EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

credit Christopher Kruegel, UCSB & Manuel Egele.

What do we need to know for Assignment 2 ???

Lets look at the stack

Procedures

```
int f(int x) {return   x + 1; }
int g(int x) {return    f(x); }
int h(int x) {return f(x *2); }
```

Procedures (functions) are intrinsically linked to the stack

– Provides space for local variables
– Records where to return to
– Used to pass arguments (sometimes)

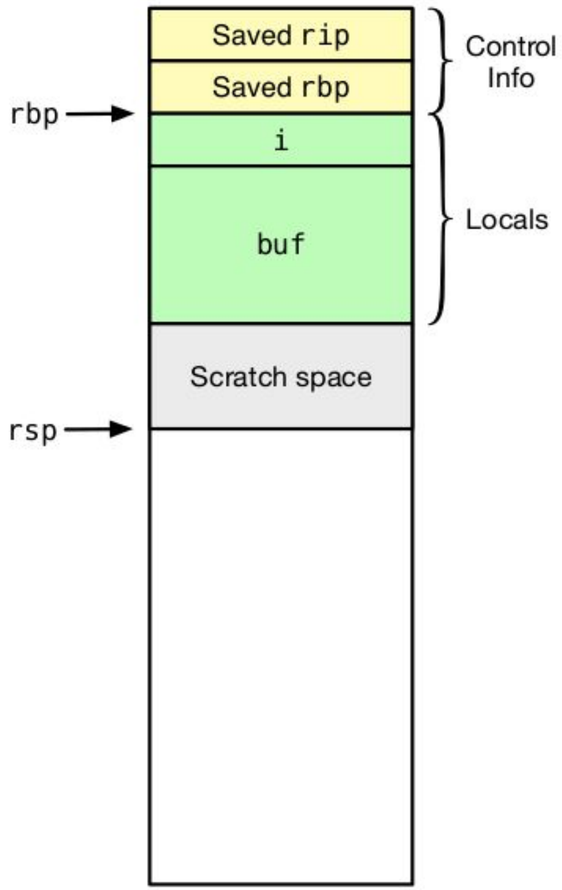
Implemented using stack frames

– Also known as activation records

Procedures: Calling and Returning

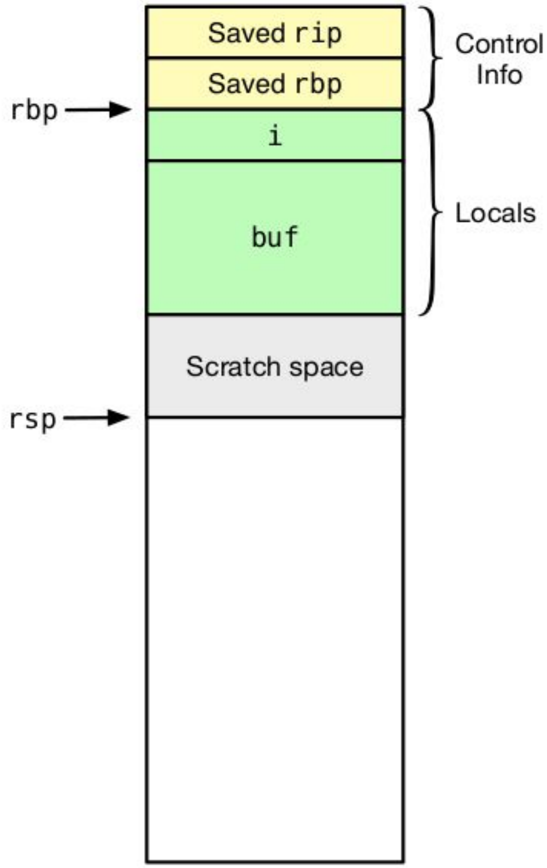
Instruction	Effect	Description
`call` `x`	rsp $\leftarrow$ rsp - 8	Decrement rsp by 8
	Mem(rsp) $\leftarrow$ Succ(rip)	Store successor
	rip $\leftarrow$ Addr(x)	Jump to address
`ret`	rip $\leftarrow$ Mem(rsp)	Pop successor into rip
	rsp $\leftarrow$ rsp + 8	Increment rsp by 8

Stack Frame




```
int auth(const char* user) {
    size_t i;
    char buf[16];
    strncpy(buf, user, sizeof(buf));
    buf[sizeof(buf) - 1] = '\0';
    for (i = 0; i < sizeof(buf); i++)
        buf[i] ^= 0xe5;
    return !memcmp(buf, "secret", 6);
}
```

Stack Frame



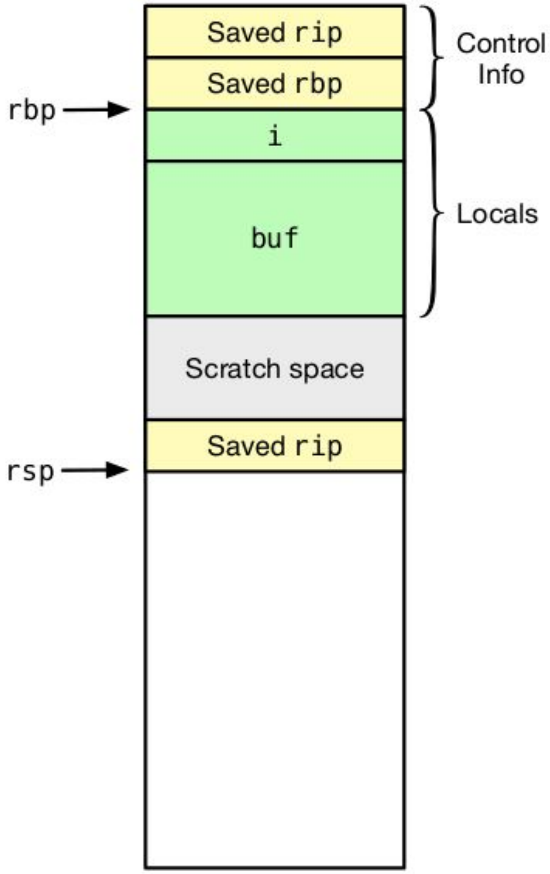

```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

Remember

Instruction	Effect	Description
`call x`	rsp ← rsp - 8	Decrement rsp by 8
	Mem(rsp) ← Succ(rip)	Store successor
	rip ← Addr(x)	Jump to address
`ret`	rip ← Mem(rsp)	Pop successor into rip
	rsp ← rsp + 8	Increment rsp by 8

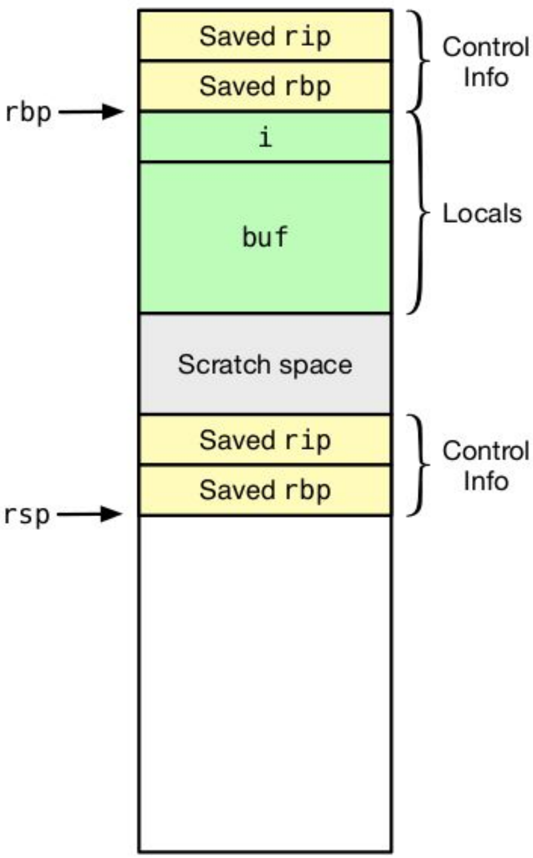
Stack Frame




