barkhausen institut

Microkernel Construction

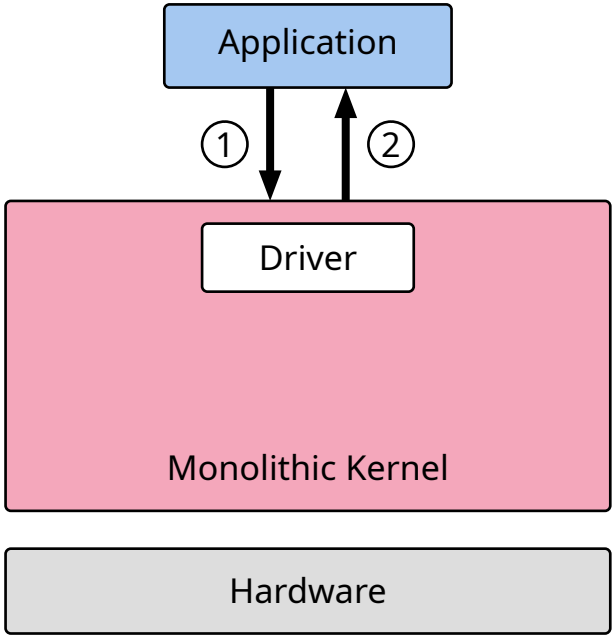
Inter-Process Communication

Nils Asmussen

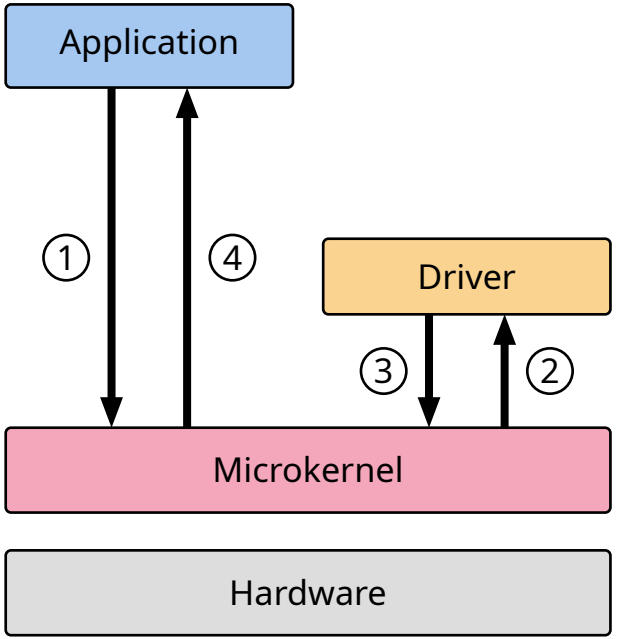
05/21/2026

- **Introduction**
  - **Synchronous vs. Asynchronous**
  - Implementation Variants
- Synchronous IPC in NOVA
- Asynchronous IPC in NOVA
- Userspace API

Performance vs. Robustness

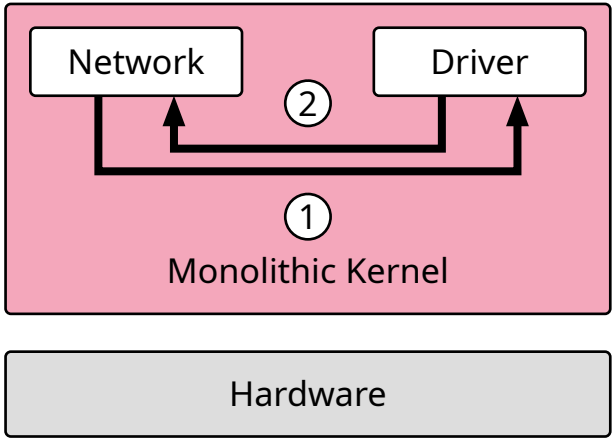


2 kernel entries/exits

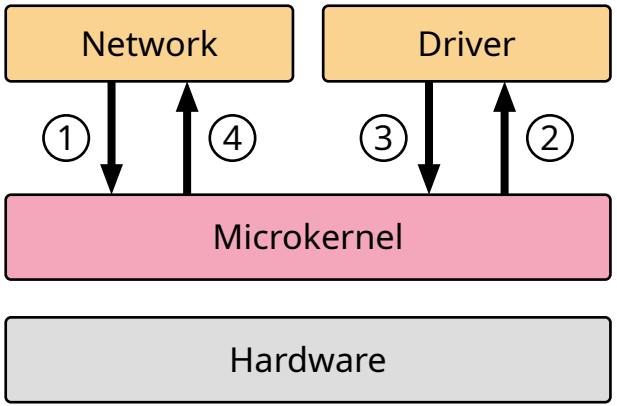


4 kernel entries/exits
2 context switches

Performance vs. Robustness



2 function calls/returns



4 kernel entries/exits
2 context switches

Synchronous vs. Asynchronous

Synchronous

- Sender is blocked until receiver is ready
- Data and control transfer directly from sender to receiver

Asynchronous

- Data is transferred to temporary location
- Sender continues execution
- If receiver arrives, the data is transferred to him

Synchronous vs. Asynchronous – Comparison

- Synchronous is typically simpler and faster
  - no buffering simplifies kernel implementation
  - usage of synchronous IPC also easier
- Synchronous is less prone to DoS attacks
  - no buffer memory required

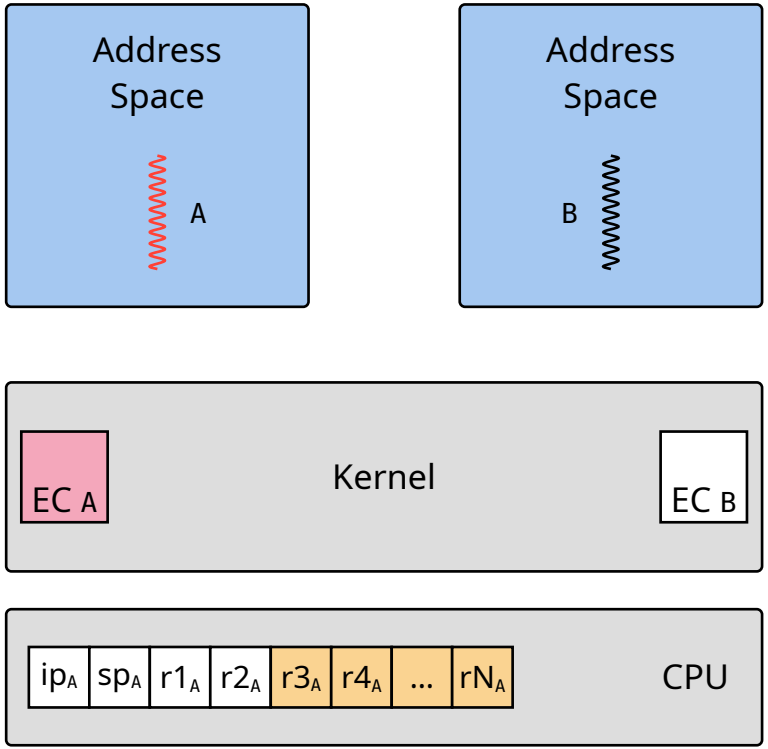
Synchronous vs. Asynchronous – Comparison

