Parallel & Concurrent Haskell 3:
Concurrent Haskell

Simon Marlow
Concurrent Haskell

• Recap:
  – concurrent programming is about *threads of control*
  – concurrency is necessary for dealing with multiple sources of input/output:
    • network connections
    • GUI, user input
    • database connections
  – *threads of control* let you handle multiple input/output sources in a *modular way*: the code for each source is written separately
In this part of the course we’re going to cover

- Basic concurrency
  - forking threads
  - communication and synchronisation
- Asynchronous exceptions
  - cancellation
  - timeouts
Forking threads

forkIO :: IO () -> IO ThreadId

• creates a new thread to run the IO ()
• new thread runs “at the same time” as the current thread and other threads
Interleaving example

```haskell
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  forkIO (forever (putChar 'A'))
  forkIO (forever (putChar 'B'))
  threadDelay (10^6)
```

$ ghc fork.hs
[1 of 1] Compiling Main            ( fork.hs, fork.o )
Linking fork ... 
$ ./fork | tail -c 300
AAAAAAAAAAABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
$
ThreadId

forkIO :: IO () -> IO ThreadId

– what can you do with a ThreadId?
  • check status with GHC.Conc.threadStatus (useful for debugging):

> import Control.Concurrent
> do { t <- forkIO (threadDelay (10^6)); GHC.Conc.threadStatus t }
ThreadRunning
> do { t <- forkIO (threadDelay (10^6)); yield; GHC.Conc.threadStatus t }
ThreadBlocked BlockedOnMVar

– Also:
  • compare for equality
  • kill / send exceptions (later...)
A note about performance

- GHC’s threads are *lightweight*

```
> ./Main 1000000 1 +RTS -s
Creating pipeline with 1000000 processes in it.
Pumping a single message through the pipeline.
Pumping a 1 messages through the pipeline.

  n   create    pump1    pump2    create/n    pump1/n    pump2/n
  s     s     s     s        us        us        us
1000000    5.98    1.77    1.77    5.98    1.77    1.77
```

- $10^6$ threads requires 1.5Gb – 1.5k/thread
  - most of that is stack space, which grows/shrinks on demand

- cheap threads makes it feasible to use them liberally, e.g. one thread per client in a server
Communication: MVars

- MVar is the basic communication primitive in Haskell.

```haskell
data MVar a -- abstract

newEmptyMVar :: IO (MVar a)
takeMVar    :: MVar a -> IO a
putMVar     :: MVar a -> a -> IO ()
```

[Diagram showing MVar usage with labels: Empty MVar and Blocked threads]
MVar is the basic communication primitive in Haskell.

```haskell
data MVar a -- abstract
newEmptyMVar :: IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

- `putMVar x`: Puts value `x` into the MVar.
- `takeMVar`: Takes the value from the MVar. Blocks if the MVar is empty.
- `Empty MVar`: Represents an empty MVar.
- `Blocked threads`: Threads waiting on the MVar to become non-empty.
Communication: MVars

• MVar is the basic communication primitive in Haskell.

```haskell
data MVar a -- abstract

newEmptyMVar :: IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

Diagram:
- `putMVar x`
- Empty MVar
- Blocked threads
- `takeMVar`
Communication: MVars

• MVar is the basic communication primitive in Haskell.

\[
\begin{align*}
data \ MVar \ a & \quad -- \ abstract \\
newEmptyMVar & : : IO (MVar \ a) \\
takeMVar & : : MVar \ a \rightarrow IO \ a \\
putMVar & : : MVar \ a \rightarrow a \rightarrow IO ()
\end{align*}
\]
MVar is the basic communication primitive in Haskell.

```haskell
data MVar a   -- abstract
newEmptyMVar :: IO (MVar a)
takeMVar      :: MVar a -> IO a
putMVar       :: MVar a -> a -> IO ()
```

```
putMVar x
x <- takeMVar
```

