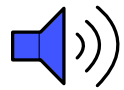
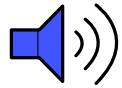
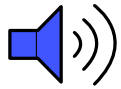
Lecture 14 – Concurrency 

Outline:

- Concurrency vs parallelism
- Threading
- Inter-thread communication
- Asynchrony

Concurrency vs Parallelism 

- Concurrency reduces the latency of a program with IO operations. Concurrent program runs faster than its sequential version even with one physical process.
- Parallelism reduces the runtime of a program by dividing computation tasks into multiple physical processes.
- Both concurrency and parallelism use threads and synchronization primitives.
- Parallel program may be implemented in models such as map-reduce that hides the details of concurrency control.
- Concurrent programs face a whole host of problems such as space/time leak, deadlock, starvation, and scheduling.

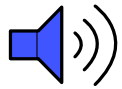
Threading 

- 'forkIO' creates a new thread that runs an IO action. The call returns an IO ThreadId immediately to the parent thread.

```
1 forkIO :: IO () -> IO ThreadID
```

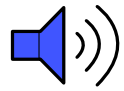
- Forked child thread races with the parent thread: one creates a file and one tests the existence of a file.

```
1 import Control.Concurrent
2 import System.Directory
3
4    forkIO (writeFile "tmp.txt" "some content")
5 >> doesFileExist "tmp.txt"
```

Reduce latency 

- Perceived latency in interactive applications can be reduced with threads. (Example adopted from Chapter 24 of RWH).

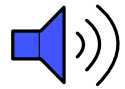
```
1  import qualified Data.ByteString.Lazy as L
2  import Codec.Compression.GZip (compress)
3
4  main = do
5      putStrLn "Enter a file to compress> "
6      line <- getLine          -- get file name
7
8      case line of
9        "" -> return ()        -- treat no name as "want to quit"
10       name ->
11             do content <- L.readFile name
12                            -- fork a thread to compress file
13                forkIO (compressFile name content)
14                main          -- run main again without waiting
15   where
16     compressFile path = L.writeFile (path ++ ".gz") . compress
```

Synchronization primitive 

- Just one primitive: MVar – a mutable variable.
  "MVar a" can be empty or filed with value a
  A thread blocks on a MVar by reading an empty MVar or writing to a full MVar.

```
1 newEmptyMVar :: IO (MVar a) -- creates an empty MVar
2
3 newMVar :: a -> IO (MVar a) -- creates a MVar with value a
4
5 takeMVar :: MVar a -> IO a  -- reads from a MVar
6
7 putMVar :: MVar a -> a -> IO () -- writes to a MVar
```

Communicate through MVar

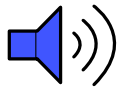


- Below is an example of communication between parent and child thread through MVar.

```
communicate = do
  m <- newEmptyMVar          -- mvar for communication

  forkIO $ do                -- fork a child thread
    putStrLn ("waiting for parent")
    v <- takeMVar m          -- block waiting
    threadDelay 1000000      -- delay child thread 1 second
    putStrLn ("received " ++ show v)

  threadDelay 1500000        -- delay parent thread 1.5 seconds
  putStrLn "parent sending"
  putMVar m "wake up!"       -- send data to MVar
```

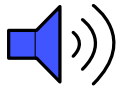
- The following output is expected.

```
*Main> communicate
waiting for parent
parent sending
(1.50 secs, 24,488 bytes)
*Main> received "wake up!"
```

Fetch files 

- Download and save the first two of the three files

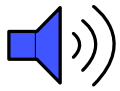
```
1  import Control.Concurrent
2  import qualified Data.ByteString.Lazy as L -- lazy list
3  import Network.HTTP.Conduit                -- http module
4
5  getThree = do
6    v <- newEmptyMVar              -- mvar to send file to main
7    id1 <- request "http://www.yahoo.com/"  v
8    id2 <- request "http://www.google.com/" v
9    id3 <- request "http://www.msn.com/"    v
10   takeMVar v >>= L.writeFile "test1.txt"   -- save 1st file
11   takeMVar v >>= L.writeFile "test2.txt"   -- save 2nd file
12   killThread id1                           -- kill them all
13   killThread id2
14   killThread id3
15  where
16    request url v = forkIO $ simpleHttp url -- http request
17                           >>= putMVar v     -- result to v
18                           >>  putStrLn url  -- print a message
```

Thread functions 

- Create, delay, and destroy a thread.

