Task Management

- Preemptive
  - Interleave on uniprocessor
  - Overlap on multiprocessor
- Serial
  - One at a time, no conflict

Execution Context Management

“Task” == “Flow of Control”
“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

Execution Context Management

"Task" == "Flow of Control"
"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

Fibers

Fibers

- Cooperative tasks
  - most desirable when reasoning about concurrency
  - usually associated with event-driven programming

Fibers

- 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 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

Threads vs Fibers

- Like threads, *just an abstraction* for flow of control

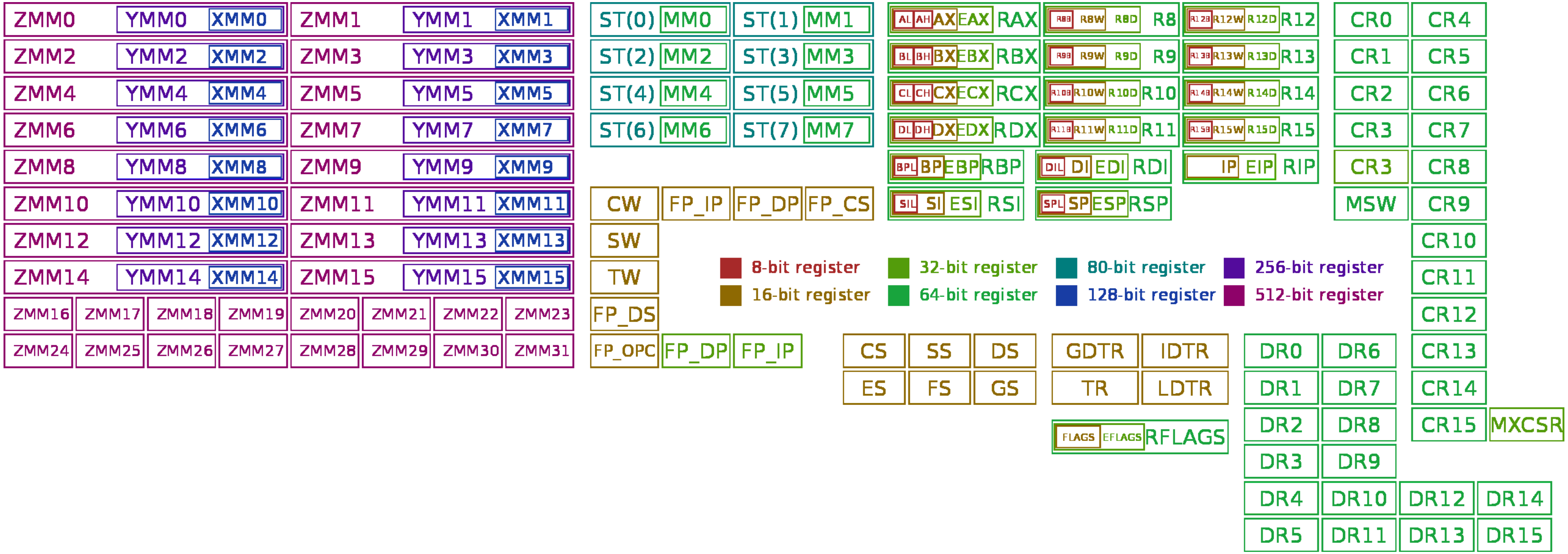
Threads vs Fibers

- Like threads, *just an abstraction* for flow of control
- *Lighter weight* than threads
  - In Windows, just a stack, subset of arch. registers, non-preemptive
  - stack management/impl has interplay with exceptions
  - Can be completely exception safe