- Synchronous is typically simpler and faster
  - no buffering simplifies kernel implementation
  - usage of synchronous IPC also easier
- Synchronous is less prone to DoS attacks
  - no buffer memory required

- Asynchronous is typically more flexible
  - send/receive buffer can be chosen by application
  - larger IPC messages possible
- Asynchronous allows to do other work instead of waiting
  - larger overall throughput
  - async programming model can be difficult without language support

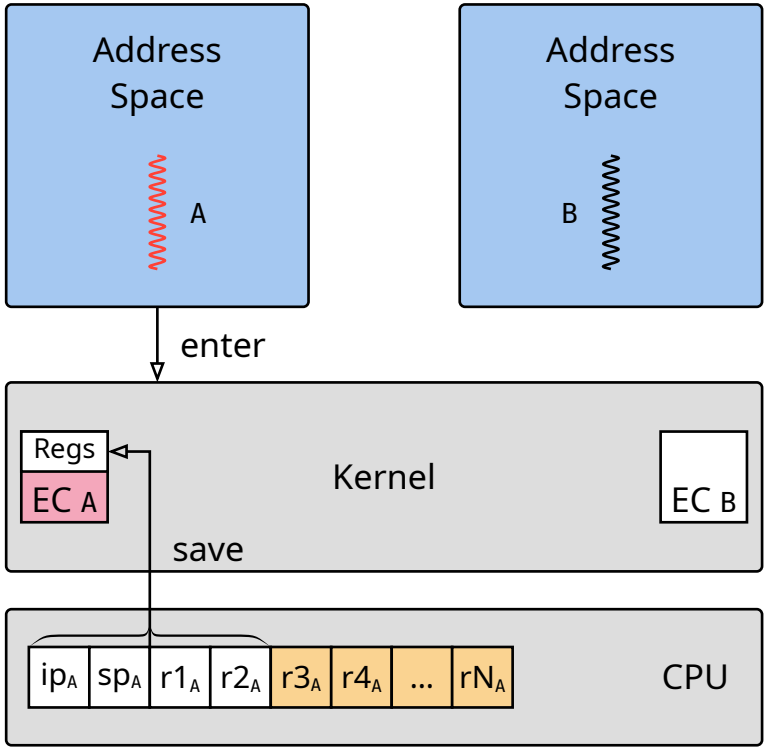
- **Introduction**
  - Synchronous vs. Asynchronous
  - **Implementation Variants**
- Synchronous IPC in NOVA
- Asynchronous IPC in NOVA
- Userspace API

Register IPC



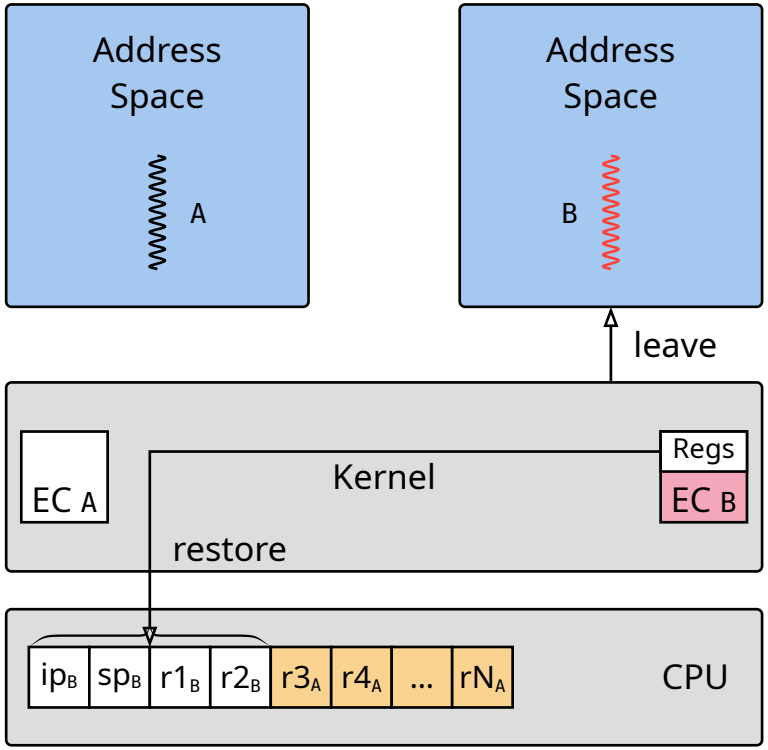
- User thread `A` is running
- Writes payload to registers `r3` .. `rN`

Register IPC



- User thread `A` is running
- Writes payload to registers `r3` .. `rN`
- Performs system call
- Kernel saves registers except `r3` .. `rN`

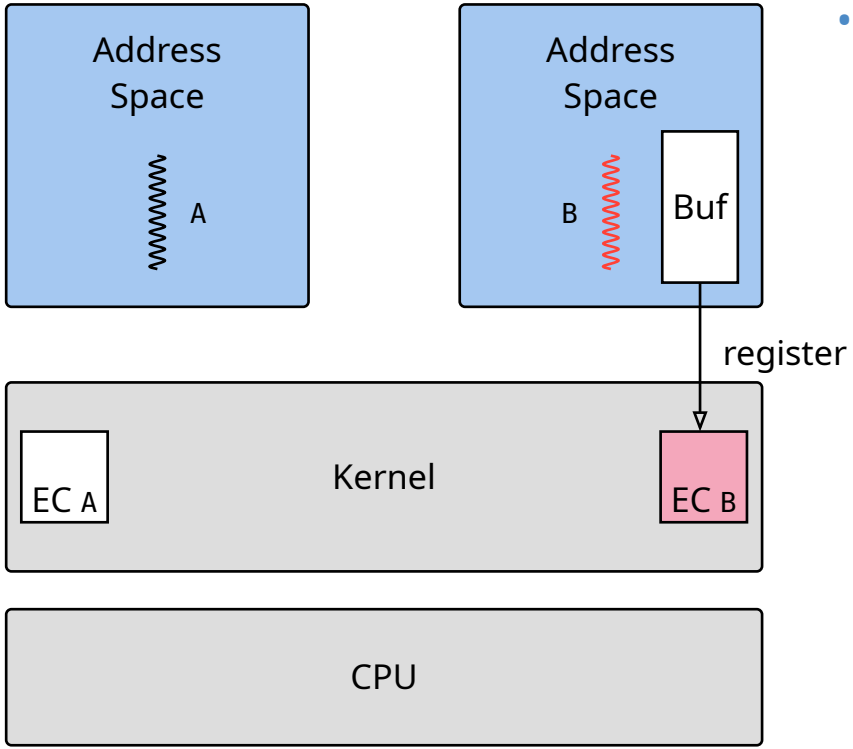
Register IPC



- User thread `A` is running
- Writes payload to registers `r3` .. `rN`
- Performs system call
- Kernel saves registers except `r3` .. `rN`