Empty MVar
Blocked threads
Communication: MVars

• MVar is the basic communication primitive in Haskell.

```
data MVar a  -- abstract
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

• And conversely: putMVar blocks when the MVar is full.
Examples

do m <- newEmptyMVar
    forkIO $ putMVar m 'a'
    takeMVar m
Examples

do m <- newEmptyMVar
    forkIO $ putMVar m 'a'
    takeMVar m

> do m <- newEmptyMVar; forkIO $ putMVar m 'a'; takeMVar m 'a'
>
Examples

```
do m <- newEmptyMVar
   forkIO (takeMVar m >>= print)
   putMVar m 'a'
```
Examples

do m <- newEmptyMVar
    forkIO (takeMVar m >>= print)
    putMVar m 'a'

> do m <- newEmptyMVar; forkIO (takeMVar m >>= print);
   putMVar m 'a'
   'a'
Examples

do m <- newEmptyMVar
    forkIO (takeMVar m >>= print)
    putMVar m 'a'

> do m <- newEmptyMVar; forkIO (takeMVar m >>= print); putMVar m 'a'
  'a'

> do m <- newEmptyMVar; forkIO (threadDelay 1000 >>=
    takeMVar m >>= print); putMVar m 'a'
> 'a'
do  m <- newEmptyMVar
    forkIO (do putMVar m 'a'; putMVar m 'b')
    takeMVar m >>= print
    takeMVar m >>= print
do  m <- newEmptyMVar
    forkIO (do putMVar m 'a'; putMVar m 'b')
    takeMVar m >>= print
    takeMVar m >>= print

> do m <- newEmptyMVar; forkIO (do putMVar m 'a'; putMVar m 'b'); takeMVar m >>= print; takeMVar m >>= print
 'a'
 'b'
do m <- newEmptyMVar
    takeMVar m
do m <- newEmptyMVar
    takeMVar m

> do m <- newEmptyMVar; takeMVar m
^CInterrupted.
>
Sometimes GHC can detect that a thread is deadlocked and send it an exception (*BlockedIndefinitelyOnMVar*)

You can catch this exception if you want.

- default behaviour is for the thread to die silently when it receives this exception => thread has been GC’d
do m <- newEmptyMVar
   forkIO (putMVar m 'a')
   forkIO (putMVar m 'b')
   takeMVar m
do m ← newEmptyMVar
    forkIO (putMVar m 'a')
    forkIO (putMVar m 'b')
    takeMVar m

> do m ← newEmptyMVar; forkIO (putMVar m 'a'); forkIO (putMVar m 'b'); takeMVar m
'a'
do m <- newEmptyMVar
    forkIO (putMVar m 'a')
    forkIO (putMVar m 'b')
    takeMVar m

> do m <- newEmptyMVar; forkIO (putMVar m 'a'); forkIO (putMVar m 'b'); takeMVar m 'a'

• What about the thread that lost the race?
• It deadlocks, receives the **BlockedIndefinitelyOnMVar** exception, and dies.
Example: overlapping I/O

• One common use for concurrency is to overlap multiple I/O operations
  – overlapping I/O reduces latencies, and allows better use of resources

  sequential I/O

  overlapped I/O

• overlapping I/O is easy with threads: just do each I/O operation in a separate thread
  – the runtime takes care of making this efficient, even when there are 100k+ blocked I/O operations

• e.g. downloading multiple web pages
getURL :: String -> IO String

do
    m1 <- newEmptyMVar
    m2 <- newEmptyMVar

    forkIO $ do
        r <- getURL "http://www.wikipedia.org/wiki/Shovel"
        putMVar m1 r

    forkIO $ do
        r <- getURL "http://www.wikipedia.org/wiki/Spade"
        putMVar m2 r

    r1 <- takeMVar m1
    r2 <- takeMVar m2
    return (r1, r2)
Abstract the common pattern

- Fork a new thread to execute an IO action, and later wait for the result