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
  - stack management/impl has interplay with exceptions
  - Can be completely exception safe
- ***Takeaway***: diversity of abstractions/containers for execution flows

x86_64 Architectural Registers

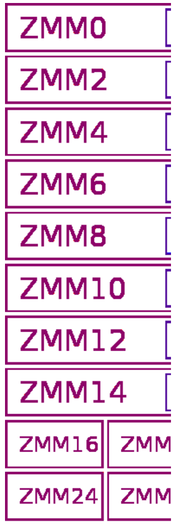


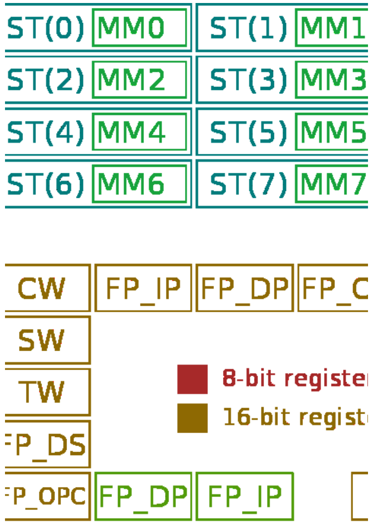
- *Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525*

Linux x86_64 context switch *excerpt*

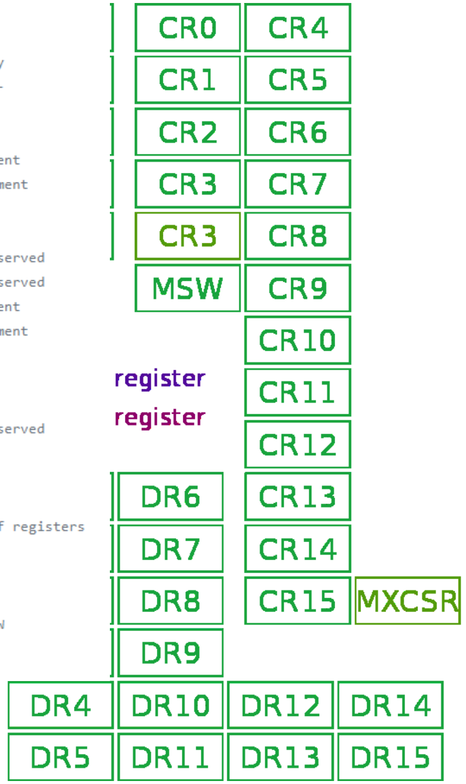
Complete fiber context switch on Unix and Windows

```c
/*
 *      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);
```