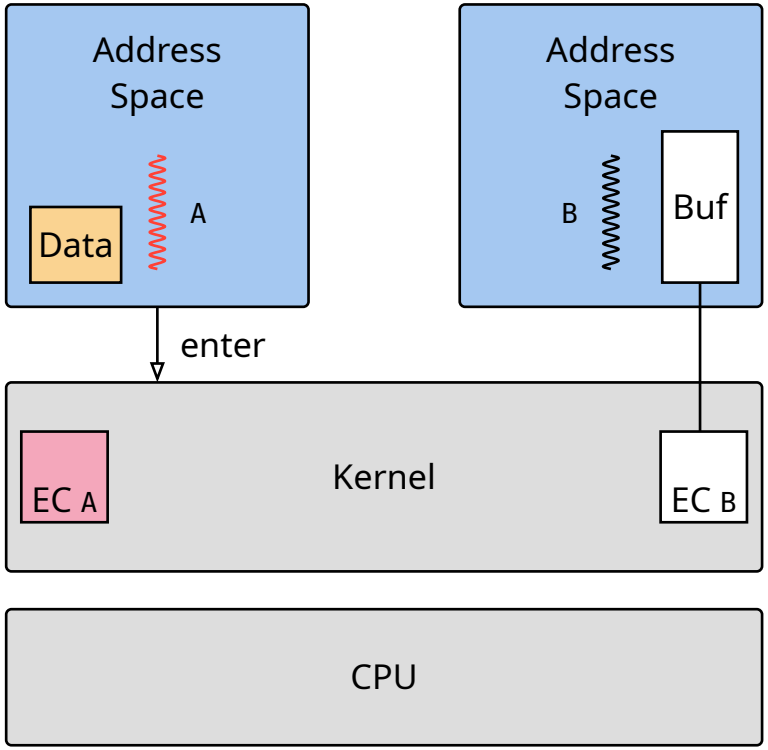
- Kernel switches to thread `B`
- Kernel restores regs except `r3` .. `rN`
- Resumes user thread `B`

User-Memory IPC



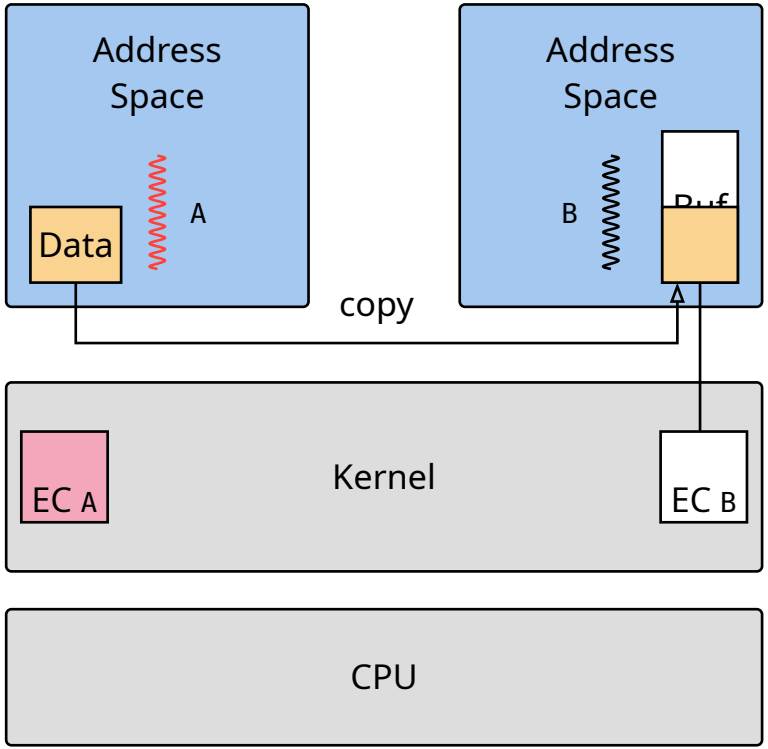
- Ahead of time: thread `B` registers `Buf`

User-Memory IPC



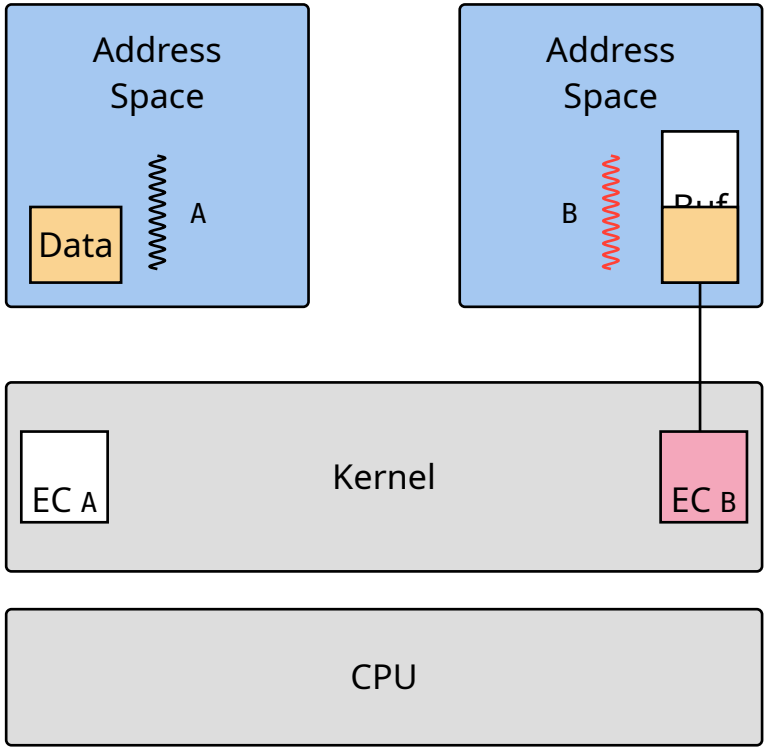
- Ahead of time: thread `B` registers `Buf`
- User thread `A` is running
- Writes payload to `Data` region
- Performs system call