```haskell
newtype Async a = Async (MVar a)

async :: IO a -> IO (Async a)
async io = do
  m <- newEmptyMVar
  forkIO $ do r <- io; putMVar m r
  return (Async m)

wait :: Async a -> IO a
wait (Async m) = readMVar m

readMVar :: MVar a -> IO a
readMVar m = do
  a <- takeMVar m
  putMVar m a
  return a
```
do
    a1 <- async $ getURL "http://www.wikipedia.org/wiki/Shovel"
    a2 <- async $ getURL "http://www.wikipedia.org/wiki/Spade"
    r1 <- wait m1
    r2 <- wait m2
    return (r1,r2)
A driver to download many URLs

sites = ["http://www.bing.com",
        "http://www.google.com",
        ...
]

main = mapM (async.http) sites >>= mapM wait
where
    http url = do
        (page, time) <- timeit $ getURL url
        printf "downloaded: %s (%d bytes, %.2fs)\n" url (B.length page) time

downloaded: http://www.google.com (14524 bytes, 0.17s)
downloaded: http://www.bing.com (24740 bytes, 0.18s)
downloaded: http://www.wikipedia.com/wiki/Spade (62586 bytes, 0.60s)
downloaded: http://www.wikipedia.com/wiki/Shovel (68897 bytes, 0.60s)
downloaded: http://www.yahoo.com (153065 bytes, 1.11s)
An MVar is many things...

- **a lock**
  - `MVar()` behaves like a lock: full is unlocked, empty is locked
  - Can be used as a mutex to protect some other shared state, or a critical region

- **one-place channel**
  - Since an `MVar` holds at most one value, it behaves like an asynchronous channel with a buffer size of one

- **a container for shared state**
  - e.g. `MVar(Map key value)`
  - convert persistent data structure into ephemeral
  - efficient (but there are other choices besides `MVar`)

- **building block**
  - `MVar` can be used to build many different concurrent data structures/abstractions...
MVar = container for shared state

```haskell
import Data.Map as Map
import Control.Concurrent

type Name = String
type PhoneNum = String
type PhoneBook = Map Name PhoneNum

insertName :: MVar PhoneBook -> Name -> PhoneNum -> IO ()
insertName m name num = do
  book <- takeMVar m
  putMVar m (Map.insert name num book)

lookupNumber :: MVar PhoneBook -> Name -> IO (Maybe PhoneNum)
lookupNumber m name = do
  book <- readMVar m
  return (Map.lookup name book)
```
MVar = container for shared state

• taking the MVar locks the state
• putting the MVar updates the state and unlocks it again
• we can make any Haskell data structure into shared state this way. No need for special concurrent versions of data structures.
• Lazy evaluation can be used to avoid locking the state for too long:

```haskell
book <- takeMVar m
putMVar m (Map.insert name num book)
```

– Note that the insert is lazy, the MVar was only locked briefly (but beware of space leaks)
Unbounded buffered channels

- Interface:

  ```haskell
  data Chan a -- abstract

  newChan :: IO (Chan a)
  writeChan :: Chan a -> a -> IO ()
  readChan :: Chan a -> IO a
  ```

- write does not block (indefinitely)
- we are going to implement this with MVars
- can we just use

  ```haskell
  data Chan a = MVar [a]
  ```
- No: think about how `readChan` will block when the channel is empty
- but in both of these, writers and readers will conflict with each other
**Structure of a channel**

type Stream a = MVar (Item a)
data Item a = Item a (Stream a)
data Chan a = Chan (MVar (Stream a))
(MVar (Stream a))
**Implementation**

```plaintext
newChan :: IO (Chan a)
newChan = do
  hole <- newEmptyMVar
  readVar <- newMVar hole
  writeVar <- newMVar hole
  return (Chan readVar writeVar)

definitions:

writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar writeVar
  putMVar writeVar new_hole
  putMVar old_hole (Item val new_hole)

readChan :: Chan a -> IO a
readChan (Chan readVar _) = do
  stream <- takeMVar readVar
  Item val new <- takeMVar stream
  putMVar readVar new
  return val
```
Concurrent behaviour

- **Multiple readers:**
  - 2\textsuperscript{nd} and subsequent readers block here

- **Multiple writers:**
  - 2\textsuperscript{nd} and subsequent writers block here

- A concurrent read might block on old\_hole until writeChan fills it in at the end

```haskell
readChan :: Chan a -> IO a
readChan (Chan readVar _) = do
    stream <- takeMVar readVar
    Item val new <- takeMVar stream
    putMVar readVar new
    return val
```