```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

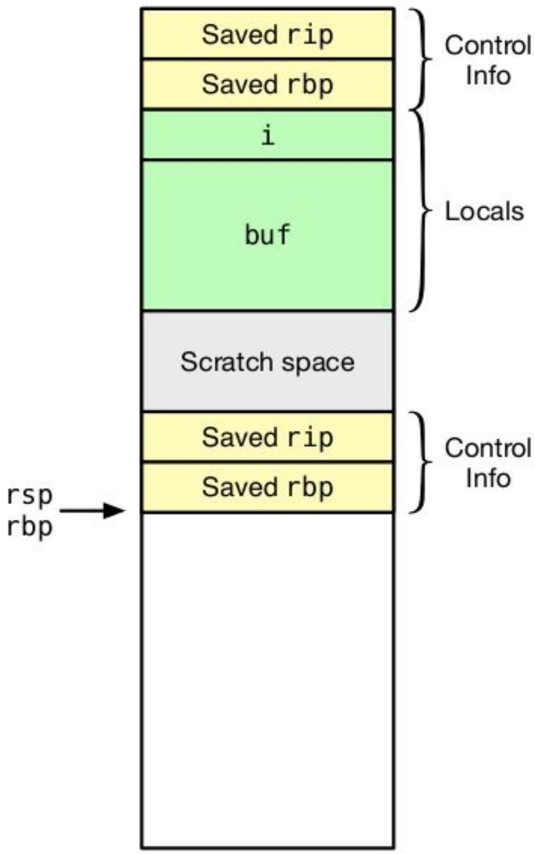
Stack Frame




```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

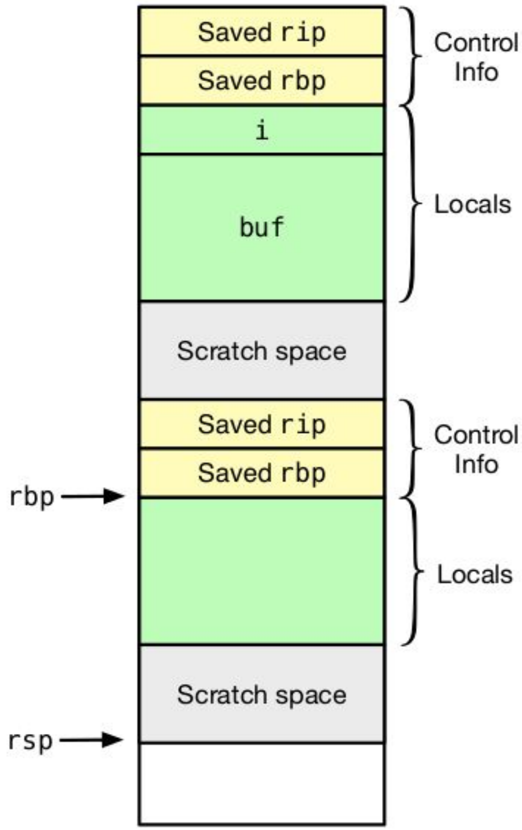
Stack Frame




```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

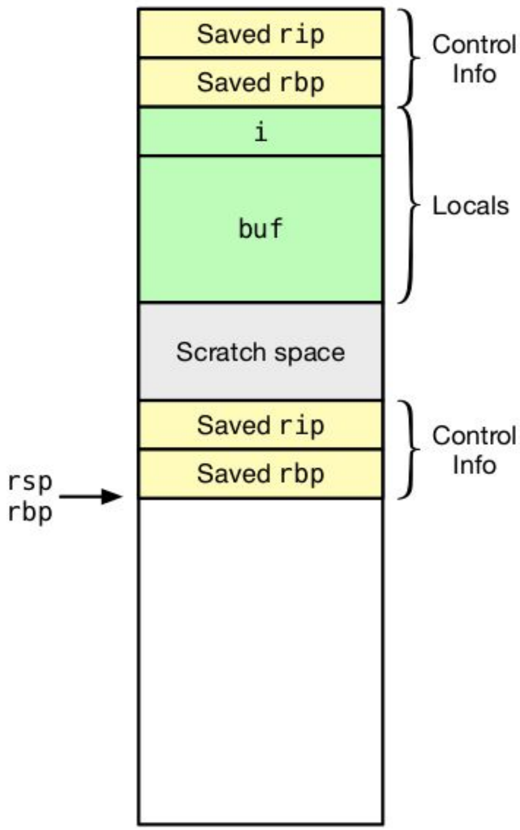
Stack Frame




```
auth:
        ...
        mov rdi, rax
        call strncpy
        mov byte [rbp-0x11], 0x00
        ...

strncpy:
        push rbp
        mov rbp, rsp
        sub rsp, 0x30
        ...
        add rsp, 0x30
        pop rbp
        ret
```

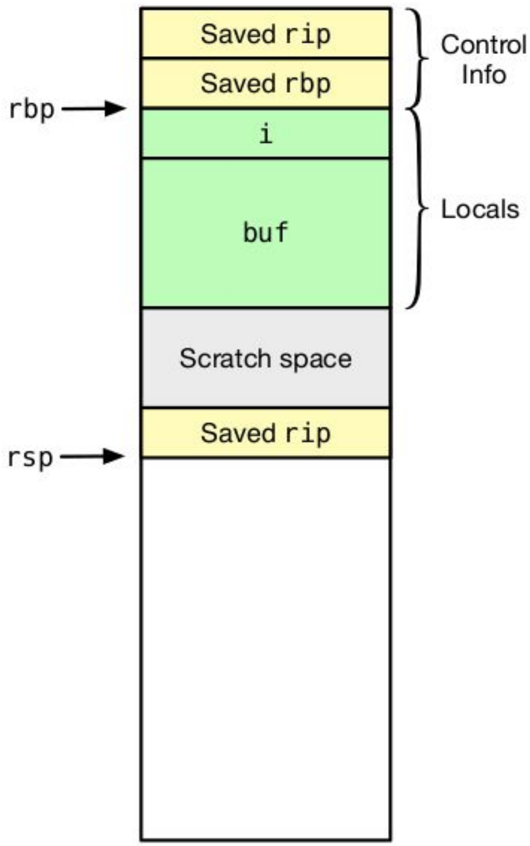
Stack Frame




```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

Stack Frame



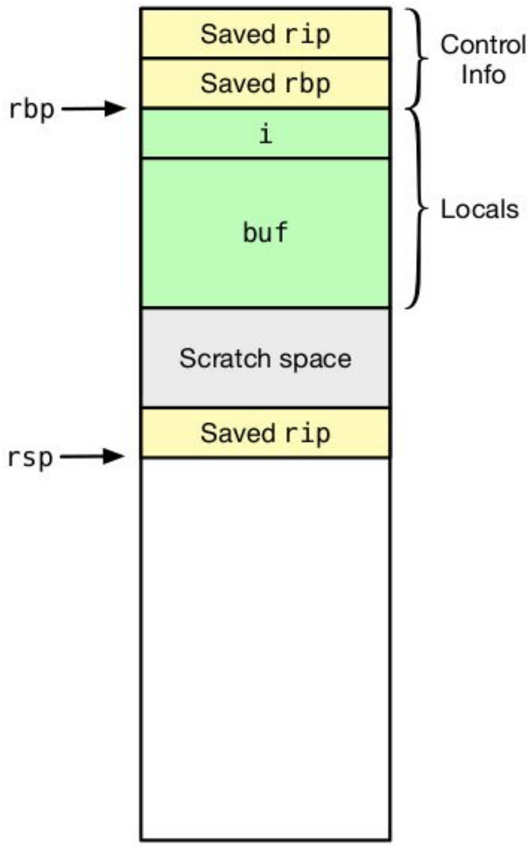

```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

Remember

Instruction	Effect	Description
`call x`	rsp ← rsp - 8	Decrement rsp by 8
	Mem(rsp) ← Succ(rip)	Store successor
	rip ← Addr(x)	Jump to address
`ret`	rip ← Mem(rsp)	Pop successor into rip
	rsp ← rsp + 8	Increment rsp by 8

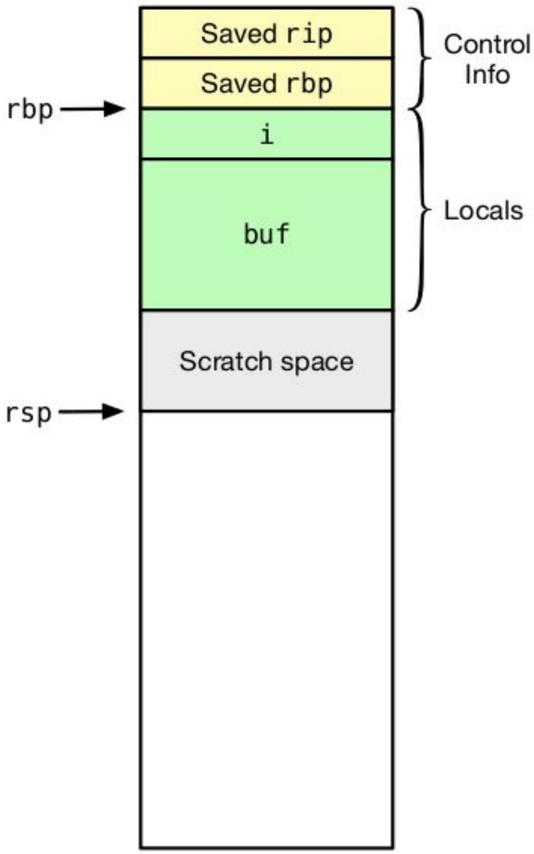
Stack Frame




```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

Stack Frame




```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

How do we pass arguments?

