Microkernel Construction
Interprocess Communication

Nils Asmussen

05/03/2018
Introduction
- Microkernel vs. Monolithic kernel
- Synchronous vs. Asynchronous
- Different Implementations

Synchronous IPC in NOVA
Asynchronous IPC in NOVA
Userspace API
**Microkernel vs. Monolithic: Syscalls**

- Monolithic kernel: 2 kernel entries/exits
- Microkernel: 4 kernel entries/exits + 2 context switches
Microkernel vs. Monolithic: Calls Between Services

- Monolithic kernel: 2 function calls/returns
- Microkernel: 4 kernel entries/exits + 2 context switches

Diagram showing the comparison between Monolithic and Microkernel.
Synchronous vs. Asynchronous

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

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

**Comparison**

- Synchronous is typically simpler and faster (no buffering)
- Synchronous is less prone to DoS attacks (buffer memory)
- Asynchronous is typically more flexible
- Asynchronous allows to do other work while waiting
Register IPC

Sender

User Stack_A

(1) save

Regs

EC_A

(2) copy

Kernel

(3) restore

EC_B

Receiver

User Stack_B

sp

ip

r1

r2

r3

r4

...

CPU
User Memory IPC

Sender

Data

(2) send

(3) copy

Receiver

Buffer

(1) register

EC

Kernel

bufobj

CPU

TODO: don’t register receive buffer. otherwise, the point “no copy to special location first” is bogus.
Kernel Memory IPC

(1) send
(2) copy

Kernel
CPU
Sender Receiver
Data
Buf
EC
Kernel
EC
Buf
CPU
Comparison

- **Register IPC**
  - Very fast
  - Amount of data limited to CPU registers

- **User Memory IPC**
  - Amount of data not limited
  - No copy to special location first
  - Pagefaults can occur
  - Slower (no direct copy)

- **Kernel Memory IPC**
  - Fast
  - No pagefaults
  - Amount of data limited
  - Copy to special location first
Introduction

Synchronous IPC in NOVA
  - Synchronous IPC in General
  - Exception IPC

Asynchronous IPC in NOVA

Userspace API
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
- A UTCB is one page, i.e., 4 KiB large
- All UTCBs are mapped in kernel space
- On EC creation, a UTCB is allocated and mapped to a specified address in user space
- UTCBs are pinned → no pagefaults
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
PORTAL static void portal_echo(void *id) {
}

