Synchronous IPC with Formal Specification and Proof

Hengchu Zhang, Advised by Prof. Shao

This report presents my experience in developing formal specification and proof of a synchronous interprocess communication extension to the CertikOS project.

Motivation

Software is used in many mission critical situations that can influence life and death. For example, flight control software of commercial airlines, software that controls throttle power and brakes in cars and software that controls medical equipment are all in the mission critical category. We cannot allow malfunctions during the execution of these mission critical softwares. Current methods of developing these software take a very long time and a lot of resources. Furthermore, since most modern software run on an operating system, it is also critical that these mission critical software can have a trusted operating system – a base that is formally specified and proven to behave within the specification, otherwise, the operating system itself could malfunction and disaster can still happen.

CertikOS is an on-going project by the Prof. Shao’s Flint group here at Yale. The project develop an OS kernel that is formally specified and proven in the language Coq. For my CS490 project, I extended CertikOS’s original IPC facility.
Original IPC and New IPC

The original IPC interface in CertikOS was very simple – user processes can send/recv one 32-bit integer at a time asynchronously. Old messages are not logged, and if process A tries to send to process B in quick succession without proper synchronization on the user side, messages can be overwritten thus lost.

To relieve users of the synchronization responsibility, one of the extensions I made to the original IPC is adding synchronicity. Users can now call send/recv without worrying about if they’re operating in lock-step.

The original IPC only sends one integer at a time. The amount of data transferred is very small compared to the system call overhead – user has to use the int operation to trap into the kernel. Doing this every time when a process needs to send an integer is wasteful. Thus, another extension to the original IPC is the ability to send and receive an entire buffer of integers at a time. This is a rather complicated issue because a buffer is passed into the kernel via a user-side pointer, and there are many cases where things can go wrong with such a pointer. I would discuss the problems encountered during the development in the following sections.

Design of the New IPC

CertikOS currently statically allocates 64 thread control blocks, representing 64 possible user-processes. To keep track of the interprocess communication among the processes, we create an array of 64 structs to record information about the communication:

```c
struct SyncIPCChan {
    unsigned int to;
    unsigned int paddr;
    unsigned int count;
};

struct SyncIPCChan ipc-chan-pool[64];
```
The `to` field contains the process id of the intended receiver process. The `count` field is the length of the buffer.

The `paddr` field contains the physical address of the sender’s buffer. It's important we understand the implicataions of having a physical address of the sender’s buffer – if the buffer crosses the page boundary, we won’t be able traverse the page table again and retrieve starting physical address of the buffer on the next page. Why would we do this? How would we handle page boundary issues then?

The answer is that we require the buffer to be page-aligned, and we limit the maximum size of the buffer to be exactly 1 page – 1024 integers. We enforced these limits is primarily for proof-simplification, but it’s quite easy to extend the currently implementation to remove these constraints.

Then what exactly happens when a user process tries to send something? I’ll list out the important functions related in order of invocation. The kernel code that eventually runs to start a send is the following:

```c
#define MAX_BUFFSIZE 1024
#define NUM_CHAN 64
extern unsigned int get_curid(void);
extern unsigned int get_state(unsigned int pidx);
extern void set_sync_chan_paddr(unsigned int chanid, unsigned int paddr);
extern void set_sync_chan_to(unsigned int chanid, unsigned int to);
extern void set_sync_chan_count(unsigned int chanid, unsigned int count);
extern unsigned int get_sync_chan_to(unsigned int chanid);
extern unsigned int get_kernel_pa(unsigned int pid, unsigned int vaddr);

unsigned int syncsendto_chan_pre(unsigned int pid, unsigned int vaddr, unsigned int scount) {
    unsigned int target_state = get_state(pid);
```
if (target_state == 3) {
    return MAX_BUFFSIZE+2;
} else {
    if (0 <= pid && pid < NUM_CHAN) {
        unsigned int myid = get_curid();
        unsigned int sender_kpa = get_kernel_pa(myid, vaddr);

        if (scount < MAX_BUFFSIZE) {
            set_sync_chan_paddr(myid, sender_kpa);
            set_sync_chan_count(myid, scount);
        } else {
            set_sync_chan_paddr(myid, sender_kpa);
            set_sync_chan_count(myid, MAX_BUFFSIZE);
        }

        if (get_sync_chan_to(myid) == NUM_CHAN) {
            set_sync_chan_to(myid, pid);
        } else {
            // Should Crash
        }

        return myid;
    } else {
        return MAX_BUFFSIZE+1;
    }
}

Note the MAX_BUFFSIZE we defined at the top of the file. Since we know that a user process cannot send more than 1 page of data across, we can use values larger than MAX_BUFFSIZE as error codes.