Procedure Arguments

- **Standards (*calling conventions*) exist for argument passing**
  - Specify where arguments are passed (registers, stack)
  - Specify the caller and callee's responsibilities
    - Who deallocates argument space on the stack?
    - Which registers can be clobbered, and who must save them?
- **Why do we need standards?**
  - There are many ways to pass arguments
  - How would code compiled by different developers and toolchains interoperate?

Calling Conventions

- **We often speak of *callers* and *callees***
  - Caller: Code that invokes a procedure
  - Callee: Procedure invoked by another function
- **Conventions must specify how registers must be dealt with**
  - Could always save them, but that is inefficient (why?)
  - Usually, some registers can be overwritten (clobbered), others cannot
  - Registers that can be clobbered: *caller* saved
  - Registers that must not be clobbered: *callee* saved

SysV AMD64 ABI

- **x86_64 calling convention used on Linux, Solaris, FreeBSD, Mac OS X**
  - This is what you'll see most often in this course
- **First six arguments passed in registers**
  - rdi, rsi, rdx, rcx, r8, r9
    - Except syscalls, rcx → r10
  - Additional arguments spill to stack
- **Return value in rax**

SysV AMD64 ABI Example

```
int auth( const char * user) {
    size_t i;
    char buf[16];
    strncpy(buf, user, sizeof (buf));

auth:
    push rbp                        ; save previous frame pointer
    mov rbp, rsp                    ; set new frame pointer
    sub rsp, 0x30                   ; allocate space for locals (i, buf)
    movabs rdx, 0x10                ; move sizeof(buf) to rdx
    lea rax, [rbp-0x20]             ; get the address of buf on the stack
    mov qword [rbp-0x08], rdi       ; move user pointer into stack
    mov rsi, qword [rbp-0x08]       ; move user pointer back into rsi
    mov rdi, rax                    ; move buf into rdi
    call strncpy                    ; call strncpy(rdi, rsi, rdx)
    ...
```

Thread Primitives (e.g., pthread_create)

$ man pthread_create

```
PTHREAD_CREATE(3)                Linux Programmer's Manual

NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.
```

Each thread has a separate stack!

Demo setjmp

Preemptive User Mode Threading Library

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

User Mode

- No support from kernel necessary
  - Portable (i.e., works even if kernel does not explicitly support threads)
  - If something goes wrong (e.g., crashes) only your program dies not the entire OS

Threads

- … next slide …

Library

- i.e., only the functions required by the project description
- must not have `main` in your library

Scheduling Algorithms

From Class…

Non-preemptive

- CPU is switched when process
  - has finished
  - executes a yield()
  - blocks

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

Preemptive User Mode Threading Library

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

User Mode

- No support from kernel necessary
  - Portable (i.e., works even if kernel does not explicitly support threads)
  - If something goes wrong (e.g., crashes) only your program dies not the entire OS

Threads

- … next slide …

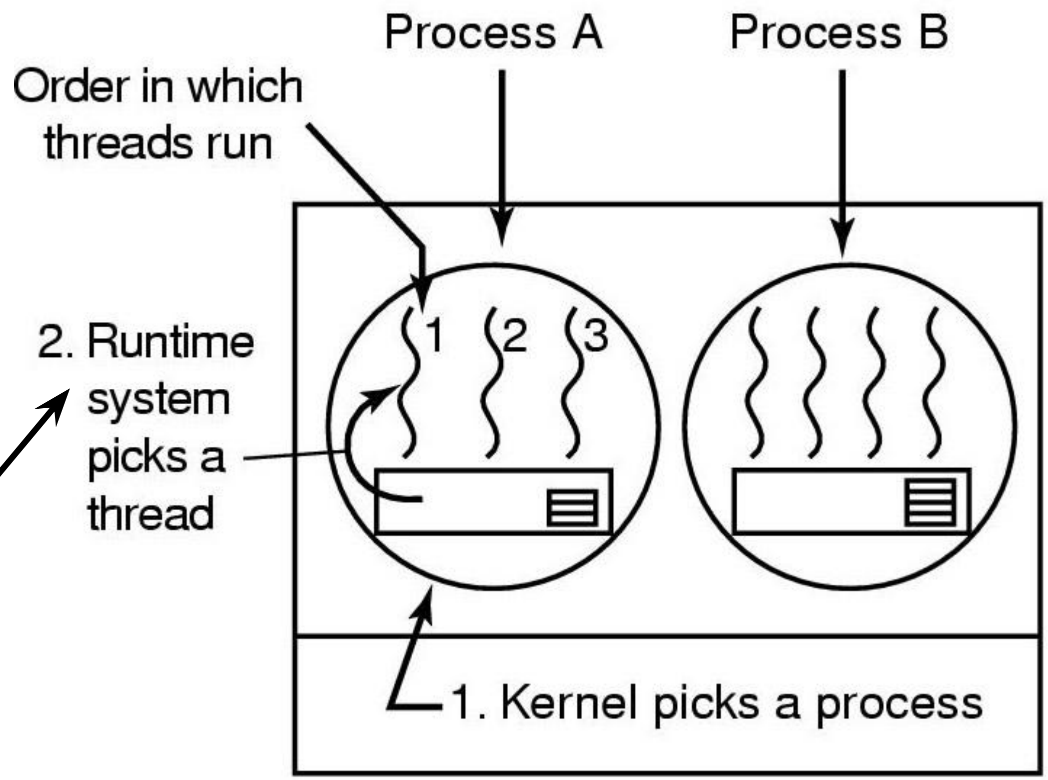
Library

- i.e., only the functions required by the project description
- must not have `main` in your library

Thread Scheduling

From Class…

We will build this!



Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

Preemptive User Mode Threading Library

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

User Mode

- No support from kernel necessary
  - Portable (i.e., works even if kernel does not explicitly support threads)
  - If something goes wrong (e.g., crashes) only your program dies not the entire OS

Threads

- … next slide …

Library

- i.e., only the functions required by the project description
- must not have `main` in your library

Threads

Multiple threads of execution can run in the same *process*

Multiple threads (in the same process) share

- Common address space (shared memory)
- Open files
- Process, user, and group IDs

Cool! But not too important for proj2.