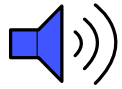
```
1  -- fork a new thread to run an IO and return thread ID
2  forkIO :: IO () -> IO ThreadId
3
4  -- delay current thread by micro-seconds.
5  threadDelay :: Int -> IO ()
6
7  -- kill thread by its ID
8  -- block until kill signal is received by the target
9  -- returns if the thread is already dead
10 killThread :: ThreadId -> IO ()
```

- Like MVar functions, every thread function results in an IO action.

Handling Exception 

- Exception should be handled with catch, handle, or try call.

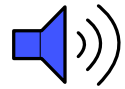
```haskell
import Control.Exception

-- catch excepton 'e' with a handler 'e -> IO a'
catch :: Exception e => IO a -> (e -> IO a) -> IO a

-- 'flip catch'
handle :: Exception e => (e -> IO a) -> IO a -> IO a

-- returns either exception 'e' or result 'a'
try :: Exception e => IO a -> IO (Either e a)

-- turn any exception 'e' into 'SomeException'
SomeException :: Exception e => e -> SomeException

-- 'show' function for exception
displayException :: Exception e => e -> String
```

Fetch file with exception handling 

- Catch exception in forked thread and print its message.

```
1  import Control.Exception
2
3  getThree = do
4    v <- newEmptyMVar                -- mvar to send file to main
5    id1 <- request "http://www.yahoo.com/"   v
6    id2 <- request "http://www.abcdefghijklmn.com/" v
7    id3 <- request "http://www.msn.com/"     v
8    takeMVar v >>= L.writeFile "test1.txt"    -- save 1st file
9    takeMVar v >>= L.writeFile "test2.txt"    -- save 2nd file
10   killThread id1                           -- kill them all
11   killThread id2
12   killThread id3
13  where
14    request url v = forkIO $(simpleHttp url -- http request
15                             >>= putMVar v      -- result to v
16                             >>  putStrLn url)  -- print a message
17                             `catch` handler    -- catch exception
18     where handler = \(SomeException e) ->
19                   putStrLn $ url ++ ": " ++ (displayException e)
```

- Can catch both http or thread exception

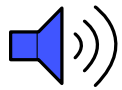
Fetch file with exception handling 

- Catch exception in main thread but can't stop the 3rd request

```
1  getThree' = do
2    v <- newEmptyMVar                -- mvar to send file to main
3    id1 <- request "http://www.yahoo.com/" v  -- file or error
4    id2 <- request "http://www.abccccddddd.com/" v
5    id3 <- request "http://www.msn.com/" v
6    deposit "test1.txt" v            -- save file or print error
7    deposit "test2.txt" v
8    deposit "test3.txt" v
9    killThread id1                   -- nothing to kill
10   killThread id2
11   killThread id3
12  where
13     deposit file v =
14       takeMVar v >>= either handler (L.writeFile file)
15     request url v = forkIO $
16       try (simpleHttp url) >>= putMVar v >> putStrLn url
17     handler =
18       \(SomeException e) -> putStrLn (displayException e)
```

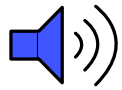
- 'try' runs 'IO a' and returns 'either' exception 'e' or result 'a'.

```
1  try :: Exception e => IO a -> IO (Either e a)
```

Concurrency control with MVar 

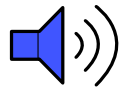
- Exception-safe modification and non-blocking access of MVar

```haskell
import Control.Concurrent.MVar

-- modify a MVar by updating its content 'a'
-- and return 'b' (often just to read 'a')
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b

-- modify a MVar by updating its content 'a' only
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()

-- atomically read the content of an MVar
readMVar :: MVar a -> IO a

-- put a new value into the MVar
-- and return the value taken
swapMVar :: MVar a -> a -> IO a

-- non-blocking take -- take Nothing or Just a
tryTakeMVar :: MVar a -> IO (Maybe a)
-- non-blocking put -- True if succeed in put
tryPutMVar :: MVar a -> a -> IO Bool
-- non-blocking read -- read Nothing or Just a
tryReadMVar :: MVar a -> IO (Maybe a)
```

Concurrency control with MVar 

- Use MVar v1 to count successful requests and MVar v2 to notify main thread to cancel.