```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
    new_hole <- newEmptyMVar
    old_hole <- takeMVar writeVar
    putMVar writeVar new_hole
    putMVar old_hole (Item val new_hole)
```
Adding more operations

• Add an operation for pushing a value onto the read end:
  
  ```haskell
  unGetChan :: Chan a -> a -> IO ()
  
  unGetChan (Chan readVar _) val = do
    new_read_end <- newEmptyMVar
    read_end <- takeMVar readVar
    putMVar new_read_end (Item val read_end)
    putMVar readVar new_read_end
  ```

• Doesn’t seem too hard:
But...

- This doesn’t work as we might expect:

  ```haskell
  > c <- newChan :: IO (Chan String)
  > forkIO $ do v <- readChan c; print v
 ThreadId 217
  > writeChan c "hello"
  "hello"
  > forkIO $ do v <- readChan c; print v
 ThreadId 243
  > unGetChan c "hello"
  ... blocks ....
  ```

- we don’t expect `unGetChan` to block
- but it needs to call `takeMVar` on the read end, and the other thread is currently holding that `MVar`
- No way to fix this...
- Building larger abstractions from `MVars` can be tricky
- Software Transactional Memory is much easier (later...)
Why go to all this trouble?

- Couldn’t we just build channels into the language, like Erlang, Go etc.?
  - MVar is a more fundamental primitive
  - You can build more than just channels with MVar, e.g. shared state
  - Haskell’s approach is to provide simple but powerful abstractions
  - Channels are provided (Control.Concurrent.Chan)
A note about fairness

- Threads blocked on an MVar are processed in FIFO order.
- No thread can be blocked indefinitely, provided there is a regular supply of putMVars (fairness).
- Each putMVar wakes exactly one thread, and performs the blocked operation atomically (single-wakeup).
A note about fairness

- Threads blocked on an MVar are processed in FIFO order.
- No thread can be blocked indefinitely, provided there is a regular supply of putMVars (fairness).
- Each putMVar wakes exactly one thread, and performs the blocked operation atomically (single-wakeup).
A note about fairness

- Threads blocked on an MVar are processed in FIFO order.
- No thread can be blocked indefinitely, provided there is a regular supply of putMVars (fairness).
- Each putMVar wakes exactly one thread, and performs the blocked operation atomically (single-wakeup).
A note about fairness

- Threads blocked on an **MVar** are processed in FIFO order.
- No thread can be blocked indefinitely, provided there is a regular supply of **putMVars (fairness)**.
- Each **putMVar** wakes exactly one thread, and performs the blocked operation atomically (**single-wakeup**).
A note about fairness

- Threads blocked on an MVar are processed in FIFO order.
- No thread can be blocked indefinitely, provided there is a regular supply of putMVars (fairness).
- Each putMVar wakes exactly one thread, and performs the blocked operation atomically (single-wakeup).
MVars and contention

- Fairness can lead to alternation when two threads compete for an MVar
  - thread A: takeMVar (succeeds)
  - thread B: takeMVar (blocks)
  - thread A: putMVar (succeeds, and wakes thread B)
  - thread A: takeMVar again (blocks)
  - cannot break the cycle, unless a thread is pre-empted while the MVar is full

- MVar contention can be expensive!
break...
Cancellation/interruption

(asynchronous exceptions)
Motivation

• Often we want to interrupt a thread. e.g.
  – in a web browser, the user presses “stop”
  – in a server application, we set a time-out on each client, close the connection if the client does not respond within the required time
  – if we are computing based on some input data, and the user changes the inputs via the GUI
Isn’t interrupting a thread dangerous?
Isn’t interrupting a thread dangerous?

• Most languages take the polling approach:
  – you have to explicitly check for interruption
  – maybe built-in support in e.g. I/O operations
Isn’t interrupting a thread dangerous?

• Most languages take the polling approach:
  – you have to explicitly check for interruption
  – maybe built-in support in e.g. I/O operations

• Why?
  – because fully-asynchronous interruption is too hard to program with, in an imperative language.
  – Most code is modifying state, so asynchronous interruption will often leave state inconsistent.
Isn’t interrupting a thread dangerous?

• Most languages take the polling approach:
  – you have to explicitly check for interruption
  – maybe built-in support in e.g. I/O operations

• Why?
  – because fully-asynchronous interruption is too hard to program with, in an imperative language.
  – Most code is modifying state, so asynchronous interruption will often leave state inconsistent.

• In Haskell, most computation is pure, so
  – completely safe to interrupt
  – furthermore, pure code cannot poll
Isn’t interrupting a thread dangerous?

- Most languages take the polling approach:
  - you have to explicitly check for interruption
  - maybe built-in support in e.g. I/O operations

- Why?
  - because fully-asynchronous interruption is too hard to program with, in an imperative language.
  - Most code is modifying state, so asynchronous interruption will often leave state inconsistent.

- In Haskell, most computation is pure, so
  - completely safe to interrupt
  - furthermore, pure code cannot poll

- Hence, interruption in Haskell is asynchronous
  - more robust: don’t have to remember to poll
  - but we do have to be careful with state in IO code
Interrupting a thread

- Throws the exception e in the given thread
- So interruption appears as an exception
  - this is good – we need exception handlers to clean up in the event of an error, and the same handlers will work for interruption too.
  - Code that is already well-behaved with respect to exceptions will be fine with interruption.