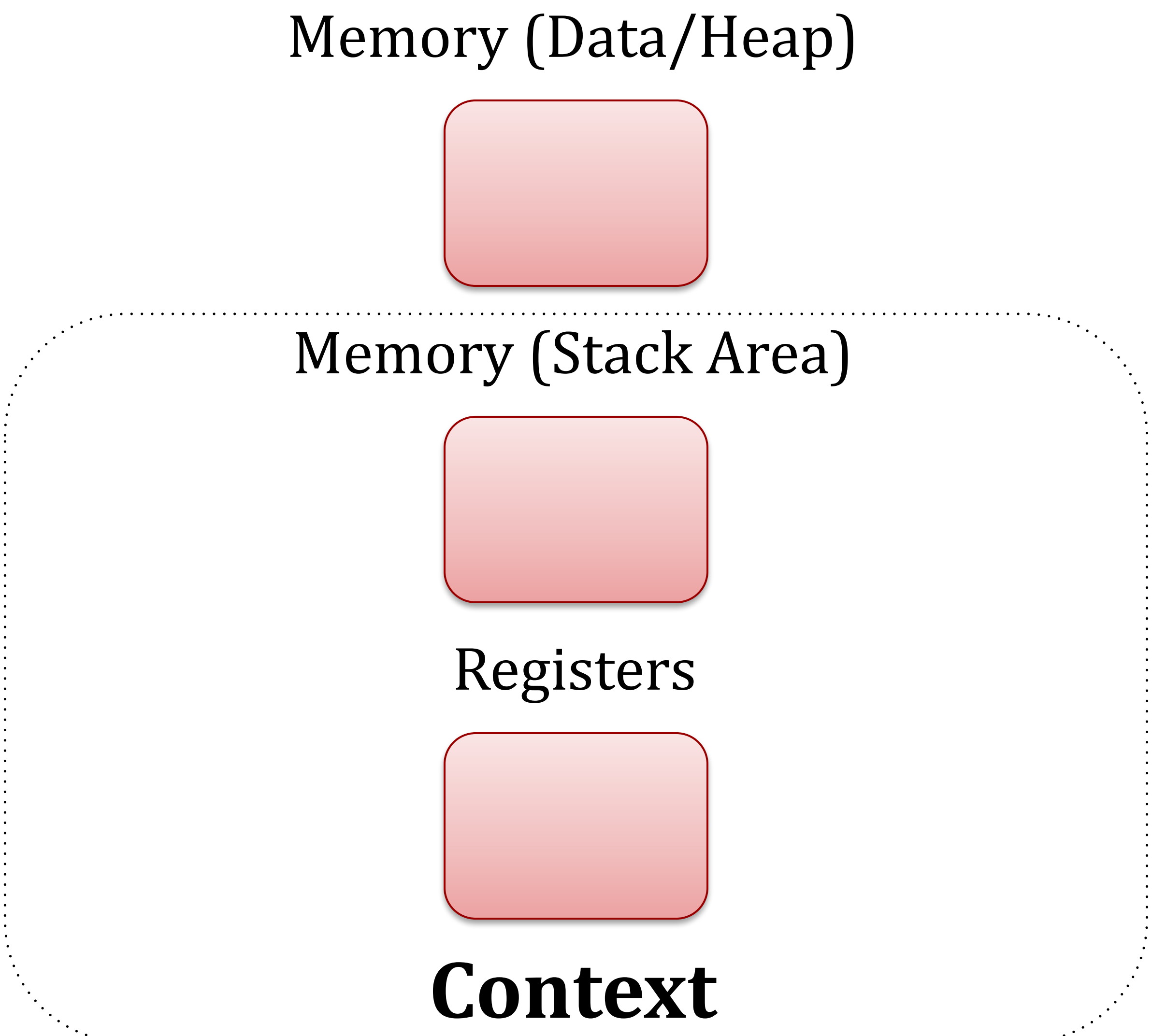
Each thread has its own *context*, consisting of

- Program counter
- Set of registers
- Stack

Really ***important*** for proj2!

Threads (Context)

From Class…



Implementation Requirements

Implement three pthreads functions:

1. `pthread_create`
2. `pthread_exit`
3. `pthread_self`

Implementation Requirements

Implement three pthreads functions:

1. `pthread_create`
2. `pthread_exit`
3. `pthread_self`

Schedule threads

4. Context switch every 50ms in round robin

1.) `phtread_create()`

2.) `phtread_exit()`

What's a good sequence of implementing these 4 components?

a.k.a. Where do I start?

3.) `pthread_self()`

(1 line of code)

4.) `schedule()`

1.) `phtread_create()`

2.) `phtread_exit()`

What does each of these 4 components do?

3.) `pthread_self()`
(Last)

4.) `schedule()`

1) phtread_create()

2) phtread_exit()

3) pthread_self()
(Last)

4) schedule()

1) phtread_create()

Create a new **thread context** for this thread and set it to READY

2) phtread_exit()

3) pthread_self()
(Last)

4) schedule()

1) `phtread_create()` Create a new **thread context** for this thread and set it to `READY`	2) `phtread_exit()` Clean up all resources that were allocated for this thread in `pthread_create()`
3) `pthread_self()` (Last)	4) `schedule()`

1) `phtread_create()`

Create a new **thread context** for this thread and set it to `READY`

2) `phtread_exit()`

Clean up all resources that were allocated for this thread in `pthread_create()`

3) `pthread_self()`

(Last)

4) `schedule()`

Perform a **context switch** from the current thread to the next thread that’s `READY`

1) `phtread_create()`

Create a new **thread context** for this thread and set it to READY

2) `phtread_exit()`

Clean up all resources that were allocated for this thread in `pthread_create()`

3) `pthread_self()`
(Last)

Still: Where do I start?!

4) `schedule()`
Perform a **context switch** from the current thread to the next thread that's READY

Possible Development Strategy

Library alone can’t run (i.e., can’t test either)

Possible Development Strategy

Library alone can’t run (i.e., can’t test either)

Incrementally build the threading library and an example program to test the most recently added functionality during development.

Simple Multi-Threaded Program

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>

#define THREAD_CNT 3

// waste some time
void *count(void *arg) {
  unsigned long int c = \
    (unsigned long int)arg;
  int i;
  for (i = 0; i < c; i++) {
    if ((i % 1000) == 0) {
      printf("id: %x cntd to %d of %ld\n", \
      (unsigned int)pthread_self(), i, c);
    }
  }
  return arg;
}
```

```
int main(int argc, char **argv) {
  pthread_t threads[THREAD_CNT];
  int i;
  unsigned long int cnt = 10000000;

  //create THRAD_CNT threads
  for(i = 0; i<THREAD_CNT; i++) {
    pthread_create(
      &threads[i], NULL, count,
      (void *)((i+1)*cnt));
  }
  //join all threads ... not important for
  //proj2
  for(i = 0; i<THREAD_CNT; i++) {
    pthread_join(threads[i], NULL);
  }
  return 0;
}
```

There is no need to copy this. It is included in the tests directory of your template code, along with suggestions to make it into a fully-automated test.

Execution of a Multi-Threaded Program

Program execution	Implementation Task
1. Program starts	(nothing to do)
2. Launches n threads	`pthread_create()` 1
3. Schedule threads s.t. each thread gets a fair share of CPU time	`schedule()` 2
4. Threads that are complete, exit	`pthread_exit()` 3
5. Program collects results from threads	(nothing to do, for proj2)
6. Program exits	"special" case in `pthread_exit()`

`pthread_create()`

Create new thread context & mark it `READY`

- Thread context (and more) is captured in the thread control block (TCB)
- What is a thread's context?

`pthread_create()`

Create new thread context & mark it `READY`

- Thread context (and more) is captured in the thread control block (TCB)
- What is a thread's context?
  - Registers
  - Stack

`pthread_create()`

Create new thread context & mark it `READY`

- Thread context (and more) is captured in the thread control block (TCB)
- What is a thread's context?
  - Registers
  - Stack
- What else does the TCB need?
  - State (READY, EXITED, RUNNING, etc.)
  - Exit status of the thread (in proj2 it's constant 0)

schedule()

We want to `schedule()` every 50ms

schedule()

We want to `schedule()` every 50ms

- Set an alarm to go off every 50ms
- In the handler do:

schedule()

We want to `schedule()` every 50ms

- Set an alarm to go off every 50ms
- In the handler do:
  1. Preserve context of the currently executing thread
  2. Choose the next thread to run (round robin)
  3. Context switch to the new thread (i.e., restore the new thread's context)

When To Schedule

Must schedule

- a thread exits `pthread_exit()`
- a thread blocks (I/O, semaphore, etc.)

May schedule

- new thread is created `pthread_create()`
- I/O interrupt
- clock interrupt `alarm-handler()`

Define Thread Control Block

Data structure to store info about threads

```
struct thread {
```

Thread id

Information about the state of the thread
(its set of registers)

Information about its stack
(a pointer to thread's stack area)

Information about the status of the thread
(ready, running or exited)