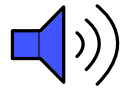
```
1  getThree'' = do
2    v1 <- newMVar 0                          -- counter mvar
3    v2 <- newEmptyMVar                       -- cancel mvar
4    id1 <- request "http://www.yahoo.com/" "test1.txt" v1 v2
5    id2 <- request "http://www.google.com/" "test2.txt" v1 v2
6    id3 <- request "http://www.msn.com/" "test3.txt" v1 v2
7
8    takeMVar v2                              -- start cancellation
9    killThread id1; killThread id2; killThread id3
10  where
11     request url f v1 v2 = forkIO $
12          do d <- simpleHttp url            -- update count below
13             count <- modifyMVar v1 (\x -> return (x+1, x+1))
14             putStrLn $ (show count) ++ ": " ++ url
15             L.writeFile f d                -- save the file
16             if count >= 2
17             then putMVar v2 ()             -- send cancel signal
18             else return ()
19          `catch` handler                   -- catch exception
20          where handler = \(SomeException e) ->
21                  putStrLn (url ++ ": " ++ displayException e)
```

Communicate with Channels 

- Send/receive messages using Channel, which is an infinite buffer built from MVars.

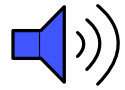
```
1  import Control.Concurrent.Chan
2
3  -- make a new channel of type 'a'
4  newChan :: IO (Chan a)
5
6  -- write 'a' to the channel and return immediately
7  writeChan :: Chan a -> a -> IO ()
8
9  -- read 'a' from the channel and blocks if empty
10 readChan :: Chan a -> IO a
11
12 -- duplicate a channel, which allows broadcast
13 dupChan :: Chan a -> IO (Chan a)
```

- Channel is never closed and a thread can block on reading a channel forever even if no thread is sending data to it.

Send messages through channels 

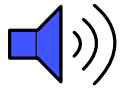
- Send success and exception messages to main via channel.

```
getThree''' = do
  let urls = [("http://www.yahoo.com/", "test1.txt"),
              ("http://www.ababbaa.com/", "test2.txt"),
              ("http://www.msn.com/", "test3.txt")]
  v1 <- newMVar 0                       -- counter mvar
  v2 <- newEmptyMVar                    -- cancel mvar
  ch <- newChan                         -- message channel
  ids <- mapM (\(url,path) -> request url path v1 v2 ch) urls
  takeMVar v2                           -- start cancellation
  mapM killThread ids                   -- cancel all
  mapM_ (\_ -> readChan ch >>= putStrLn) ids
 where
    request url f v1 v2 ch = forkIO $
          do d <- simpleHttp url        -- get file; update counter
             count <- modifyMVar v1 (\x -> return (x+1, x+1))
             writeChan ch $ (show count) ++ ": " ++ url
             L.writeFile f d            -- save file
             if count >= 2
             then putMVar v2 ()         -- signal cancel
             else return ()
          `catch` handler               -- catch exception
          where handler = \(SomeException e) ->
             writeChan ch (url ++ ": " ++ displayException e)
```

Send messages through list of MVars 

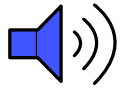
- Main thread assigns each child thread an empty MVar.

```
1  getThree'''' = do
2    let urls = [("http://www.yahoo.com/", "test1.txt"),
3                ("http://www.google.com/", "test2.txt"),
4                ("http://www.msn.com/", "test3.txt")]
5    v1 <- newMVar 0                                  -- counter mvar
6    v2 <- newEmptyMVar                               -- cancel mvar
7    vs <- mapM (\_ -> newEmptyMVar) urls             -- message mvars
8    ids <- mapM (\((url, path), v) ->                -- launch threads
9                    request url path v1 v2 v) $ zip urls vs
10   takeMVar v2                                      -- start cancellation
11   mapM killThread ids                              -- cancel all
12   messages <- mapM (tryTakeMVar) vs                -- nonblocking take
13   mapM_ (putStrLn . show) messages                 -- print message
14 where
15    request url f v1 v2 v = forkIO $
16          do d <- simpleHttp url
17             count <- modifyMVar v1 (\x -> return (x+1, x+1))
18             putMVar v $ (show count) ++ ": " ++ url
19             L.writeFile f d
20             if count >= 2 then putMVar v2 () else return ()
21          `catch` handler
22          where handler = \(SomeException e) ->
23                  putMVar v $ (url ++ ": " ++ displayException e)
```