User-Memory IPC



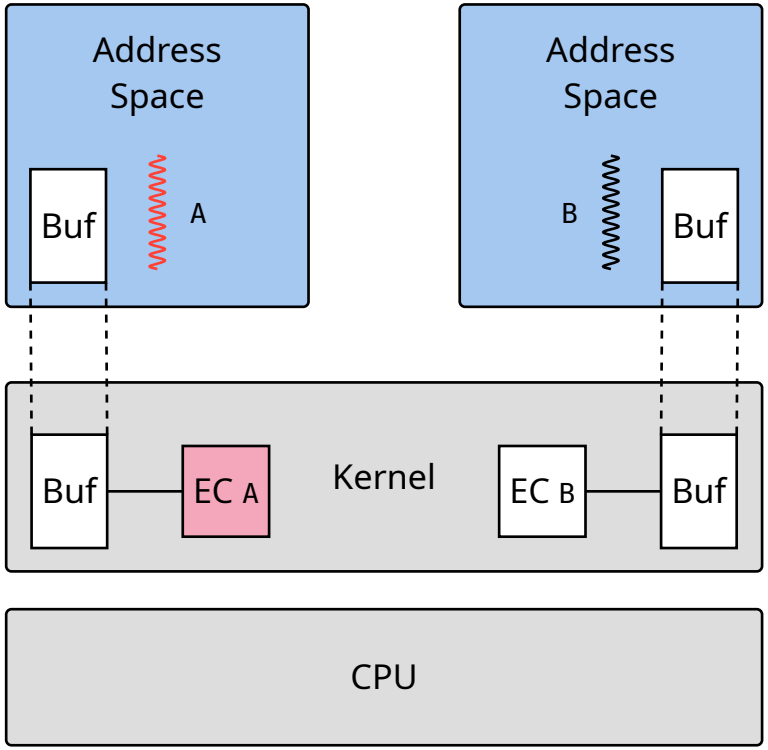
- Ahead of time: thread `B` registers `Buf`
- User thread `A` is running
- Writes payload to `Data` region
- Performs system call
- Kernel copies from `Data` to `Buf`

User-Memory IPC



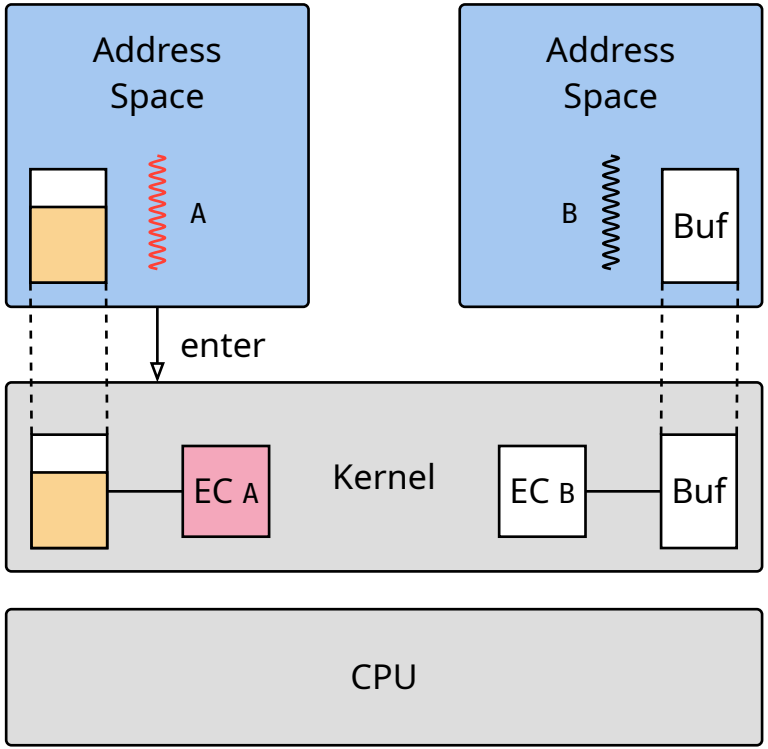
- Ahead of time: thread `B` registers `Buf`
- User thread `A` is running
- Writes payload to `Data` region
- Performs system call
- Kernel copies from `Data` to `Buf`
- Kernel switches to user thread `B`

