Untyped Allocation

Dan Pittman dan@auxon.io

September 10, 2021

Introduction

In seL4 parlance, an `Untyped` is a kernel object representing a yet-to-be-spoken-for chunk of physical memory. When a kernel object is created, it must be done so through the seL4 system call `seL4_Untyped_Retype`. `seL4_Untyped_Retype` take an untyped kernel object, an object representing the kernel object to retype this untyped chunk into, as well as the target object's *size*. That size is used by the kernel to track offsets in an untyped object. Sizes are described in terms of the number of "bits" in the object's size, where a bit x is used in the following equation $2^x = \text{Size}$. For example, a object whose size is 4, can be thought of as 2^4 bytes, or 16 bytes.

Because `Untyped`s are themselves kernel objects with an associated, but variable, size, they can also be the target of a `seL4_Untyped_Retype` invocation. In Ferros this is common practice: A process acquires an `Untyped` large enough to hold all of the capabilities it intends to create, then breaks that object down into the necessary sizes in a $\log_2$ fashion. I.e., an untyped object of size `6` can be broken into 2 of size `5`, 4 of size `4`, 8 of size `3`, and so on.

Buddy Allocator

A *buddy* allocator is one which does this $\log_2$ break down automatically by being given a desired allocation size, and then recursively breaking down the object until an object of that size is available. In Ferros, a buddy allocator is used by wrapping an `Untyped`, and then using the buddy algorithm to allocate chunks of it.

Construction

`OneHotUList` builds a `UList`, the buddy allocator's state, given the index at which that initial object ought to live. The size, sz, of the objects in the list are determined via the equation $sz = 2^{i+\text{MinUtSize}}$.

`OneHotUList`'s definition:

```rust
pub trait _OneHotUList: Unsigned {
    type Output;
}

type OneHotUList<Index> = <Index as _OneHotUList>::Output;

impl _OneHotUList for U0 {
    type Output = ULCons<U1, ULNull>;
}

impl<IHead: Bit, ITail: Unsigned> _OneHotUList for UInt<ITail, IHead>
where
    UInt<ITail, IHead>: Sub<U1>,
    Diff<UInt<ITail, IHead>, U1>: _OneHotUList,
    OneHotUList<Diff<UInt<ITail, IHead>, U1>>: UList,
{
    type Output = ULCons<U0, OneHotUList<Diff<UInt<ITail, IHead>, U1>>>;
}
```

Is directly translated to the buddy constructor:

```agda
buddy : ∀ n → Vec ℕ (suc n)
buddy zero = 1 :: []
buddy (suc n) = 0 :: buddy n
```

Allocation

To allocate an `Untyped`, we take its expected position in the list which can be acquired by solving for i in the equation stated in the previous section. Once we have the index, we begin the process of folding the buddy allocator's state, a list of available `Untyped`s, splitting if necessary until we have the size we'd set out for. get-untyped also tracks its number of splits. This is because each split creates a new kernel object whose `cptr` must have a slot to live in; the split count tells us how many cspace slots will be needed.

```agda
get-untyped : ∀ {n} → Fin n → Vec ℕ n → Maybe (ℕ × Vec ℕ n)
```

As stated before get-untyped is implemented as a fold over the allocator's state.

```
get-untyped index untypeds =
  let state = foldr GetUtState
                    fold-ut
                    record { uts = []
                    ; idx = (toℕ index)
                    ; splits = 0
                    ; done = false
                    }
                    untypeds
  in if (done state) then
     just ((splits state) , (uts state))
     else nothing
```

The fold function uses with-abstraction to build a truth table telling us whether:

1. We've found the object we're looking for.
2. Whether the index we've been given is zero.
3. Whether the untyped in the current position is absent.

```
fold-ut : ∀ {n} → ℕ → GetUtState n → GetUtState (suc n)
fold-ut ut state with done state | is-zero (idx state) | is-zero ut
```

We will break down the meaning of the truth table's states one by one.

1. T | _ | _

   We've found the untyped we're looking for and can be done. We just tack on the remaining untypeds as they are.

   ```
   ... | true | _ | _ = record { uts = ut :: (uts state)
                               ; idx = (idx state)
                               ; splits = (splits state)
                               ; done = (done state)
                               }
   ```

2. F | F | _

   We haven't reached our desired untyped size yet, our index is non-zero.