Asynchrony 

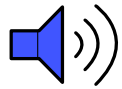
- Perform asynchronous IO without using threads or MVar.

```haskell
import Control.Concurrent.Async

-- run an action in a separate thread
async :: IO a -> IO (Async a)

-- blocking wait for an async action to return
wait :: Async a -> IO a

-- cancel an async action
cancel :: Async a -> IO ()

-- race two actions and cancel the loser
race  :: IO a -> IO b -> IO (Either a b)
race_ :: IO a -> IO b -> IO ()

-- run two actions concurrently
concurrently  :: IO a -> IO b -> IO (a, b)
concurrently_ :: IO a -> IO b -> IO ()

-- run a list of actions concurrently
mapConcurrently  :: (a -> IO b) -> [a] -> IO [b]
mapConcurrently_ :: (a -> IO b) -> [a] -> IO ()
```

Asynchrony 

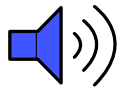
- Run http requests as a list of async actions

```
1  import Control.Concurrent.Async
2
3  getAsync =  do
4    let urls = [("http://www.yahoo.com/", "test1.txt"),
5                ("http://www.google.com/", "test2.txt"),
6                ("http://www.abcdefghijklmn.com/", "test3.txt")]
7
8    mapConcurrently_ request urls -- run requests w. no results
9   where
10     request (url, f) = do
11           d <- simpleHttp url
12           putStrLn url
13           L.writeFile f d
14           `catch` handler
15           where handler = \(SomeException e) ->
16                     putStrLn (url ++ ": " ++ displayException e)
```

Asynchrony 

- Run http requests as a list of async actions, where messages are printed independently.

```
-- output
*Main> getAsync
http://www.google.com/
http://www.abcdefghijklmn.com/: HttpExceptionRequest Request {
  host                 = "www.abcdefghijklmn.com"
  port                 = 80
  secure               = False
  requestHeaders       = [("Connection","close")]
  path                 = "/"
  queryString          = ""
  method               = "GET"
  proxy                = Nothing
  rawBody              = False
  redirectCount        = 10
  responseTimeout      = ResponseTimeoutDefault
  requestVersion       = HTTP/1.1
}
 (ConnectionFailure user error (Network.Socket.gai_strerror nc
http://www.yahoo.com/
(1.08 secs, 52,608 bytes)
```

Asynchrony 

- Return messages to the main thread.

```haskell
import Control.Concurrent.Async

getAsync' =  do
  let urls = [("http://www.yahoo.com/", "test1.txt"),
              ("http://www.google.com/", "test2.txt"),
              ("http://www.abcdefghijklmn.com/", "test3.txt")]

  messages <- mapConcurrently request urls -- run requests
  mapM_ putStrLn messages                  -- print messages
 where
    request (url, f) = do
          d <- simpleHttp url
          L.writeFile f d
          return url
          `catch` handler
          where handler = \(SomeException e) ->
                    return (url ++ ": " ++ displayException e)
```

Asynchrony

- Return messages to the main thread, where messages are printed in the same order as the actions.

```
-- output
*Main> getAsync'
http://www.yahoo.com/
http://www.google.com/
http://www.abcdefghijklmn.com/: HttpExceptionRequest Request {
  host                 = "www.abcdefghijklmn.com"
  port                 = 80
  secure               = False
  requestHeaders       = [("Connection","close")]
  path                 = "/"
  queryString          = ""
  method               = "GET"
  proxy                = Nothing
  rawBody              = False
  redirectCount        = 10
  responseTimeout      = ResponseTimeoutDefault
  requestVersion       = HTTP/1.1
}

 (ConnectionFailure user error (Network.Socket.gai_strerror nc
(1.07 secs, 541,480 bytes)
```