This function does the following thing: it first checks if the receiver is a dead process by reading its thread control block state (value 3 means DEAD). If receiver is not dead, then it proceeds to set the corresponding fields in the SyncIPCChan struct we defined above, then it returns the sender’s process id.

The caller of this function lives in the system call level of CertikOS, which is the function that get’s called when the user performs an int operation. As the name of syncsendto_chan_pre
suggests, there is a second part to the send operation, which is called `syncsendto_chan_post`. In between calls to `pre` and `post`, we invoke `thread_sleep(sid)` where `sid` is the sender’s process id. This would put the sender process to sleep, thus the call to `syncsendto_chan_post` would be blocked until the sender is woken up.

In fact, it is the receiver’s responsibility to wake up the sender. Once the receive code finishes reading and copying values from the sender’s buffer, it would call `thread_wakeup` on the sending processes id. The kernel code that handles receive is the following:

```c
#define NUM_CHAN 64
#define TCB_STATE_DEAD 3
#define MAX_BUFFSIZE 1024

extern unsigned int get_curid(void);
extern unsigned int get_state(unsigned int pid);

extern unsigned int get_sync_chan_to(unsigned int chanid);
extern unsigned int get_sync_chan_count(unsigned int chanid);
extern unsigned int get_sync_chan_paddr(unsigned int chanid);

extern unsigned int get_kernel_pa(unsigned int pid, unsigned int vaddr);
extern void set_sync_chan_to(unsigned int chanid, unsigned int to);
extern void set_sync_chan_count(unsigned int chanid, unsigned int count);
extern void set_sync_chan_paddr(unsigned int chanid, unsigned int paddr);

extern void flatmem_copy(unsigned int to, unsigned int from, unsigned int len);
extern void thread_wakeup(unsigned int pid);

unsigned int syncreceive_chan(unsigned int pid, unsigned int vaddr, unsigned int rcount)
{
    unsigned int myid = get_curid();
    unsigned int sender_state = get_state(pid);

    if (sender_state == TCB_STATE_DEAD) {
        return MAX_BUFFSIZE+2;
    } else {
```
unsigned int sender_to = get_sync_chan_to(pid);

if (sender_to == myid) {
    unsigned int sender_count = get_sync_chan_count(pid);
    unsigned int sbuffpa = get_sync_chan_paddr(pid);

    unsigned int arecvcount = (rcount < sender_count) ? rcount : sender_count;

    unsigned int rbuffva = vaddr;

    unsigned int rbuffpa = get_kernel_pa(myid, rbuffva);
    flatmem_copy(rbuffpa, sbuffpa, arecvcount);

    set_sync_chan_to(pid, NUM_CHAN);
    set_sync_chan_paddr(pid, 0);
    set_sync_chan_count(pid, arecvcount);

    thread_wakeup(pid);

    return arecvcount;
} else {
    return MAX_BUFFSIZE+3;
}

This function also first checks if the sender is a dead process, and returns an error code if
that’s the case. It then reads the to field of the SyncIPCChan struct, check if the intended
receiver is the currently running process (i.e. the caller of the system call that invoked this
kernel function), and returns an error code if they don’t match.

The receive function then translates the virtual address of receiver buffer into a physical one
by calling get_kernel_pa. It then uses flatmem_copy to copy the data from the sender’s
buffer to the receiver’s buffer. flatmem_copy acts very similarly to the memcpy function
that most of us are familiar with, with one difference: the length parameter to flatmem_copy
is the number of words (i.e. 4 bytes) instead of the number of bytes.
Note that at this moment, the sender process is still asleep and it is the only process that has access to the send buffer, thus we can be sure that the sender’s buffer has not been tampered with. This is important because we can confidently read from memory and be sure that we’re not accessing invalid memory regions.

Once the copy is complete, the receive function would reset the struct fields to signify that the sender’s message has been received. The most important thing in this step is that the receive function sets the count field to the actual number of items sent across.

The receive function then returns the number of items sent across as a return value. It would be returned to used-side code such that user process would know how many items it received.