```
throwTo :: Exception e => ThreadId -> e -> IO ()
```

- threads can also catch interruption exceptions and do something – e.g. useful for time-out

```
bracket (newTempFile "temp")
  (\file -> removeFile file)
  (\file -> ...)
```
Example

• Let’s extend the async API with cancellation

• So far we have:

```haskell
newtype Async a = Async (MVar a)

async :: IO a -> IO (Async a)
async io = do
  m <- newEmptyMVar
  forkIO $ do r <- io; putMVar m r
  return (Async m)

wait :: Async a -> IO a
wait (Async m) = readMVar m
```

• we want to add:

```haskell
cancel :: Async a -> IO ()
```
• **cancel** is going to call `throwTo`, so it needs the `ThreadId`. Hence we need to add `ThreadId` to `Async`.

```
data Async a = Async ThreadId (MVar a)

async :: IO a -> IO (Async a)
async io = do
    m <- newEmptyMVar
    t <- forkIO $ do r <- io; putMVar m r
    return (Async t m)

cancel :: Async a -> IO ()
cancel (Async t _) = throwTo t ThreadKilled
```

• but what about **wait**? previously it had type:

```
wait :: Async a -> IO a
```

• but what should it return if the `Async` was cancelled?
• Cancellation is an exception, so wait should return the exception that was thrown.
  • This also means that wait will correctly handle exceptions caused by errors.

```haskell
data Async a = Async ThreadId (MVar (Either SomeException a))

async :: IO a -> IO (Async a)
async io = do
  m <- newEmptyMVar
  t <- forkIO (do r <- try action; putMVar m r)
  return (Async t m)

wait :: Async a -> IO (Either SomeException a)
wait (Async _ var) = takeMVar var

try :: Exception e => IO a -> IO (Either e a)
```
main = do
    as <- mapM (async.http) sites

    forkIO $ do
        hSetBuffering stdin NoBuffering
        forever $ do
            c <- getChar
            when (c == 'q') $ mapM_ cancel as

    rs <- mapM wait as
    printf "%d/%d finished\n" (length (rights rs)) (length rs)

Hit ‘q’ to stop the downloads
Points to note:

- We are using a large/complicated HTTP library underneath, yet it supports interruption automatically.
- Having asynchronous interruption be the default is very powerful.
- However: dealing with truly mutable state and interruption still requires some care...
Masking asynchronous exceptions

Problem:

- call `takeMVar`
- perform an operation \( (f :: a \rightarrow \text{IO} \ a) \) on the value
- put the new value back in the MVar
- if an interrupt or exception occurs anywhere, put the old value back and propagate the exception

```haskell
problem m f = do
  a <- takeMVar m
  r <- f a `catch` \e -> do putMVar m a; throw e
  putMVar m r
```

**Attempt 1**