Kernel-Memory IPC



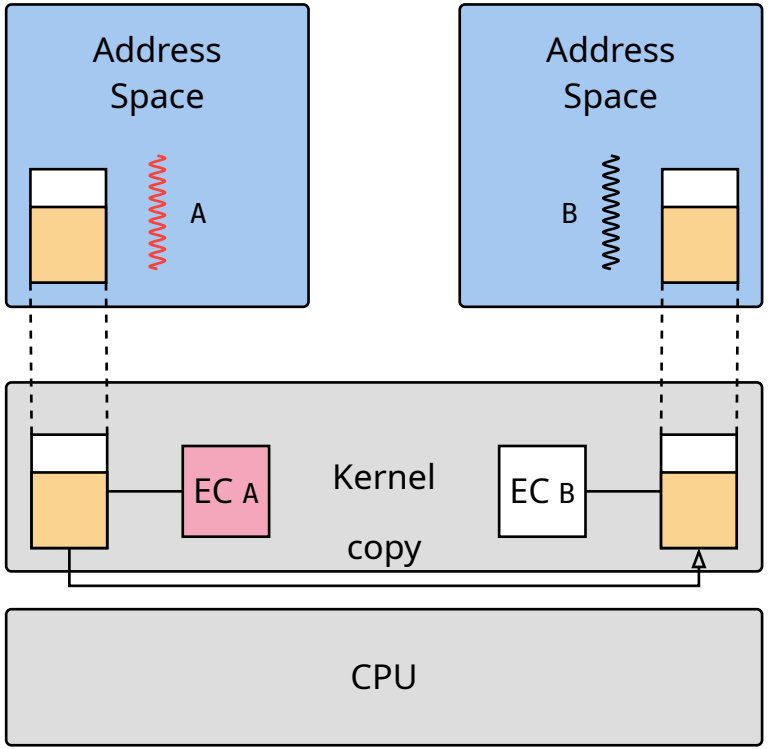
- User thread A is running

Kernel-Memory IPC



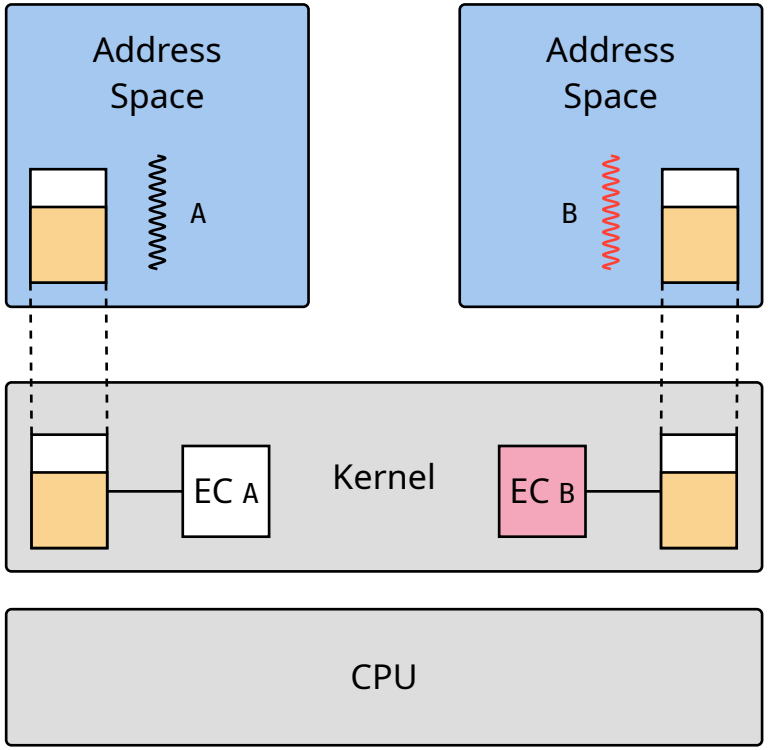
- User thread `A` is running
- Writes payload to `Buf`
- Performs system call

Kernel-Memory IPC



- User thread `A` is running
- Writes payload to `Buf`
- Performs system call
- Kernel copies between `Bufs`

Kernel-Memory IPC



- User thread `A` is running
- Writes payload to `Buf`
- Performs system call
- Kernel copies between `Buf`s
- Kernel switches to user thread `B`

Comparison

IPC Type	🙂 Pros	😕 Cons
Register IPC	• Very fast • No pagefaults	• Limited to registers • Specific data location

Comparison

IPC Type	🙂 Pros	🙁 Cons
Register IPC	• Very fast • No pagefaults	• Limited to registers • Specific data location
User-Memory IPC	• Limited data size • Arbitrary data location	• Pagefaults can occur • Slower (no direct copy)

Comparison

IPC Type	🙂 Pros	😕 Cons
Register IPC	• Very fast • No pagefaults	• Limited to registers • Specific data location
User-Memory IPC	• Limited data size • Arbitrary data location	• Pagefaults can occur • Slower (no direct copy)
Kernel-Memory IPC	• Fast • No pagefaults	• Limited data size • Specific data location

- Introduction
- **Synchronous IPC in NOVA**
  - **Synchronous IPC in General**
  - Exception IPC
  - Cross-Core IPC
- Asynchronous IPC in NOVA
- Userspace API

Introduction to IPC in NOVA

- NOVA uses synchronous kernel memory IPC to
  - Exchange data
  - Exchange capabilities
- Asynchronous IPC by semaphores for
  - Signaling
  - Deliver interrupts to user space

Introduction to IPC in NOVA

- NOVA uses synchronous kernel memory IPC to
  - Exchange data
  - Exchange capabilities
- Asynchronous IPC by semaphores for
  - Signaling
  - Deliver interrupts to user space
- Synchronous IPC is core-local
- Asynchronous IPC can be used cross-core

Synchronous IPC

- Uses kernel-memory IPC
- Message buffer is called User Thread Control Block (UTCB)
- Each EC has exactly one UTCB

Synchronous IPC

- Uses kernel-memory IPC
- Message buffer is called User Thread Control Block (UTCB)
- Each EC has exactly one UTCB

- A UTCB is one page, i.e., 4 KiB large
- All UTCBs are mapped in kernel part of each address space

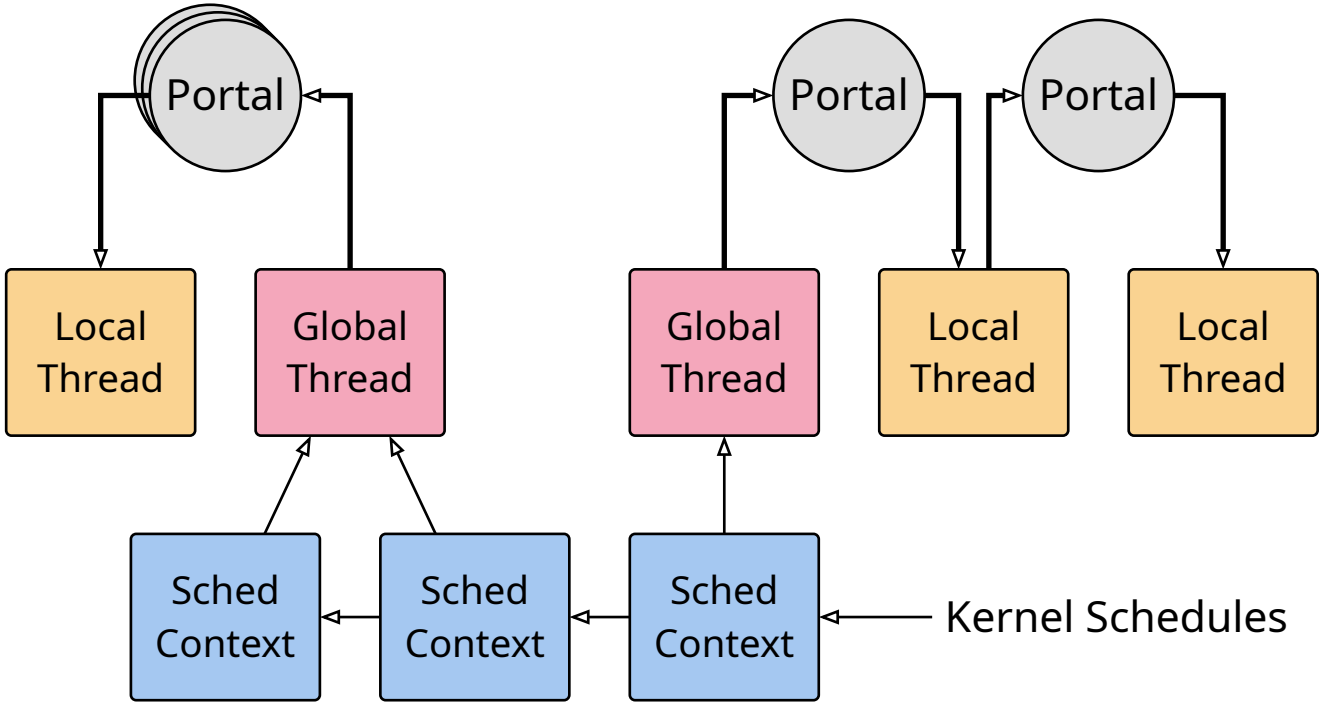
Synchronous IPC

- Uses kernel-memory IPC
- Message buffer is called User Thread Control Block (UTCB)
- Each EC has exactly one UTCB