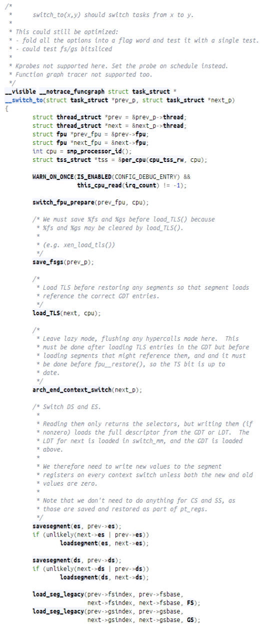



```c
* The AMD64 architecture provides 16 general 64-bit registers together with 16
* 128-bit SSE registers, overlapping with 8 legacy 80-bit x87 floating point
* registers.
*
*              Both                 Unix only            Windows only
*              ----                 ---------         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

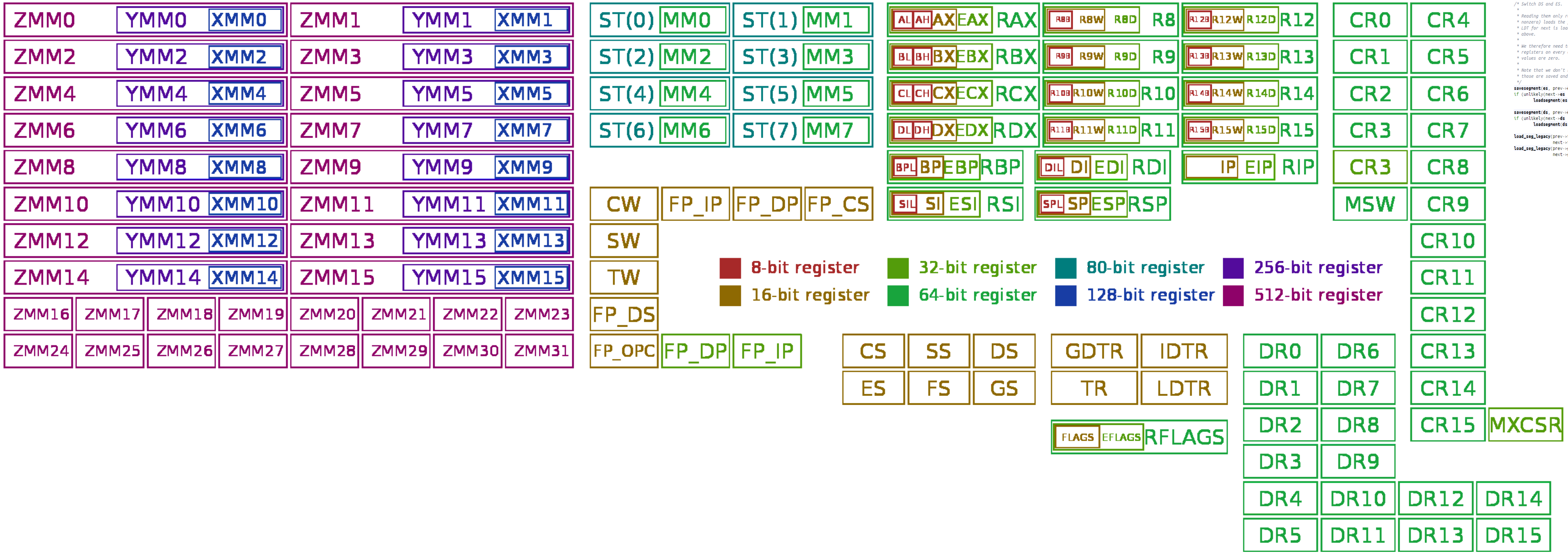


- *Reg*

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

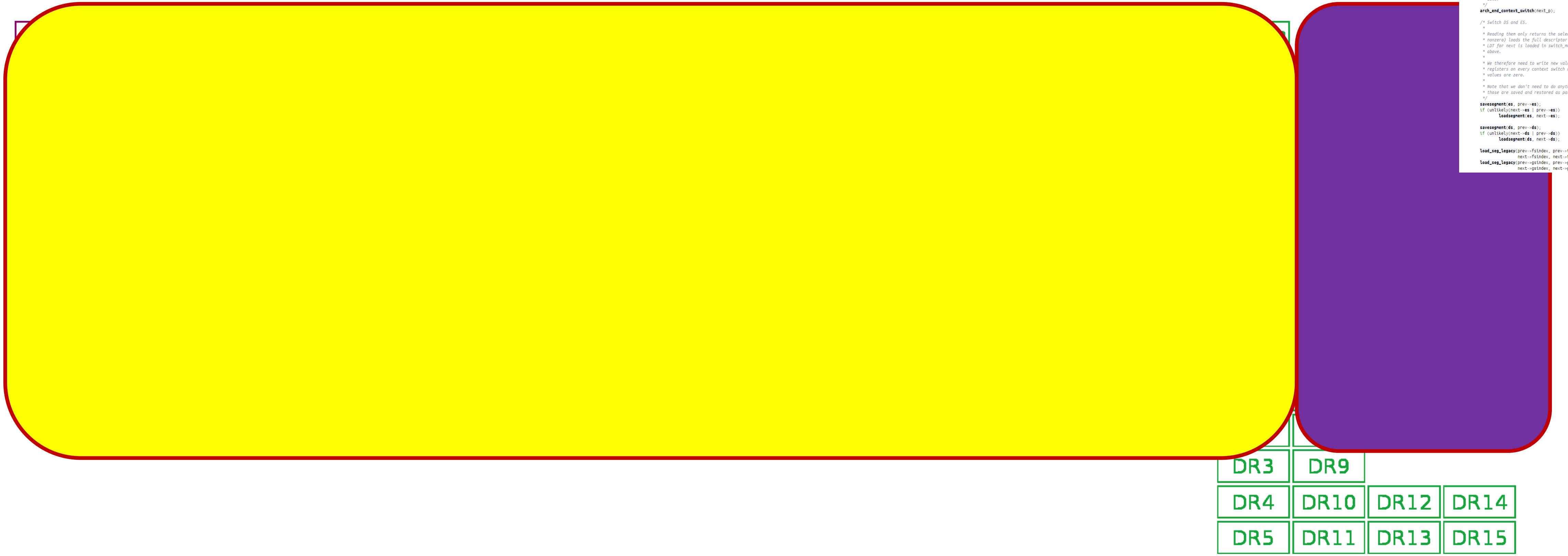


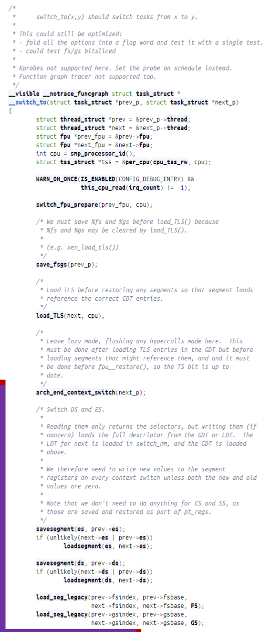
x86_64 Registers and Threads



- *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 Threads



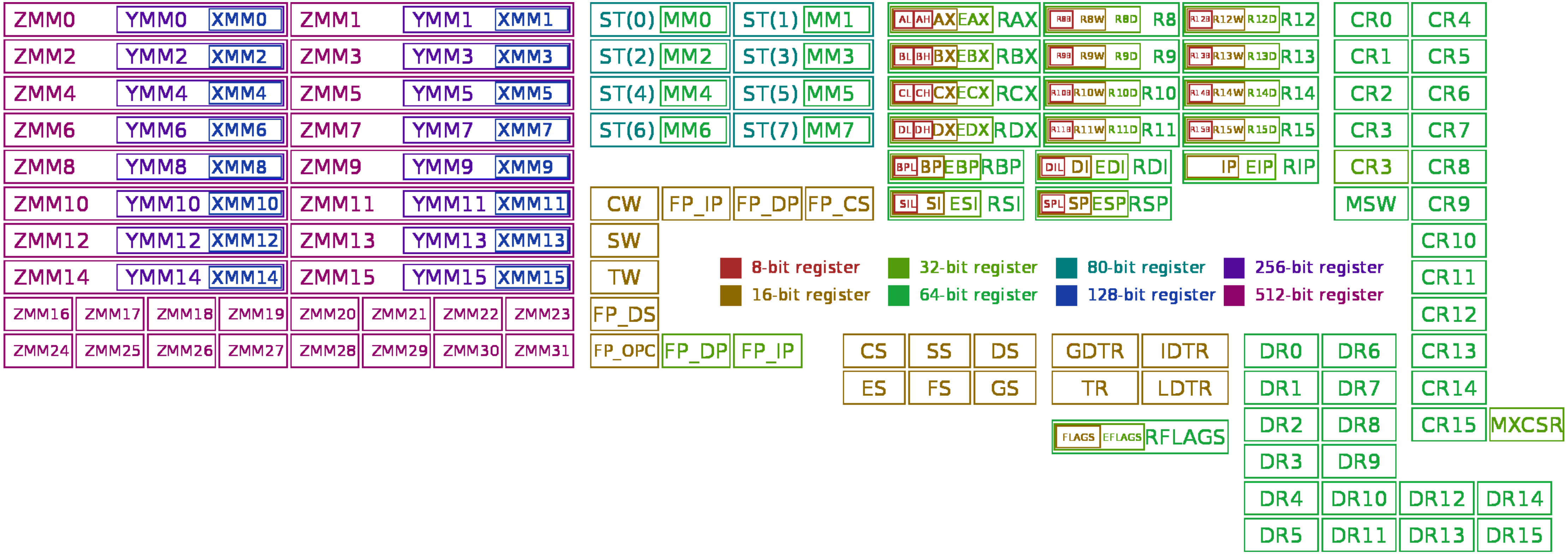