   ```
   ... | false | false | _ = record { uts = ut :: (uts state)
                                    ; idx = pred (idx state)
                                    ; splits = (splits state)
                                    ; done = (done state)
                                    }
   ```

3. F | T | F

   The index is zero and this cell has an untyped for us to take. Take it, mark it as done, move on.

   $$\begin{aligned} \ldots \mid \mathsf{false} \mid \mathsf{true} \mid \mathsf{false} = \mathsf{record}\ \{\ & \mathsf{uts} = (\mathsf{pred}\ ut) :: (\mathsf{uts}\ state) \\ ;\ & \mathsf{idx} = (\mathsf{idx}\ state) \\ ;\ & \mathsf{splits} = (\mathsf{splits}\ state) \\ ;\ & \mathsf{done} = \mathsf{true} \\ \}\ & \end{aligned}$$

4. F | T | T

   We've reached our index (or moved past it), and have not yet found an untyped of the right size. So we add one untyped in this cell, and count our splits.

   $$\begin{aligned} \ldots \mid \mathsf{false} \mid \mathsf{true} \mid \mathsf{true} = \mathsf{record}\ \{\ & \mathsf{uts} = (\mathsf{suc}\ ut) :: (\mathsf{uts}\ state) \\ ;\ & \mathsf{idx} = (\mathsf{idx}\ state) \\ ;\ & \mathsf{splits} = \mathsf{suc}\ (\mathsf{splits}\ state) \\ ;\ & \mathsf{done} = (\mathsf{done}\ state) \\ \}\ & \end{aligned}$$

This code corresponds to the following type-level implementation in Rust:

```rust
/// Type-level function to track the result of an allocation
pub trait _TakeUntyped<Index> {
    type ResultPoolSizes;
    type NumSplits;
}

// Index is non-zero, and there are pools left: recur with Index-1, and the
// remaining pools
impl<IndexU: Unsigned, IndexB: Bit, Head: Unsigned, Tail: UList>
    _TakeUntyped<UInt<IndexU, IndexB>>
    for ULCons<Head, Tail>
where
    UInt<IndexU, IndexB>: Sub<U1>,
    Diff<UInt<IndexU, IndexB>, U1>: Unsigned,

    Tail: _TakeUntyped<Diff<UInt<IndexU, IndexB>, U1>>,
    TakeUntyped_ResultPoolSizes<Tail, Diff<UInt<IndexU, IndexB>, U1>>: UList,
{
    type ResultPoolSizes =
        ULCons<Head,
```

```
                TakeUntyped_ResultPoolSizes<Tail,
                                            Diff<UInt<IndexU, IndexB>, U1>>>;
    type NumSplits = TakeUntyped_NumSplits<Tail,
                                            Diff<UInt<IndexU, IndexB>, U1>>;
}

// Index is 0, and the head pool has resources: remove one from it,
// with no splits.
impl<HeadU: Unsigned, HeadB: Bit, Tail: UList>
    _TakeUntyped<U0> for ULCons<UInt<HeadU, HeadB>, Tail>
where
    UInt<HeadU, HeadB>: Sub<U1>,
    Diff<UInt<HeadU, HeadB>, U1>: Unsigned,
{
    type ResultPoolSizes = ULCons<Diff<UInt<HeadU, HeadB>, U1>, Tail>;
    type NumSplits = U0;
}

// index is zero, and the head pool is empty. Take one from the next
// pool (which we will split, and return one of), and put one (the
// remainder) in the head pool.
impl<Tail: UList> _TakeUntyped<U0> for ULCons<U0, Tail>
where
    Tail: _TakeUntyped<U0>,
    U1: Add<TakeUntyped_NumSplits<Tail, U0>>,
    Sum<U1, TakeUntyped_NumSplits<Tail, U0>>: Unsigned,
    TakeUntyped_ResultPoolSizes<Tail, U0>: UList,
{
    type ResultPoolSizes = ULCons<U1, TakeUntyped_ResultPoolSizes<Tail, U0>>;
    type NumSplits = Sum<U1, TakeUntyped_NumSplits<Tail, U0>>;
}
```

The differences to note are that, of course, Rust's type system does not provide a right fold function. Instead, the truth table mentioned above is broken into the "match clauses"—the different type-level patterns for which `_TakeUntyped` is implemented.

Secondarily, in Agda, our implementation must be total, this is what we get through our use of Fin, for instance—an index cannot be larger than the vector comprising the allocator's state. However in Rust, we intentionally write partial type-level functions leaving the unhandled cases to cause a compilation error. For example, in Rust, if a request comes in for an unavailable size, we leave that case dangling, however in Agda, we must handle it via a Maybe.