```
};
```

`pthread_create()`

1. Create new TCB
   - Stack
     - Hint: Draw the stack diagram of the empty stack at pthread_create()
   - Registers, in particular
     - PC – Program Counter
     - SP – Stack Pointer
     - How? Remember `jmp_buf` from `setjmp/longjmp`?
2. Once TCB is initialized set state `<- READY`
3. Call schedule()

pthread_create()

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

`thread` <- just the new id

`attr` <- always NULL, i.e., don't care

start_routine … this is the address of where our thread should start execution (i.e., a pointer to the start_routine function, cf. function pointer)

arg … this is the only argument for the new thread (i.e., start_routine)

How to Allocate a Stack

- malloc() will do fine.
- **First entry** in the stack must be address of *pthread_exit*. Why?
  - HW requirement: return from thread calls *pthread_exit*
  - See AMD64 calling convention
- Simply *env*'s RSP = malloc()?
  - No. See stack diagram to right
- Then let RSP = size of stack?
  - No. Remember *pthread_exit*
  - How many bits in an AMD64 address?

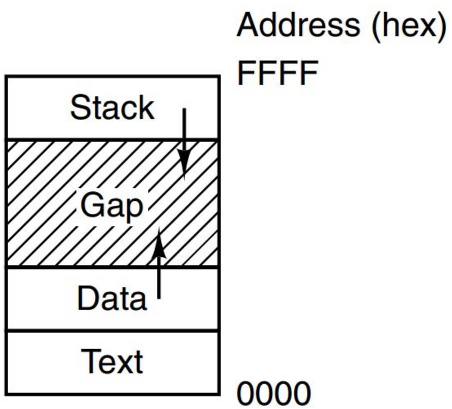
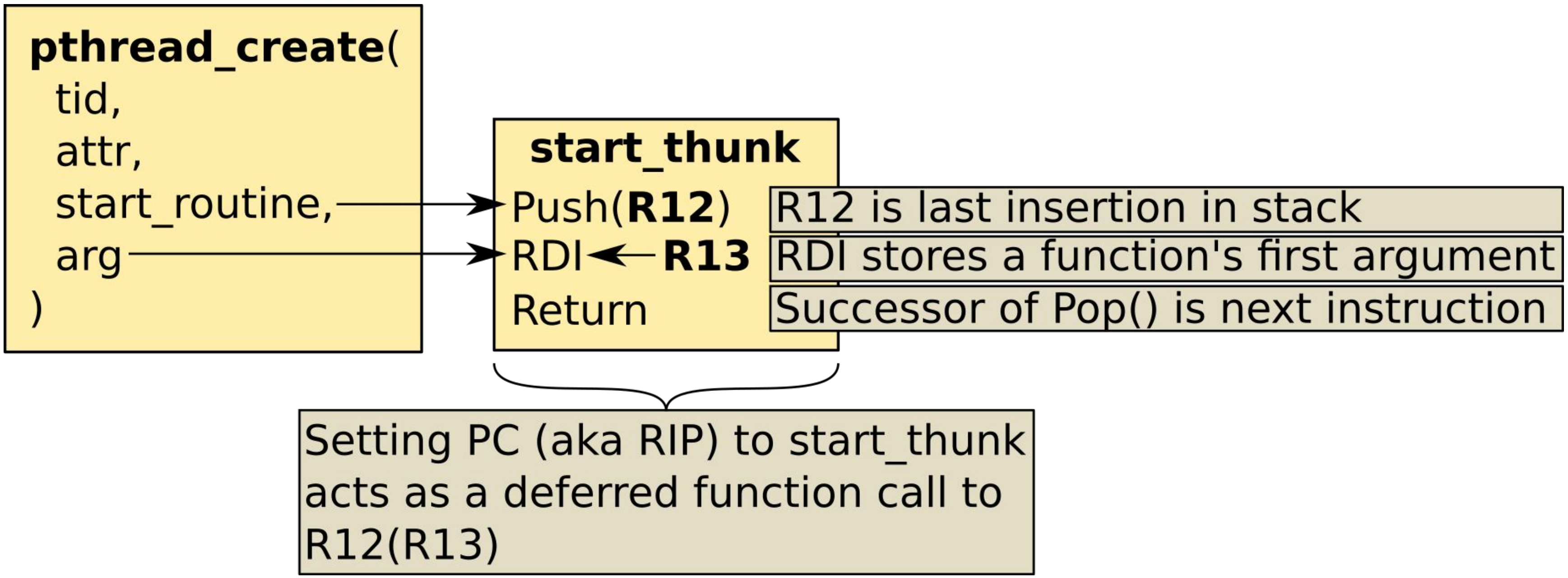


Fig. 1-20 from Modern Operating Systems

How to Launch start_routine

Use *start_thunk* to save a future function call:

```
env->__jmpbuf[JB_PC] = ptr_mangle((unsigned long int)start_thunk);
env->__jmpbuf[JB_R12] = (unsigned long int)start_routine;
env->__jmpbuf[JB_R13] = (unsigned long int)arg;
```



pthread_exit()

1. Free all resources for the current thread.
2. Set the thread's state to EXIT
3. Must automatically be called when start_routine finishes (i.e., returns)! How?

pthread_self()

- Return the thread-id of the currently running thread (at any given time there can only be one thread running)
- The scheduler is the only component that can switch threads
- Thus, the scheduler can maintain a global variable that contains the thread-id of the currently running thread.

```
pthread_t pthread_self(void) {
    return gCurrent;
}
```

Preemptive User Mode Threading Library

Preemptive

- CPU is switched independently of the process behavior
  - A clock interrupt is required

User Mode

- No support from kernel necessary
  - Portable (i.e., works even if kernel does not explicitly support threads)
  - If something goes wrong (e.g., crashes) only your program dies not the entire OS

Threads

- … next slide …

Library

- i.e., only the functions required by the project description
- must not have `main` in your library

How to Compile & Test

Remember to
`#include<pthread.h>`
`#include “ec440threads.h”`
in your sources (or copy contents of `ec440threads.h` into your source)

Running `make` must produce a `threads.o` ELF executable. You can get this via

```
$ gcc -Werror -Wall -g -c -o threads.o threads.c
```

Link this with your test file (e.g., main.c)

```
$ gcc -Werror -Wall -g -o main main.c threads.o
```

Exactly what our `makefile` does, provided you have your implementation in `threads.c` and `main.c`

Things Missing (Incomplete List)

- First time `pthread_create` is called it must:
  - set up all data structures
  - set up the scheduler
  - make a TCB for the main program

Questions?

pthread_create()

```
void *(*start_routine)(void *)
```

Where does this go?

Hint: The thread must start execution there!

`pthread_create()`