- A UTCB is one page, i.e., 4 KiB large
- All UTCBs are mapped in kernel part of each address space

- On EC creation, UTCB is allocated and mapped into user space
- UTCBs are pinned → no pagefaults

Reminder: Threads



Portal

Properties

- Local Thread, that handles the portal
- Instruction Pointer (address of portal function)
- Id, delivered to the portal (parameter of portal function)

Portal

Properties

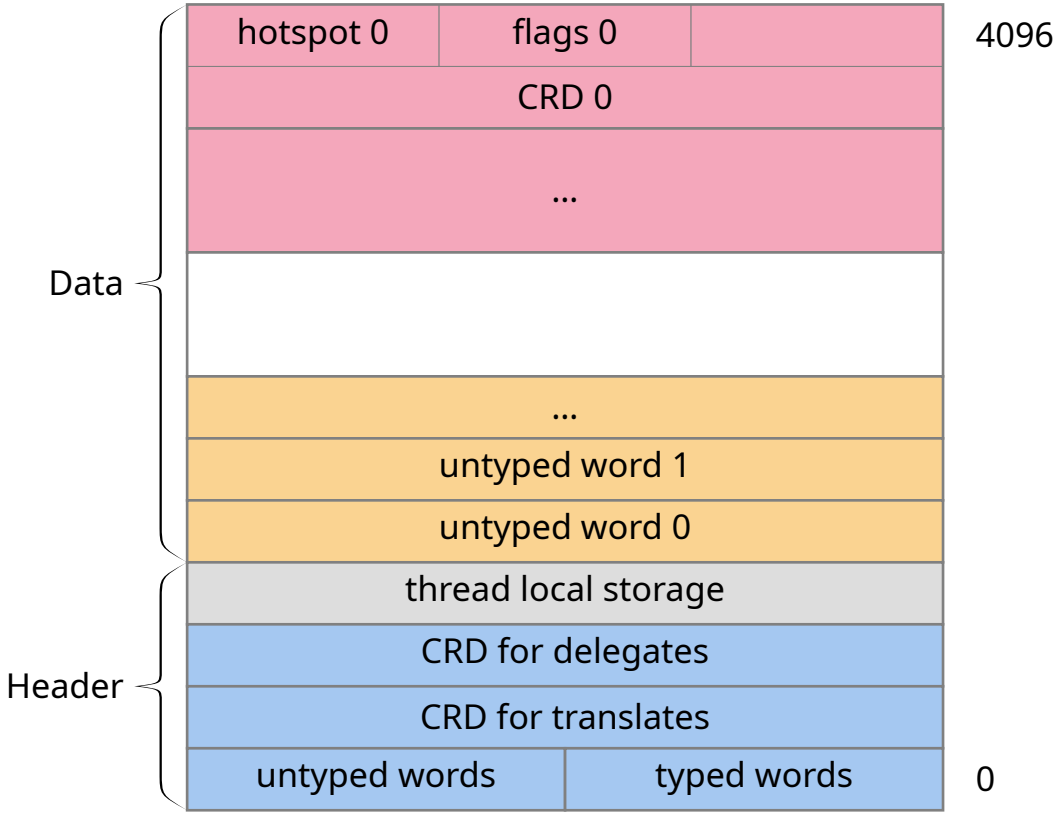
- Local Thread, that handles the portal
- Instruction Pointer (address of portal function)
- Id, delivered to the portal (parameter of portal function)

Code example from NRE

C++

```
1  PORTAL static void portal_echo(void *id) { ... }
2
3  int main() {
4      Reference<LocalThread> lt = LocalThread::create();
5      Pt echo(lt, portal_echo);
6      echo.set_id(0x1234);
7      echo.call();
8  }
```

UTCB Layout


hotspot 0
flags 0
4096
CRD 0
...
Data
...
untyped word 1
untyped word 0
thread local storage
CRD for delegates
Header
CRD for translates
untyped words
typed words
0

Syscall: Call Portal (1)

C++

```
1  Sys_call *s = static_cast<Sys_call*>(current->sys_regs());
2  Kobject *obj = Space_obj::lookup(s->pt()).obj();
3  Pt *pt = static_cast<Pt*>(obj);
4  Ec *ec = pt->ec;
5  if(EXPECT_FALSE(current->cpu != ec->xcpu))
6      sys_finish<Sys_regs::BAD_CPU>();
7  if(EXPECT_TRUE(!ec->cont)) {
8      current->cont = ret_user_sysexit;
9      current->set_partner(ec);     // sets Ec::rcap
10     ec->cont = recv_user;
11     ec->regs.set_pt(pt->id);
12     ec->regs.set_ip(pt->ip);
13     ec->make_current();
14 }
15
16 ec->help(sys_call);
```

Syscall: Call Portal (2)

C++

```
1   void Ec::recv_user() {
2       Ec *ec = current->rcap;
3       ec->utcb->save(current->utcb);
4       if(EXPECT_FALSE(ec->utcb->tcnt()))
5           delegate<true>();
6       ret_user_sysexit();
7   }
8   void Ec::help(void (*c)()) {
9       current->cont = c;
10      if(EXPECT_TRUE(++Sc::ctr_loop < 100)) {
11          Ec *ec = this;
12          while(ec->partner)
13              ec = ec->partner;
14          ec->make_current();
15      }
16      die("Livelock"); }
```

- Introduction
- **Synchronous IPC in NOVA**
  - Synchronous IPC in General
  - **Exception IPC**
  - Cross-Core IPC
- Asynchronous IPC in NOVA
- Userspace API

Exception IPC

- The kernel should have no policy
- Userland should decide what to do in case of an exception
- In particular, memory management is done in userland
- Each EC has an exception portal selector offset
- At this offset, portals are expected for all exceptions

Exception IPC: Details

C++

```
1  void Ec::handle_exc(Exc_regs *r) {
2      switch(r->vec) {
3          case Cpu::EXC_NM:
4              handle_exc_nm();
5              return;
6
7          case Cpu::EXC_PF:
8              if(handle_exc_pf(r))
9                  return;
10             break;
11         ...
12     }
13
14     send_msg<ret_user_iret>();
15 }
```

Exception IPC: Details

C++

```
1  template<void (*C)()>
2  void Ec::send_msg() {
3      Exc_regs *r = &current->regs;
4      Kobject *obj = Space_obj::lookup(current->evt + r->dst_portal).obj();
5      Pt *pt = static_cast<Pt*>(obj);
6      Ec *ec = pt->ec;
7
8      if(EXPECT_TRUE(!ec->cont)) { ec->cont = recv_kern; /* ... */ }
9      ec->help(send_msg<C>);
10 }
11
12 void Ec::recv_kern() {
13     Ec *ec = current->rcap;
14     current->utcb->load_exc(&ec->regs);
15     ret_user_sysexit();
16 }
```

- Introduction
- **Synchronous IPC in NOVA**
  - Synchronous IPC in General
  - Exception IPC
  - **Cross-Core IPC**
- Asynchronous IPC in NOVA
- Userspace API

Cross-Core IPC

- More complicated
  - Interruption of another core
  - Synchronization between cores (when is the other core done?)