int main() {
    Reference<LocalThread> lt = LocalThread::create();
    Pt echo(lt, portal_echo);
    echo.set_id(0x1234);
    echo.call();
}
```
Timeslice Donation and Helping

- **Timeslice donation:**
  - \( EC_1 \) calls portal with \( SC_L \)
  - \( SC_L \) is donated to \( EC_3 \)

- **Priority inversion:**
  - \( SC_H \) is blocked by \( SC_L \)
    - (indirectly with \( SC_M \))

- **Helping:**
  - If \( SC_L \) has no time left,\n    \( SC_H \) helps \( EC_3 \)
  - i.e., \( EC_3 \) runs with \( SC_H \)
Timeslice Donation and Helping

- **Timeslice donation:**
  - $EC_1$ calls portal with $SC_L$
  - $SC_L$ is donated to $EC_3$

- **Priority inversion:**
  - $SC_H$ is blocked by $SC_L$
    (indirectly with $SC_M$)

- **Helping:**
  - If $SC_L$ has no time left, $SC_H$ helps $EC_3$
  - I.e., $EC_3$ runs with $SC_H$
Timeslice Donation and Helping

Timeslice donation:
- $EC_1$ calls portal with $SC_L$
- $SC_L$ is donated to $EC_3$

Priority inversion:
- $SC_H$ is blocked by $SC_L$ (indirectly with $SC_M$)

Helping:
- If $SC_L$ has no time left, $SC_H$ helps $EC_3$
- I.e., $EC_3$ runs with $SC_H$
Timeslice Donation and Helping

- **Timeslice donation:**
  - $EC_1$ calls portal with $SC_L$
  - $SC_L$ is donated to $EC_3$

- **Priority inversion:**
  - $SC_H$ is blocked by $SC_L$
    (indirectly with $SC_M$)

- **Helping:**
  - If $SC_L$ has no time left,
    $SC_H$ helps $EC_3$
  - I.e., $EC_3$ runs with $SC_H$
Timeslice Donation and Helping

- **Timeslice donation:**
  - $EC_1$ calls portal with $SC_L$
  - $SC_L$ is donated to $EC_3$

- **Priority inversion:**
  - $SC_H$ is blocked by $SC_L$
    (indirectly with $SC_M$)

- **Helping:**
  - If $SC_L$ has no time left, $SC_H$ helps $EC_3$
  - I.e., $EC_3$ runs with $SC_H$
Syscall: Call Portal

Sys_call *s = static_cast<Sys_call*>(current->sys_regs());
Kobject *obj = Space_obj::lookup(s->pt()).obj();
Pt *pt = static_cast<Pt*>(obj);
Ec *ec = pt->ec;

if (EXPECT_FALSE(current->cpu != ec->xcpu))
    sys_finish<Sys_regs::BAD_CPU>();

if (EXPECT_TRUE(!ec->cont)) {
    current->cont = ret_user_sysexit;
    current->set_partner(ec);    // sets Ec::rcap
    ec->cont = recv_user;
    ec->regs.set_pt(pt->id);
    ec->regs.set_ip(pt->ip);
    ec->make_current();
}

ec->help(sys_call);
void Ec::recv_user() {
    Ec *ec = current->rcap;
    ec->utcb->save (current->utcb);
    if (EXPECT_FALSE (ec->utcb->tcnt()))
        delegate<true>();
    ret_user_sysexit();
}

void Ec::help (void (*c)()) {
    current->cont = c;
    if (EXPECT_TRUE (++Sc::ctr_loop < 100)) {
        Ec *ec = this;
        while(ec->partner)
            ec = ec->partner;
        ec->make_current();
    }
    die ("Liveloop");
}
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
void Ec::handle_exc (Exc_regs *r) {
    switch (r->vec) {
        case Cpu::EXC_NM:
            handle_exc_nm();
            return;
        case Cpu::EXC_PF:
            if (handle_exc_pf (r))
                return;
            break;
        ...
    }
    send_msg<ret_user_iret>();
}
template <void (*C)()>
void Ec::send_msg() {
    Exc_regs *r = &current->regs;
    Kobject *obj = Space_obj::lookup (current->evt + r->dst_portal).obj();
    Pt *pt = static_cast<Pt *>(obj);
    Ec *ec = pt->ec;
    if (EXPECT_TRUE (!ec->cont)) {
        ec->cont = recv_kern;
        ...}
    ec->help (send_msg<C>);
}

void Ec::recv_kern() {
    Ec *ec = current->rcap;
    current->utcb->load_exc (&ec->regs);
    exc (ec->regs);
    ret_user_sysexit();
}
Outline

- Introduction
- Synchronous IPC in NOVA
- Asynchronous IPC in NOVA
  - Synchronization
  - Interrupts
- Userspace API
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
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
static void kbhandler(void*) {
    Gsi gsi(KEYBOARD_IRQ);
    while(1) {
        gsi.down();

        Keyboard::Packet data;
        if(hostkb->read(data))
            broadcast(kbsrv, data);
    }
}
```
Semaphore Operations

```c++
void Sm::dn (bool zero) {
    Ec *e = Ec::current;
    { Lock_guard <Spinlock> guard (lock);
        if (counter) {
            counter = zero ? 0 : counter - 1;
            return;
        }
        enqueue (e);
    }
    e->block_sc();
}

void Sm::up() {
    Ec *e;
    { Lock_guard <Spinlock> guard (lock);
        if (!(e = dequeue())) { counter++; return; }
    }
    e->release();
}
```
• 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 UT CB
- 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, . . .)

Supports multiple frames within one UTCB:

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

Sender

EC₁

UTCB₁

Frame

Frame

pop

push

2 0

1 0

Frame

Frame

Receiver

EC₂

UTCB₂

call()
Usage Example

Client

UtcbFrame uf;
uf << 1 << String("foo");
portal.call(uf);
int res;
uf >> res;

Server

PORTAL static void myportal(void*) {
    UtcbFrameRef uf;
    int i; String s;
    uf >> i >> s;
    // handle the request
    uf << 0;
}
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;
}

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;
}
May 17th: Exercise 1 (Kernel entry, exit, syscalls) in room APB E046