```
void *(*start_routine)(void *)
```

Where does this go?

Hint: The thread must start execution there!

Answer: That's the PC for the *new thread*.

pthread_create()

`void *(*start_routine)(void *)`

Where does this go?

Hint: The thread must start execution there!

Answer: That's the PC for the *new thread*.

`void *arg`

Where does this go?

Hint: This is an argument to the `start_routine` function!

Answer: In AMD64 calling convention first six arguments are passed in registers (`RDI`, `RSI`, …). That's where this goes, in `RDI` for the *new thread*.

pthread_create() cont.

While it would be easy to store `start_routine` in `JB_PC` (remember `ptr_mangle`), we cannot easily ensure that `arg` will be the first argument (in `EDI`) when `start_routine` gets "called" (or rather scheduled).

To solve this problem, your implementation of `pthread_create` should store `start_routine` in `R12` and `arg` in `R13` and ensure that the new thread commences as `start_thunk`

`start_thunk` moves the value from `R13` to `RDI` and "jumps" to `R12`, hence faking a call to `start_routine`

Use `ec440threads.h` from piazza

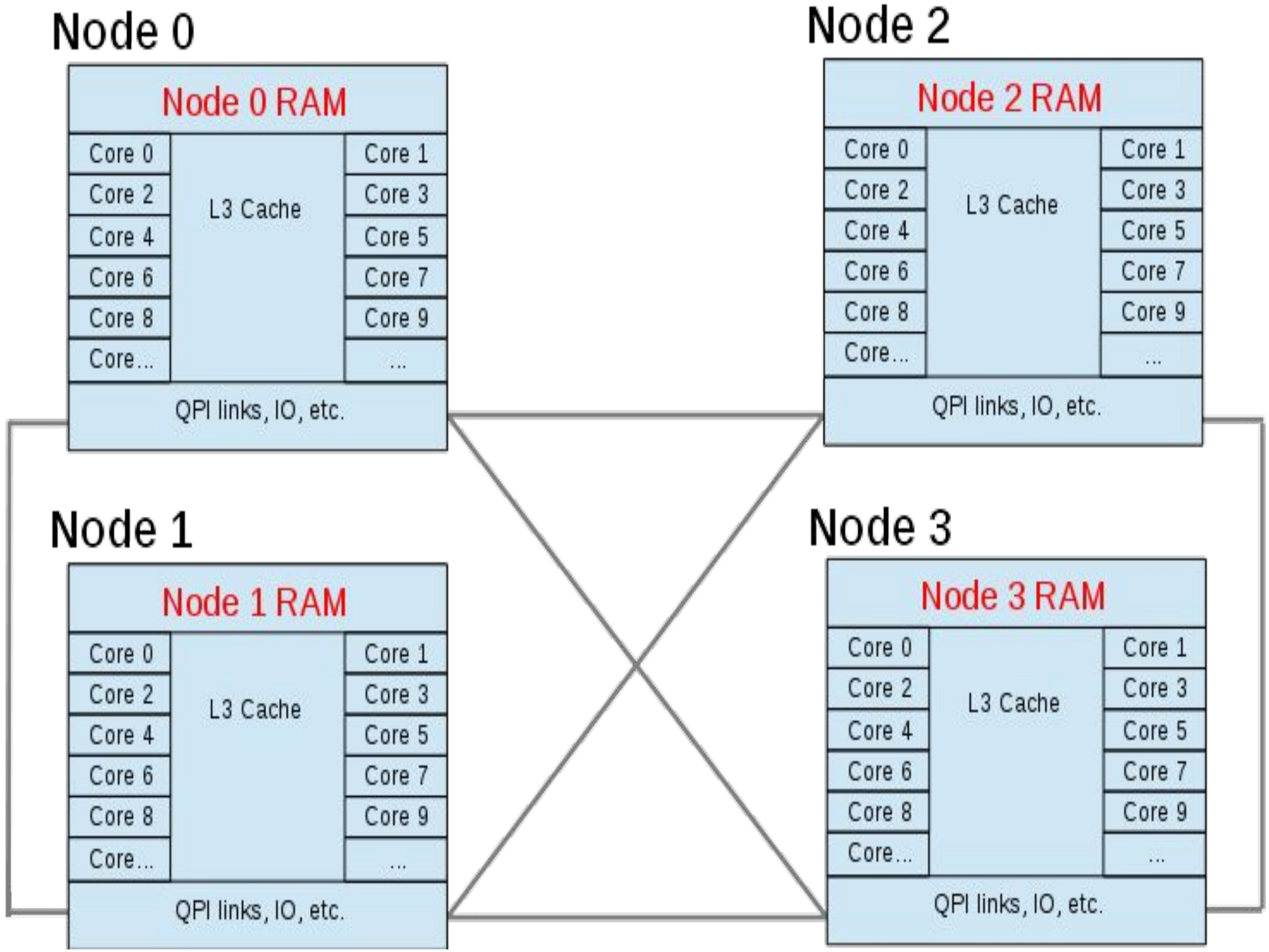
schedule()

We want to `schedule()` every 50ms

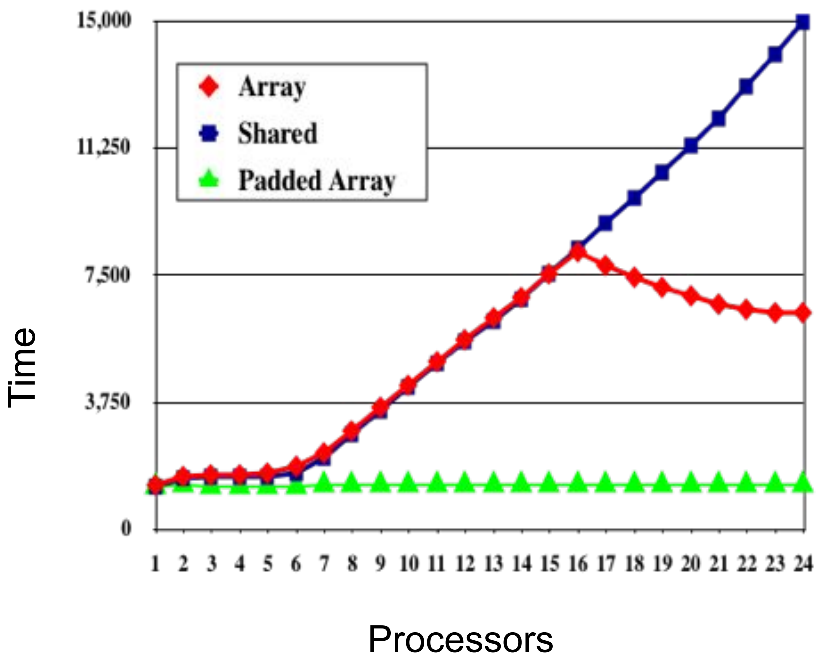
- Set an alarm to go off every 50ms