```haskell
catch :: \text{Exception e} => \text{IO} \ a \rightarrow (\text{e} \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a
```
Masking asynchronous exceptions

• Problem:
  – call `takeMVar`
  – perform an operation \((f :: a \rightarrow \text{IO} \ a)\) on the value
  – put the new value back in the `MVar`
  – if an interrupt or exception occurs anywhere, put the old value back and propagate the exception

\[
\text{catch} :: \text{Exception} \ e \Rightarrow \text{IO} \ a \rightarrow (e \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a
\]

Attempt 1

```haskell
problem m f = do
  a <- takeMVar m
  r <- f a `catch` \e -> do putMVar m a; throw e
  putMVar m r
```
Masking asynchronous exceptions

- Problem:
  - call `takeMVar`
  - perform an operation \((f :: a \rightarrow \text{IO } a)\) on the value
  - put the new value back in the \text{MVar}
  - if an interrupt or exception occurs anywhere, put the old value back and propagate the exception

```haskell
problem m f = do
  a <- takeMVar m
  \(r \leftarrow f a \ `\text{catch}` \ e \rightarrow \) do putMVar m a; throw e
  putMVar m r
```

Attempt 1

\[\text{catch} :: \text{Exception } e \Rightarrow \text{IO } a \rightarrow (e \rightarrow \text{IO } a) \rightarrow \text{IO } a\]
Masking asynchronous exceptions

- Problem:
  - call `takeMVar`
  - perform an operation \((f :: a \rightarrow \text{IO} \ a)\) on the value
  - put the new value back in the `MVar`
  - if an interrupt or exception occurs anywhere, put the old value back and propagate the exception

```haskell
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

Attempt 2

```haskell
problem m f = do
  a <- takeMVar m
  (do r <- f a
      putMVar m r
      `catch` \e -> do putMVar m a; throw e
  )
```
Masking asynchronous exceptions

• Problem:
  – call `takeMVar`
  – perform an operation \((f :: a -> \text{IO} a)\) on the value
  – put the new value back in the \text{MVar}
  – if an interrupt or exception occurs anywhere, put the old value back and propagate the exception

```haskell
catch :: \text{Exception} e => \text{IO} a -> (e -> \text{IO} a) -> \text{IO} a
```

**Attempt 2**

```haskell
problem m f = do
  a <- takeMVar m
  r <- f a
  `catch` \(\text{e} \rightarrow \text{do putMVar} m a; \text{throw} \text{e}`
  putMVar m r
```
• Clearly we need a way to manage the delivery of asynchronous exceptions during critical sections.
• Clearly we need a way to manage the delivery of asynchronous exceptions during critical sections.
• Haskell provides mask for this purpose:

```haskell
mask :: ((IO a -> IO a) -> IO b) -> IO b
```
Clearly we need a way to manage the delivery of asynchronous exceptions during critical sections.

Haskell provides mask for this purpose:

```haskell
mask :: ((IO a -> IO a) -> IO b) -> IO b
```

Use it like this:

```haskell
problem m f = mask $ \restore ->
do putMVar m a;
  throw e ->
do putMVar m r;
  restore (f a) . catch \e ->
do takeMVar m r
```
Clearly we need a way to manage the delivery of asynchronous exceptions during critical sections.

Haskell provides mask for this purpose:

```
mask :: ((IO a -> IO a) -> IO b) -> IO b
```

Use it like this:

```
problem m f = mask $ \restore -> do
  a <- takeMVar m
  r <- restore (f a) `catch` \e -> do putMVar m a; throw e
  putMVar m r
```

mask takes as its argument a function (\restore -> ...)

Clearly we need a way to manage the delivery of asynchronous exceptions during critical sections.