Cross-Core IPC

- More complicated
  - Interruption of another core
  - Synchronization between cores (when is the other core done?)
- Significantly slower
  - Inter-Processor Interrupts (IPIs) are very expensive
  - Interruption of OoO cores is particularly expensive

Cross-Core IPC

- More complicated
  - Interruption of another core
  - Synchronization between cores (when is the other core done?)
- Significantly slower
  - Inter-Processor Interrupts (IPIs) are very expensive
  - Interruption of OoO cores is particularly expensive
- Timeslice Donation?
- Not supported in upstream NOVA (patches exist)

Multicore Servers on L4Re/NOVA

NOVA

- Local threads for portal handling with time-slice donation and helping
- Needs one portal and local thread per CPU core
- Client threads call the portal on their CPU core
- Cross-core data exchange via shared memory and semaphores

Multicore Servers on L4Re/NOVA

NOVA

- Local threads for portal handling with time-slice donation and helping
- Needs one portal and local thread per CPU core
- Client threads call the portal on their CPU core
- Cross-core data exchange via shared memory and semaphores

L4Re

- Has also timeslice donation and helping on one core (but “SC” not exposed)
- One receiving thread per IPC endpoint
- Can receive messages from all cores (but cross-core is slow)
- Can use per-client receive threads with (almost) empty time budget

- Introduction
- Synchronous IPC in NOVA
- **Asynchronous IPC in NOVA**
  - **Asynchronous IPC in General**
  - Interrupts
- Userspace API

Semaphores

- A semaphore is a kernel object
- Properties:
  - Counter
  - Queue of ECs
- Operations (via syscall):
  - Down
  - Down to zero
  - Up

Semaphores: Usecases

- Synchronization with shared memory (e.g., multithreading)
  - Typically combined with atomic operations
  - Atomic operations in case of no contention
  - System call in case of contention
- Signaling (e.g., producer-consumer scenarios)
- Delivery of interrupts to userspace

- Introduction
- Synchronous IPC in NOVA
- **Asynchronous IPC in NOVA**
  - Asynchronous IPC in General
  - **Interrupts**
- Userspace API

Interrupt Semaphores

- Object cap space of root PD has semaphore per interrupt
- Can be delegated to device drivers, …
- Is up’ed by the kernel on IRQ

Interrupt Semaphores

- Object cap space of root PD has semaphore per interrupt
- Can be delegated to device drivers, …
- Is up’ed by the kernel on IRQ

Usage example: Keyboard driver in NRE

C++

```
1 static void kbhandler(void*) {
2     Gsi gsi(KEYBOARD_IRQ);
3     while(1) {
4         gsi.down();
5         Keyboard::Packet data;
6         if(hostkb->read(data))
7             broadcast(kbsrv, data);
8     }
9 }
```

Semaphore Operations – Down/Zero

```
1   void Sm::dn(bool zero) {                                    C++
2       Ec *e = Ec::current;
3       {
4           Lock_guard<Spinlock> guard(lock);
5           if(counter) {
6               counter = zero ? 0 : counter - 1;
7               return;
8           }
9           enqueue(e);
10      }
11      e->block_sc();
12  }
```

Semaphore Operations – Up

C++

```
1   void Sm::up() {
2       Ec *e;
3       {
4           Lock_guard<Spinlock> guard(lock);
5           if(!(e = dequeue())) {
6               counter++;
7               return;
8           }
9       }
10      e->release();
11  }
```

- Introduction
- Synchronous IPC in NOVA
- Asynchronous IPC in NOVA
- **Userspace API**
  - **UTCB Frames**
  - IPC with C++ Shift Operators

Many Approaches

- Plain C API
- C++ shift operators to get/put values from/into UTCB
- C++ templates generate server and client stubs
- IDL compiler
- ...

NOVA Runtime Environment (NRE)

Uses C++ shift operators:

- 🙂 No external tool required
- 🙂 No separate language to learn
- 🙂 Rather simple to implement
- 🙂 Much simpler to use than C implementations
- 😕 Need to implement stub functions manually, if desired
- 😕 Need to keep client and server consistent (types, order, …)

NOVA Runtime Environment (NRE)

Uses C++ shift operators:

🙂 No external tool required

🙂 No separate language to learn

🙂 Rather simple to implement

🙂 Much simpler to use than C implementations

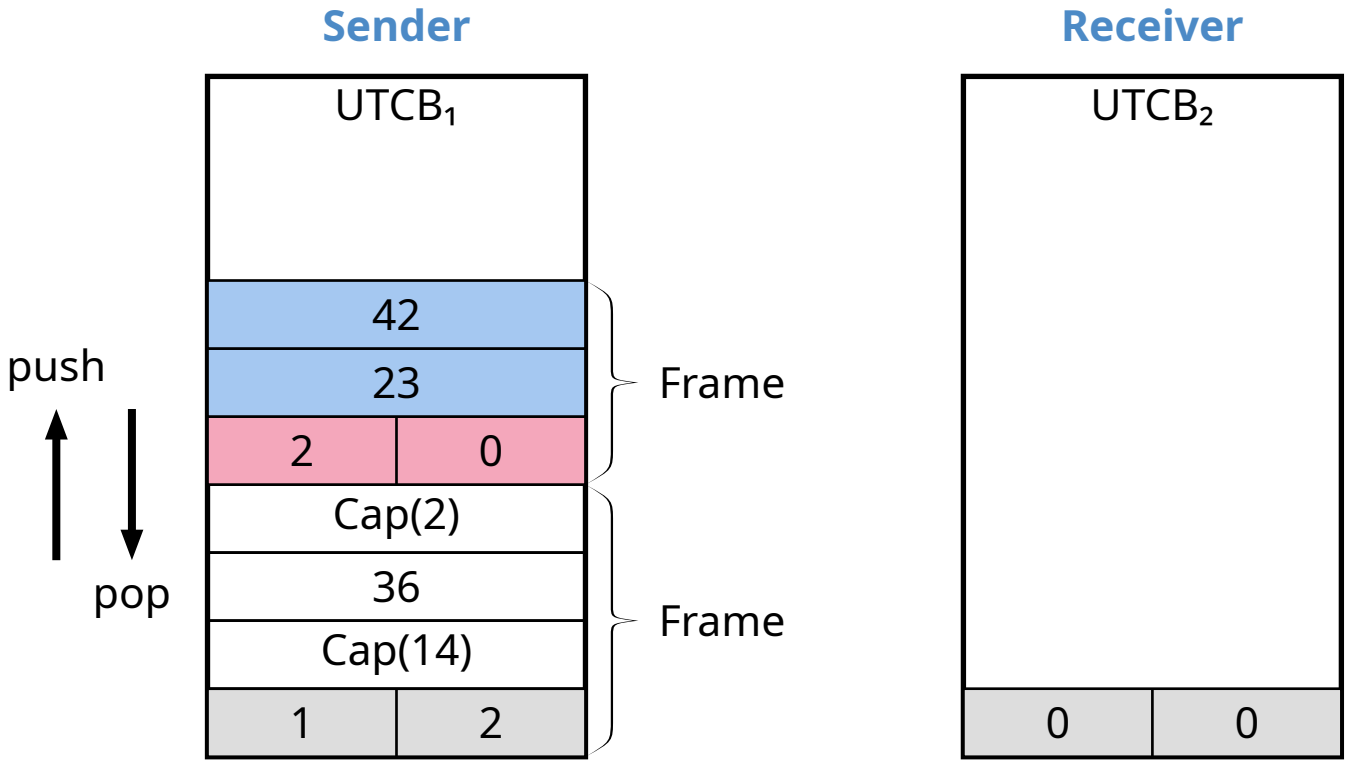
🙁 Need to implement stub functions manually, if desired