- In the handler do:
  1. Preserve context of the currently executing thread
     How? Call `setjmp()` & preserve the `jmp_buf`.
     Where?
  2. Choose the next thread **T** to run (round robin)
  3. Context switch to **T** (i.e., restore **T**'s context)
     How? Get `jmp_buf` for **T** and call `longjmp()` with it.

Now back to synchronization...

Typical Four-Node NUMA System



MP performance is hard (bonus information)



- Locks are now implemented as atomic operations on cache lines
- Shared counter example (Tornado 1999)
- Imagine a simple counter.
  - Break apart to multiple counters locked separately
  - Pad into separate cache lines
- These techniques now used pervasively...

False Sharing

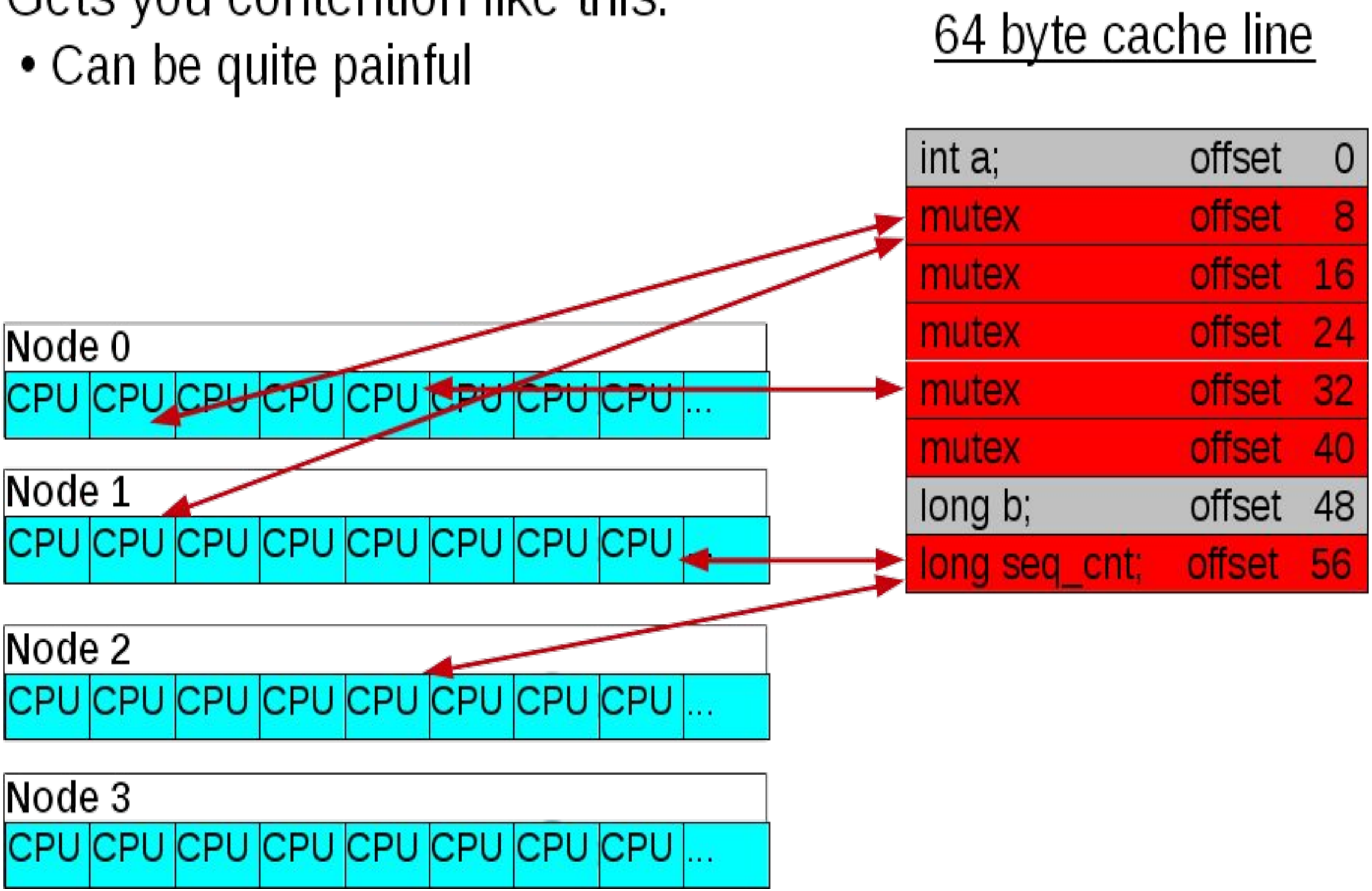
- Different threads sharing common data struct
- Different processes sharing common shared memory.

Ex: Two hotly contended data items sharing a 64-byte cacheline.

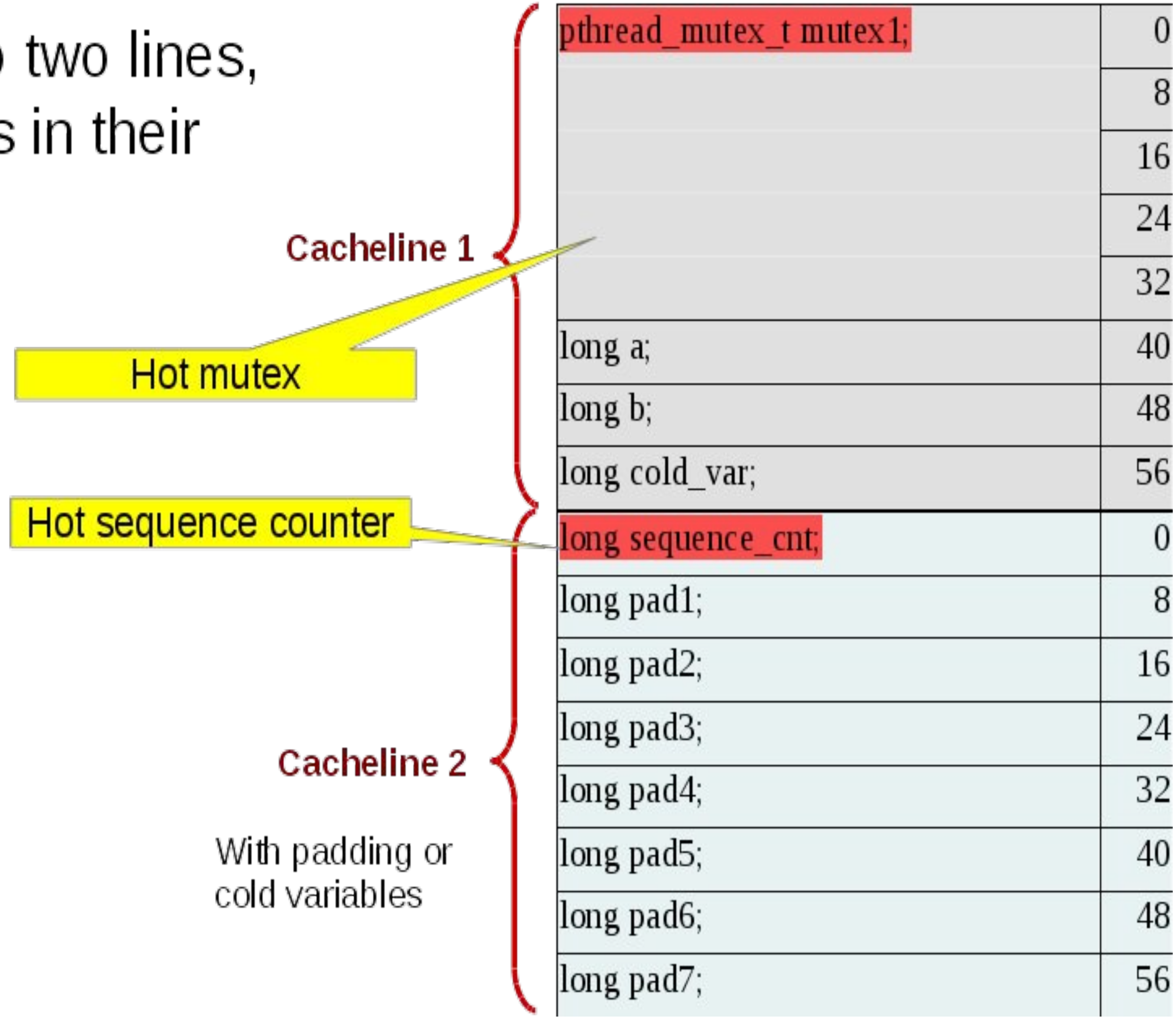
int a;	0	
pthread_mutex_t mutex1;	8	
	16	
	24	*Hot pthread mutex*
	32	
	40	
long b;	48	
long sequence_cnt;	56	*Hot variable*

Gets you contention like this:

- Can be quite painful



Split it up into two lines, with hot items in their own lines:



Future Red Hat update to perf: “c2c data sharing” tool