Once the receive function wakes up the sender, it would resume running once the scheduler restarts the sender process’ code execution, which would proceed to the call to syncsendto_chan_post.

```c
#define NUM_CHAN 64
#define MAX_BUFFSIZE 1024
extern unsigned int get_curid(void);
extern unsigned int get_sync_chan_to(unsigned int chanid);
extern unsigned int get_sync_chan_count(unsigned int chanid);
extern void set_sync_chan_to(unsigned int chanid, unsigned int to);

unsigned int syncsendto_chan_post(void)
{
    unsigned int myid = get_curid();
    unsigned int to = get_sync_chan_to(myid);

    if (to == NUM_CHAN) {
        return get_sync_chan_count(myid);
    } else {
        set_sync_chan_to(myid, NUM_CHAN);
        return MAX_BUFFSIZE+3;
    }
}
```

This function is relatively simple: it reads the to field of the struct, checks that it is equal to 64, which means the receiver has received the items in the buffer, and if the check passes, it reads the count field and use that as a return value so that the sending user process knows how
many items were sent across.

This concludes the design of my IPC implementation.

**Specification of synchronous IPC**

Just having the implementation is not enough to ensure correctness of the IPC extension. What exactly does correctness mean in this case?

In the existing CertikOS Coq proof code base, we use the following structure to represent the state of the kernel.

```plaintext
Record RData :=
  mkRData {
    MM: MMTable; (**r table of the physical memory’s information*)
    MMSize: Z; (**r size of MMTable*)
    vmxinfo: VMXInfo; (**r information of vmx*)
    CR3: globalpointer; (**r abstract of CR3, stores the pointer to page table*)
    ti: trapinfo; (**r abstract of CR2, stores the address where page fault happens*)
    pg: bool; (**r abstract of CR0, indicates whether the paging is enabled or not*)
    ikern: bool; (**r pure logic flag, shows whether it’s in kernel mode or not*)
    ihost: bool; (**r logic flag, shows whether it’s in the host mode or not*)
    HP: flatmem; (**r we model the memory from 1G to 3G as heap*)
    AC : ContainerPool; (**r container tree for all agents *)
    AT: ATable; (**r allocation table*)
    nps: Z; (**r number of the pages*)
    init: bool; (**r pure logic flag, show whether the initialization at this layer has been called or not*)
    pperm: PPermT; (**r physical page permission table *)
    PT: Z; (**r the current page table index*)
    ptpool: PMapPool; (**r page table pool*)
    idpde: IDPDE; (**r shared identity maps *)
    ipt: bool; (**r pure logic flag, shows whether the current page map is the kernel’s page map*)
  }
```
LAT: LATable; (**r allocation table*)
pb: PTBitMap; (**r [page table bit map], indicating which page
table has been used*)
smspool: SharedMemSTPool; (**r the shared-memory pool for IPC*)
kctxt: KContextPool; (**r kernel context pool*)
tcb: TCBPool; (**r thread control blocks pool*)
tdq: TDQueuePool; (**r thread queue pool*)
abtcb: AbTCBPool; (**r thread control blocks pool*)
abq: AbQueuePool; (**r thread queue pool*)
cid: Z; (**r current thread id*)
synchpool : SyncChanPool; (**r the channel pool for synchronous
IPC*)
uctxt : UContextPool; (**r user context pool*)
ept: EPT; (**r nested page table for guest mode*)
vmcs: VMCS; (**r virtual machine control structure for current VM
*)
vmx: VMX (**r VMX structure to store the extra registers of host
*)

The CertikOS code base defines a series of invariants that takes a value of type RData and
creates a proposition. For the new IPC functions we introduced, to prove their correctness, we
need to create specifications of those functions, and prove that they preserve these invariants.
We also need to prove that the C implementation match the specification.

Here I present the relevant specifications:

```plaintext
Function syncsendto_chan_pre_spec (chid vaddr scount : Z) (adt : 
RData) : option (RData * Z) :=
match (pg adt, ikern adt, ihost adt, ipt adt) with
| (true, true, true, true) ⇒
  if zle_lt 0 chid num_proc then
  match ZMap.get chid (abtcb adt) with
  | AbTCBValid st _ ⇒
    if ThreadState_dec st DEAD then
    Some (adt, 1024+2)
  else
    match ZMap.get (cid adt) (synchpool adt) with
    | SyncChanValid to _ _ ⇒
      match get_kernel_pa_spec (cid adt) vaddr adt with
      | Some skpa ⇒
        if zle_le 0 skpa Int.max_unsigned then
        if zeq (Int.unsigned to) 64 then
```

9
let asendval := Z.min (scount) (1024) in
let adt' := adt {syncchpool : ZMap.set (cid adt)
(SyncChanValid (Int.repr chid) (Int.repr skpa)
(Int.repr asendval))
(syncchpool adt)} in
Some (adt', (cid adt))
else None
else None
| _ ⇒ None
end
| _ ⇒ None
end
else None
| _ ⇒ None
end.