Haskell provides mask for this purpose:

```
mask :: ((IO a -> IO a) -> IO b) -> IO b
```

Use it like this:

```
problem m f = mask $ \restore -> do
  a <- takeMVar m
  r <- restore (f a) `catch` \e -> do putMVar m a; throw e
  putMVar m r
```

- mask takes as its argument a function (\restore -> ...)
- during execution of (\restore -> ...), asynchronous exceptions are masked (blocked until the masked portion returns)
• Clearly we need a way to manage the delivery of asynchronous exceptions during critical sections.
• Haskell provides mask for this purpose:

```haskell
mask :: ((IO a -> IO a) -> IO b) -> IO b
```

• Use it like this:

```haskell
problem m f = mask $ \restore -> do
  a <- takeMVar m
  r <- restore (f a) `catch` \e -> do putMVar m a; throw e
  putMVar m r
```

• mask takes as its argument a function (\restore -> ...)
• during execution of (\restore -> ...), asynchronous exceptions are masked (blocked until the masked portion returns)
• The value passed in as the argument restore is a function (:: IO a -> IO a) that can be used to restore the previous state (unmasked or masked) inside the masked portion.
• So did this solve the problem?

```haskell
problem m f = mask $ \restore -> do
  a <- takeMVar m
  r <- restore (f a) `catch` \e -> do putMVar m a; throw e
  putMVar m r
```

• The code ensures that unexpected exceptions are not raised in the red portions, so we always safely put back the MVar, restoring the invariant.

• But: what if `takeMVar` blocks?
  • We are inside `mask`, so the thread cannot be interrupted. Bad!!
  • We didn’t really want to mask `takeMVar`, we only want it to atomically enter the masked state when `takeMVar` takes the value.
Solution:

- Operations that block are declared to be *interruptible*.
- An interruptible operation may receive asynchronous exceptions while blocked, even inside mask.

How does this help?

- **takeMVar** is now interruptible, so the thread can be interrupted while blocked
- in general, it is now very hard to accidentally write code that is uninterruptible for long periods (it has to be in a busy loop)
- Think of **mask** as *switch to polling mode*

  - interruptible operations poll
  - we know which ops poll, so we can use exception handlers
  - asynchronous exceptions become *synchronous* inside mask
hmm, don’t we have another problem now?

\[
\text{problem } m \ f = \text{mask } $ \backslash \text{restore } \rightarrow \text{ do } \\
\quad a \gets \text{takeMVar } m \\
\quad r \gets \text{restore } (f \ a) \ `\text{catch` } \backslash e \rightarrow \text{ do putMVar } m \ a; \text{ throw } e \\
\quad \text{putMVar } m \ r
\]

- putMVar is interruptible too!
- Interruptible operations only receive asynchronous exceptions if they actually block
  - In this case, we can ensure that putMVar will never block, by requiring that all accesses to this MVar use a take/put pair, not just a put.
  - Alternatively, use the non-blocking version of putMVar, tryPutMVar
Async-exception safety

- IO code that uses state needs to be made safe in the presence of async exceptions
- ensure that invariants on the state are maintained if an async exception is raised.
- We make this easier by providing combinators that cover common cases.
- e.g. the function problem we saw earlier is a useful way to safely modify the contents of an MVar:

\[
\text{modifyMVar} :: \text{MVar}\ a \rightarrow (a \rightarrow \text{IO}\ a) \rightarrow \text{IO}(\)
\]
Making Chan safe

• We had this:

```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
    new_hole <- newEmptyMVar
    old_hole <- takeMVar writeVar
    putMVar writeVar new_hole
    putMVar old_hole (Item val new_hole)
```
Making Chan safe

• We had this:

```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar writeVar
  putMVar writeVar new_hole
  putMVar old_hole (Item val new_hole)
```

danger!
Making Chan safe

• We had this:

```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar writeVar
  putMVar writeVar new_hole
  putMVar old_hole (Item val new_hole)
```

danger!

• use `mask_`

```haskell
writeChan (Chan _ writeVar) val = do
  new_hole <- newEmptyMVar
  mask_ $ do
    old_hole <- takeMVar writeVar
    putMVar writeVar new_hole
    putMVar old_hole (Item val new_hole)
```
Recap

- **Basic concurrency operations:**
  - `forkIO`
    - lightweight: make lots of them
  - `MVar`, `takeMVar`, `putMVar`
    - generalises mutexes, 1-place channels, state containers

- **Asynchronous exceptions:**
  - `throwTo`
    - throw an exception to another thread
  - `mask`
    - prevent exceptions from being thrown here for a while