```
=============================================================================================
Cache                                                                                                    CPU
 #  Refs   Stores   Data Address   Pid   Tid    Inst Address              Symbol    Object  Participants
=============================================================================================
 0  118789  273709        0x602380   37878
     17734  136078        0x602380   37878  37878        0x401520             read_wrt_thread    a.out    0{0}
     13452  137631        0x602388   37878  37883        0x4015a0             read_wrt_thread    a.out    0{1}
     15134       0        0x6023a8   37878  37882        0x4011d7             reader_thread     a.out    1{5}
     14684       0        0x6023b0   37878  37880        0x4011d7             reader_thread     a.out    1{6}
     13864       0        0x6023b8   37878  37881        0x4011d7             reader_thread     a.out    1{7}

 1       31      69   0xffff88023960df40   37878
         13      69   0xffff88023960df70   37878    ***   0xffffffff8109f8e5    update_cfs_rq_blocked_load   vmlinux   0{0,1,2}; 1{14,16}
         17       0   0xffff88023960df60   37878    ***   0xffffffff8109fc2e   __update_entity_load_avg_contrib   vmlinux   0{0,1,2}; 1{14,16}
          1       0   0xffff88023960df78   37878  37882   0xffffffff8109fc4e   __update_entity_load_avg_contrib   vmlinux   0{2}
```

This shows who is contributing to the false sharing:

- The hottest contended cachelines
- The process names, data addr, ip, pids, tids
- The node and CPU numbers they ran on,
- And how the cacheline is being accessed (read or write)
- Disassemble the binary to find the ip, and track back to the sources.

Techniques we use today (bonus information)

- Atomic operations, e.g., increment/decrement
- Per-core/per-socket locks/replication - data on different cache lines
- Fine grained locks embedded in data/objects: cache-miss gets you the data and lock
- Scalable locks: e.g. MCS locks
  - enqueue blocked threads on list
  - each spins on a local variable
  - communication one-to-one
- Read Copy Update (RCU) synchronization (in book)

Key idea RCU

- Let multiple readers access data concurrently with one writer
- Writer copies data, modifies, atomically inserts modification
- Readers see consistent version,
  - either before writer or after writer
- Data not deleted until guaranteed all readers done

Avoiding Locks: Read-Copy-Update (1)

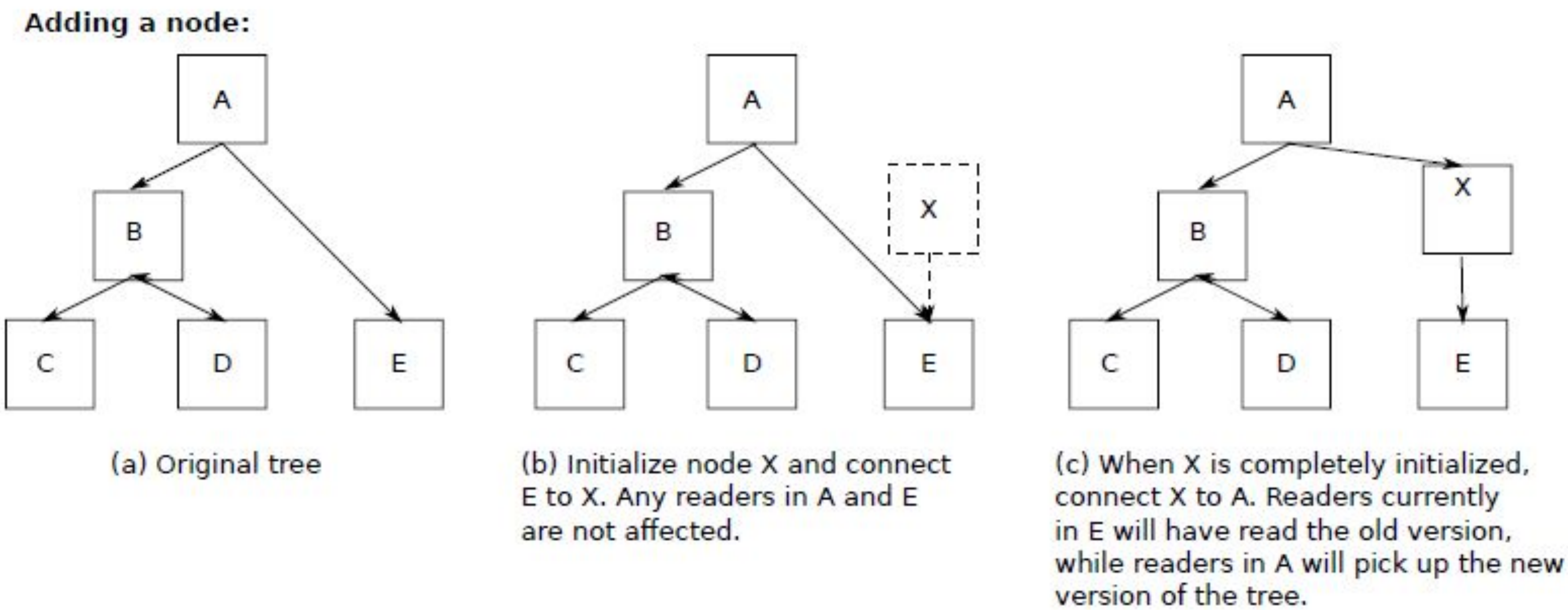
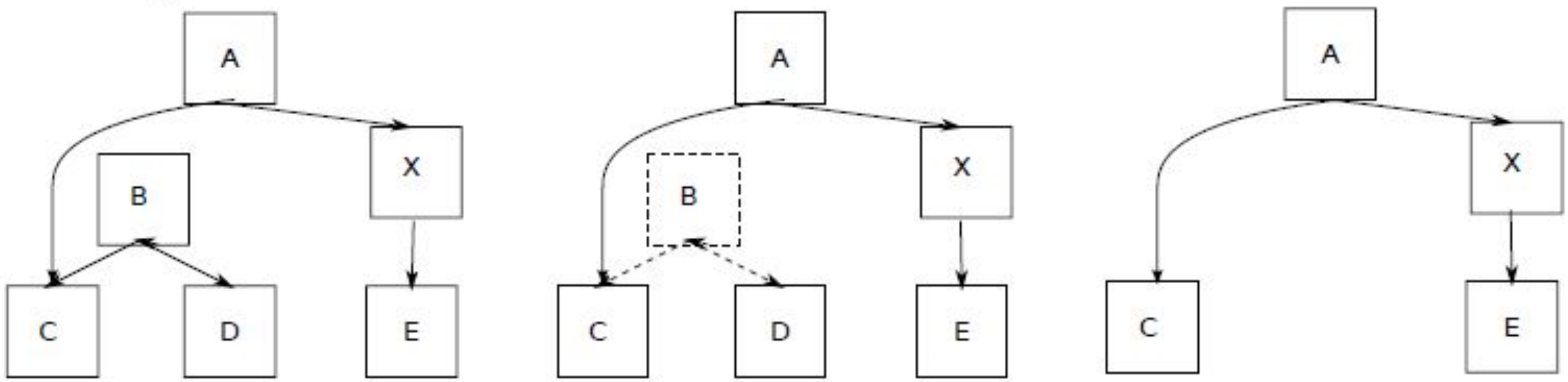


Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Avoiding Locks: Read-Copy-Update (2)

Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.

(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed by anymore.

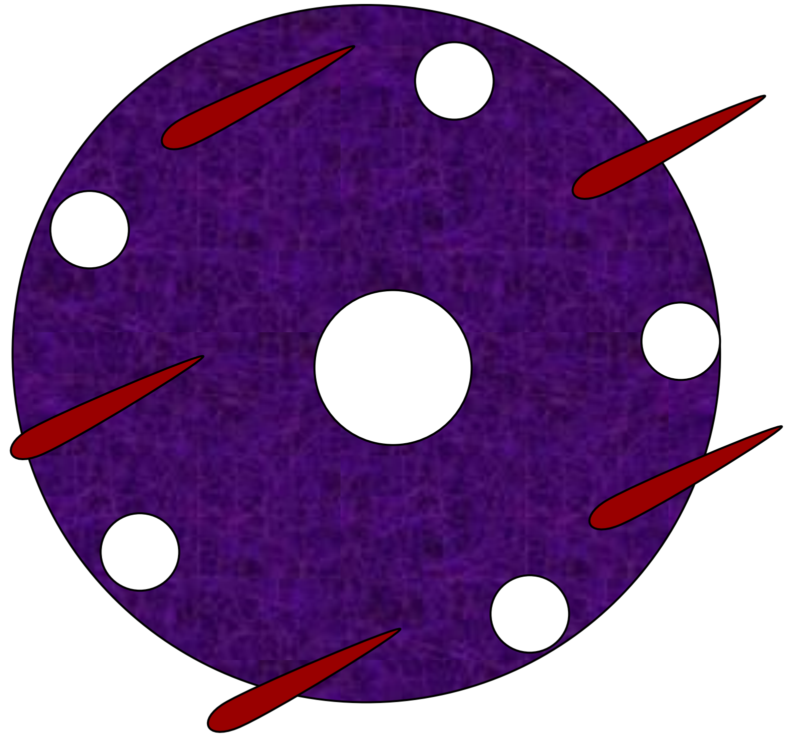
(f) Now we can safely remove B and D

Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

A bit of history

- Introduced concurrently in
  - Sequent's large MP ptx system
  - K42/Tornado systems Toronto/IBM
- We presented Ottawa Linux Symposium 2001
- Immediately started getting wide use in Linux
- Forced Paul to make it his PhD
- Subject of $3B SCO lawsuit in 2003
- oops
- Paul finally could release his PhD in 2004

Dining Philosophers Problem



First Solution

```
philosopher(int i) {
 while (1) {
    think();
    take_chopstick(i);
    take_chopstick((i + 1) % N);
    eat();
    put_chopstick(i);
    put_chopstick((i + 1) % N);
  }
}
```

If all the philosopher take their left chopsticks they get stuck

Second Solution

```
philosopher(int i) {
 while (1) {
    think();
    take_chopstick(i);
    if (!available((i + 1) % N))
   {
      put_chopstick(i);
      continue();
   }
    take_chopstick((i + 1) % N);
    eat();
    put_chopstick(i);
    put_chopstick((i + 1) % N);
  }
}
```

It is possible that all the philosophers put down and pick up their chopsticks at the same time, leading to starvation

`think()` should be randomized

Third Solution

Use one mutex

- Do a `down()` when acquiring chopsticks
- Do an `up()` when releasing chopsticks

Problem:

Only one philosopher can eat at once

Fourth Solution

- Maintain state of philosophers
  - Switch to `HUNGRY` when ready to eat
  - Sleep if no chopsticks available
  - When finished wake up your neighbors
- Use one semaphore for each philosopher, to be used to suspend in case no chopsticks are available
- Use one mutex for critical regions
- Use `take_chopsticks/put_chopsticks` to acquire both chopsticks

Fourth Solution

```
philosopher(i) {
  think();
  take_chopsticks(i);
  eat();
  put_chopsticks(i);
}
```

```
take_chopsticks(i) {
  mutex.down();
  state[i] = HUNGRY;
  test(i);
  mutex.up();
  philosopher[i].down();
}
```

```
put_chopsticks(i) {
  mutex.down();
  state[i] = THINKING;
  test((i + 1) % N);
  test((i + N - 1) % N);
  mutex.up();
}
```

```
test(i) {
  if (state[i] == HUNGRY && state[(i + 1) % N] != EATING &&
                            state[(i + N - 1) % N] != EATING) {
        state[i] = EATING;
        philosopher[i].up();
  }
}
```