Function syncsendto_chan_post_spec (adt : RData) : option (RData * Z) :=
match (pg adt, ikern adt, ihost adt, ipt adt) with
| (true, true, true, true) ⇒
match ZMap.get (cid adt) (syncchpool adt) with
| SyncChanValid to paddr count ⇒
if zeq (Int.unsigned to) num_chan then
Some (adt, (Int.unsigned count))
else
let adt' := adt{syncchpool :
ZMap.set (cid adt)
(SyncChanValid (Int.repr num_chan) paddr
count) (syncchpool adt)} in
Some (adt', 1024+3)
| _ ⇒ None
end
| _ ⇒ None
end.

Function syncreceive_chan_spec (fromid vaddr rcount : Z) (adt : RData) :
option (RData * Z) :=
match (pg adt, ikern adt, ihost adt, ipt adt) with
| (true, true, true, true) ⇒
if zle_lt 0 fromid num_proc then
match ZMap.get fromid (abtcb adt) with
| AbTCBValid st _ ⇒
if ThreadState_dec st DEAD then
Some (adt, 1024+2)
else
  match ZMap.get fromid (syncchpool adt) with
    | SyncChanValid to spaddr scount ⇒
      if \( \text{zeq}(\text{Int.unsigned}\ to)\ (\text{cid}\ \text{adt})\) then
        let arecvcount := \(\text{Z.min}(\text{Int.unsigned}\ \text{scount})\)\ rcount in
        match get_kernel_pa_spec (cid adt) vaddr adt with
          | Some rbuffpa ⇒
            match flatmem_copy_spec arecvcount (Int.unsigned spaddr) rbuffpa adt with
              | Some adt1 ⇒
                let adt2 := adt1
                (syncchpool : ZMap.set fromid
                  (SyncChanValid (Int.repr num_chan) (Int.zero
                    arecvcount))
                  (syncchpool adt1))
                in
                match thread_wakeup_spec fromid adt2 with
                  | Some adt3 ⇒ Some (adt3, arecvcount)
                  | _ ⇒ None
                end
              | _ ⇒ None
            end
          | _ ⇒ None
        end
      else
        Some (adt, 1024+3)
    | _ ⇒ None
  end
end.

Note that these specification are written in the pure functional language Gallina used by Coq, all the states our C implementation depend on are made explicit in the specification.

Take the example of \text{syncsendto\_chan\_pre\_spec}, its parameters are the receiver’s id, user-side address to the buffer and the length of the buffer. It also takes a value of type \text{RData}, which is the kernel state. Compared to the C implementation, the kernel state is the implicit state that the send function depended on, which is made explicit in the spec.
Since the IPC functions modify the kernel state as well as has a return value, the output of the specifications is an optional pair of \( RData \) and \( Z \) (mathematical integer). Why optional? The \texttt{None} values are returned when code encounters unexpected inconsistencies in the kernel state.

**Proof checking**

For the actual theorems and proofs, please refer to the \texttt{Coq} files submitted with this report. I’ll briefly describe how a proof is checked.

How does \texttt{Coq} know that a proof is correct? This is possible thanks to Curry Howard Correspondence. Curry Howard Correspondence says that types in functional programming correspond to propositions in logic, and values that inhabit a type correspond to proofs for a proposition.

Here are some examples of these correspondence with a Haskell-like syntax:

```haskell
data True = I -- Correspond to the logical proposition True.
data Void   -- Correspond to the logical proposition False.
data Pair a b = Pair a b -- Correspond to Logical
                -- Conjunction.
    Pair a b ⇔ A ∧ B

data Either a b = Left a
                  | Right b -- Correspond to Logical
                  -- Disjunction.
    Either a b ⇔ A ∨ B

-- Function type
(->)  -- Correspond to Logical Implication.
    A ⇒ B -- Translates to
    a -> b -- assuming type a corresponds to proposition A
             -- and type b correspond to proposition B

-- How can we represent negation?
¬ a ⇔ a -> Void
```

A typechecker is a program that checks whether a value conforms to a type, thus, given Curry Howard Correspondence, a typechecker in Coq would check if a proof (which is really
just a functional value) conforms to a proposition (which is a type). This is how Coq provides a logical reasoning framework.

Acknowledgement

I’d like to thank Prof. Shao for giving me the opportunity to work on CertikOS, and I want to thank Newman Wu and Ronghui Gu for helping me navigate the code base and teaching me how to use Coq.

I have worked closely with Alex Lew, who is also a Maths and Computer Science major of Class 2015. It was a great experience working on proofs and discussing the design of IPC with Alex.

Future work

I plan on removing the alignment restriction of send/recv buffer. I would continue to work on this after exam period.

References


3. G.Heiser. From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels? SOSP, 2013