🙁 Need to keep client and server consistent (types, order, …)

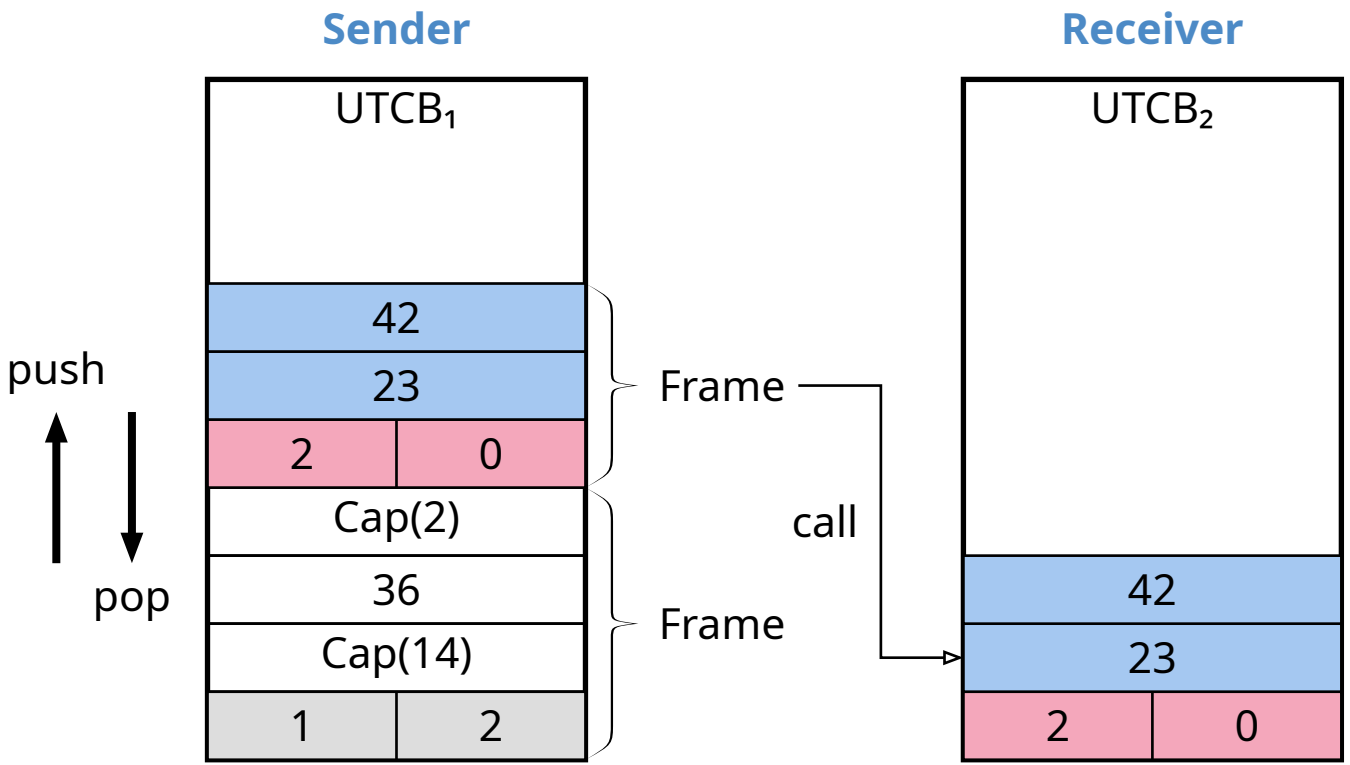
Supports multiple frames within one UTCB:

- Allows nested usages of the UTCB
- Important for calling library functions

NRE UTCB Frames



NRE UTCB Frames



- Introduction
- Synchronous IPC in NOVA
- Asynchronous IPC in NOVA
- **Userspace API**
  - UTCB Frames
  - **IPC with C++ Shift Operators**

Usage Example

Client

C++

```
1 UtcbFrame uf;
2 uf << 1 << String("foo");
3 portal.call(uf);
4 int res;
5 uf >> res;
```

Usage Example

Client

C++

```
1 UtcbFrame uf;
2 uf << 1 << String("foo");
3 portal.call(uf);
4 int res;
5 uf >> res;
```

Server

C++

```
1 PORTAL static void myportal(void*) {
2     UtcbFrameRef uf;
3     int i; String s;
4     uf >> i >> s;
5     // handle the request
6     uf << 0;
7 }
```

Implementation

Push Value

```cpp
template<typename T> UtcbFrameRef & operator<<(const T& value) {
    const size_t words = (sizeof(T) + sizeof(word_t) - 1) / sizeof(word_t);
    *reinterpret_cast<T*>(_utcb->msg + untyped() * sizeof(word_t)) = value;
    _utcb->untyped += words;
    return *this;
}
```

Implementation

Push Value

```C++
template<typename T> UtcbFrameRef & operator<<(const T& value) {
    const size_t words = (sizeof(T) + sizeof(word_t) - 1) / sizeof(word_t);
    *reinterpret_cast<T*>(_utcb->msg + untyped() * sizeof(word_t)) = value;
    _utcb->untyped += words;
    return *this;
}
```

Pop Value

```C++
template<typename T> UtcbFrameRef & operator>>(T &value) {
    const size_t words = (sizeof(T) + sizeof(word_t) - 1) / sizeof(word_t);
    value = *reinterpret_cast<T*>(_utcb->msg + _upos * sizeof(word_t));
    _upos += words;
    return *this;
}
```