```c
/*
 *      switch_to(x,y) should switch tasks from x to y.
 *
 * This could still be optimized:
 * - fold all the options into a flag word and test it with a single test.
 * - could test fs/gs bitsliced
 *
 * Kprobes not supported here. Set the probe on schedule instead.
 * Function graph tracer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

x86_64 Registers and Fibers

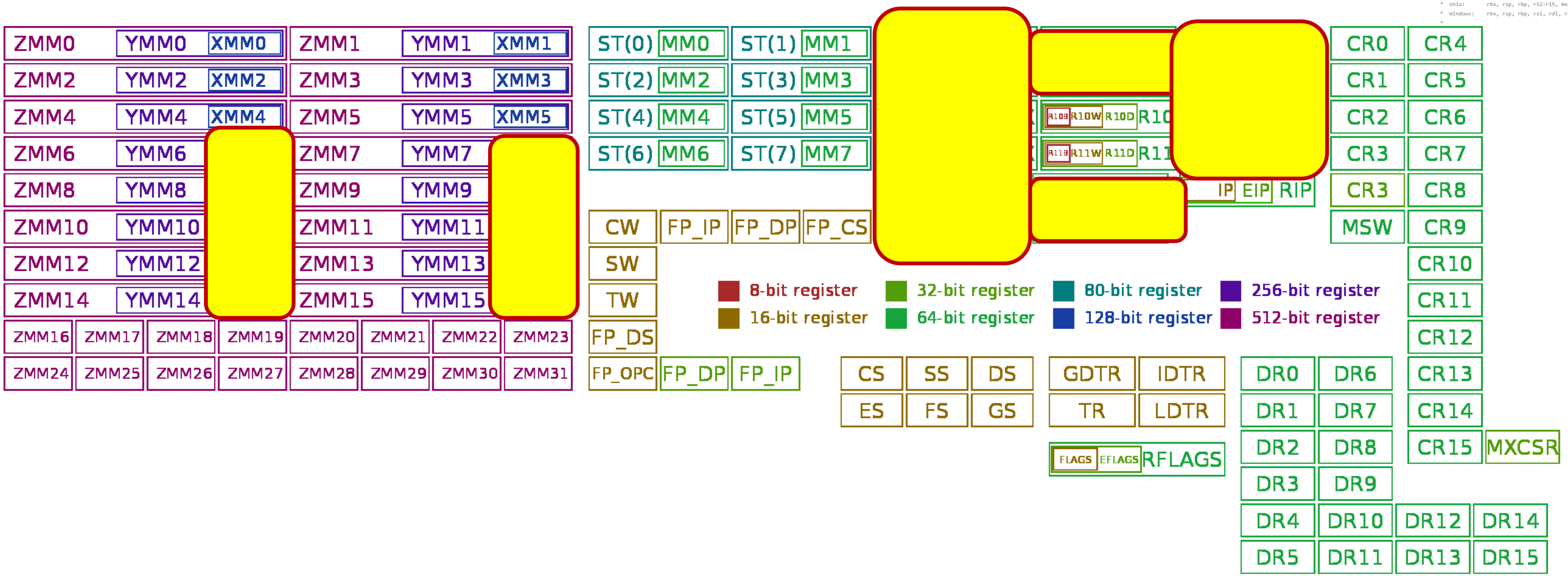
```
* The AMD64 architecture provides 16 general 64-bit registers together with 16
* 128-bit SSE registers, overlapping with 8 legacy 80-bit x87 floating point
* registers.
*
*              Both                Unix only          Windows only
*              ----                ---------          ------------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



- *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

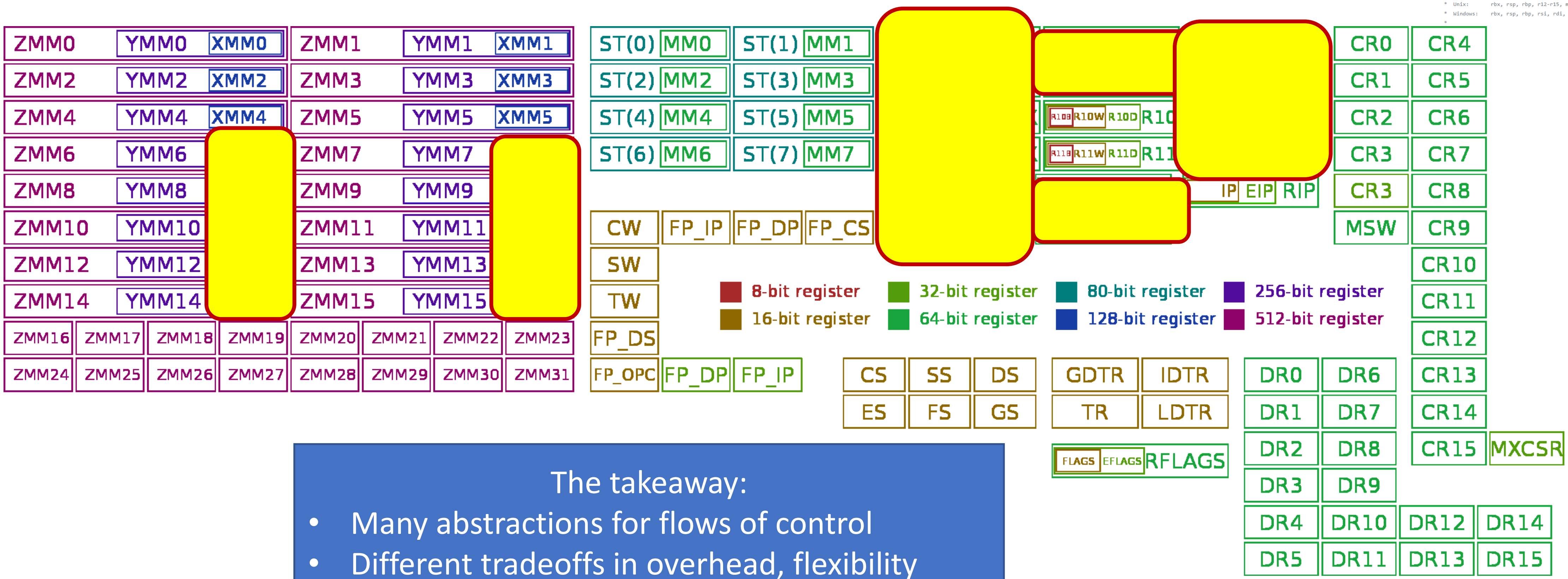
```
* The AMD64 architecture provides 16 general 64-bit registers together with 16
* 128-bit SSE registers, overlapping with 8 legacy 80-bit x87 floating point
* registers.
*
*              Both                  Unix only           Windows only
*              ----                  ---------         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



- *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

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;
  - Don't always 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

Pthread Spinlock

- Type: `pthread_spinlock_t`

Pthread Spinlock

- Type: `pthread_spinlock_t`

```
int pthread_spinlock_init(pthread_spinlock_t *lock);
```

Pthread Spinlock

- Type: `pthread_spinlock_t`

```
int pthread_spinlock_init(pthread_spinlock_t *lock);
int pthread_spinlock_destroy(pthread_spinlock_t *lock);
```

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);
```

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);
```

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);
```

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);
```

Lab #1

- Basic synchronization, prefix sum
- http://www.cs.utexas.edu/~rossbach/cs380p/lab/lab1.html

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