The Structuring of Systems Using Upcalls
Overview

A method for implementing layers with synchronous procedure calls between layers

**Upcalls** – a lower layer calls an upper layer

**Multi-task module** – multithreading
Outline

Background
Upcalls and Multi-task Modules
Example
Advantages
Swift Operating System
Problems
Conclusions
Background

1968 – THE operating system
  - OS using strict top-down layering model

1982 – ISO network protocol stack
  - protocol stack using strict layering model

1985 – Swift operating system
  - OS with performance optimizations
  - inspired by networking problems
Layered Operating System

- System Services
- File System
- Memory and I/O Device Management
- Processor Scheduling
- Hardware
Advantages of Layered OS

- Black box abstraction
- Divide and conquer approach
- Easier to test and debug
- Easier to modify
- Acyclic dependencies
- Flow control matches trust relationships
7-Layer OSI Model

1. Physical
2. Data link
3. Network
4. Transport
5. Session
6. Presentation
7. Application
Disadvantages of Layered OS

Poor match for bottom-up control flow
Example: TCP congestion control

Other problems:
- Interrupt handlers are not scheduled
- They are typically asynchronous
- Buffering and copying overhead for data
Upcalls and Multi-Task Modules

Upcall
- Lower layer calls an upper layer via procedure call

Multi-Task Modules
- Multithreaded subroutines in a layer
- State variables in shared memory
- Subroutines callable from above or below
System Organization
Example - Network Protocol

- Display
- Transport
- Network

Create | Receive | Send
Control Flow in Network Protocol
Example - Display Layer

display-start():
    local-port = transport-open(display-receive)
end

display-receive(char):
    write char to display
end

display-get-data (packet):
    //upcalled by to send data
    copy data from keyboard-buffer into packet
end

display-keyboard-handler():
    //upcalled by interrupt handler for keyboard
get character from keyboard device
    and put in keyboard-buffer
transport-arm-for-send
    (port, display-get-data)
end
Example - Transport Layer

transport-open(receive-handler):
    local-port = net-open(transport-receive)
    transport-handler-array(local-port)=
        receive-handler
    return(local port)
end

transport-get-port(packet):
    //upcalled by interrupt layer
    extract port from packet
    return(port)
end

transport-receive(packet,port):
    //upcalled by net layer
    handler=transport-handler-array(port)
    validate packet header
    if packet authorizes sending then
        if want-to-send(port)
            then wakeup-task(send-task-id)
        else ok-to-send(port)=true
    for each char in packet do
        handler(char)
end

transport-arm-for-send (port,send-handler):
    transport-send-handler-array (port)=
        send handler
    if ok-to-send(port)
        then wakeup-task(send-task-id)
    else want-to-send(port)=true
end

transport-send(port):
    //runs in task identified by send-task-id
    if ok-to-send(port)=false then block()
    allocate packet and fill in headers
    send-handler=
        transport-send-handler-array(port)
    send-handler(packet)
    //upcall display level to get data
    net-send(packet,port)
    ok-to-send(port)=false
    want-to-send(port)=false
end
Example - Network Layer

```python
net-open(receive-handler):
    port = generate-uid()
    task-id = create-task (net-receive
        (port, receive-handler))
    net-task-array(port) = task-id
    return(port)
end

net-receive(port, handler):
    handler = net-handler-array(port)
    do forever
        remove packet from per port queue
        handler(packet, port)
        block()
    end
end

net-dispatch(): //upcalled by interrupt handler
    read packet from device
    restart device
    port=transport-get-port(packet)
    put packet on per port queue
    task-id = net-task-array(port)
    wakeup-task(task-id)
end
```

```python
net-send(packet, port):
    start net device to send packet
end
```
Example – Create Task

display-start:
  local-port = transport-open(display-receive)
end

transport-open(receive-handler):
  local-port = net-open(transport-receive)
  transport-handler-array(local-port) = receive-handler
  return(local port)
end

net-open(receive-handler):
  port = generate-uid()
  task-id = create-task (net-receive
                  (port, receive-handler))
  net-task-array(port) = task-id
  return(port)
end

net-receive(port, handler):
  handler = net-handler-array(port)
  do forever
    remove packet from per port queue
    handler(packet, port)
    block()
  end
end
Example – Receive Task

display-receive(char):
  write char to display
end

transport-receive(packet, port):
  //upcalled by net layer
  handler=transport-handler-array(port)
  validate packet header
  if packet authorizes sending then
    if want-to-send(port)
      then wakeup-task(send-task-id)
    else ok-to-send(port)=true
  for each char in packet do handler(char)
end

net-receive(port, handler):
  handler = net-handler-array(port)
  do forever
    remove packet from per port queue
    handler(packet, port)
    block()
  end
end

transport-get-port(packet):
  //upcalled by interrupt layer
  extract port from packet
  return(port)
end

net-dispatch(): //upcalled by interrupt handler
  read packet from device
  restart device
  port=transport-get-port(packet)
  put packet on per port queue
  task-id = net-task-array(port)
  wakeup-task(task-id)
Example – Send Task

```plaintext
display-get-data (packet):
  // upcalled by to send data
  copy data from keyboard-buffer into packet
end

display-keyboard-handler():
  // upcalled by interrupt handler for keyboard
get character from keyboard device
and put in keyboard-buffer
transport-arm-for-send
  (port, display-get-data)
end

transport-send(port):
  // runs in task identified by send-task-id
  if ok-to-send(port)=false then block()
allocate packet and fill in headers
send-handler=
    transport-send-handler-array(port)
send-handler(packet)
  // upcall display level to get data
net-send(packet, port)
ok-to-send(port)=false
want-to-send(port)=false
end

transport-arm-for-send (port, send-handler):
  transport-send-handler-array (port)=
    send handler
  if ok-to-send(port)
    then wakeup-task(send-task-id)
  else want-to-send(port)=true
end

net-send(packet, port):
  start net device to send packet
end
```
Advantages of Upcalls

Upward, synchronous control transfer via procedure call is more efficient than IPC.
No need for data buffering between layers.
Simplicity of implementation.
Lower layer can ask advice from upper layer resulting in simplification of algorithm.
Advantage of Multi-Task Modules

Subroutine interfaces are easier to deal with than IPC interfaces
No system wide IPC message format needed
Decisions about task usage can be made later, the tasks are not hard coded into layers
Acknowledgements can be sent with new data packet - Piggybacking
Swift Operating System

Explore upcall and multi-module usage
Single address space with a typesafe language
Monitor locks control access to shared state
Initially used for network protocols
Test to determine if useful in other contexts
Develop solutions for upcall problems
Problems with Upcall

Violates basic principle of layering
- violates trust dependencies
- upcall failure can leave lower layer in an unstable condition

Difficult to recover resources when an upcall fails

What if an upcall causes a recursive down call that changes layer state?
Mitigating Upcall Failures

2 classes of data
- Client-specific private data which is expendable
- Shared data shared that must be unlocked and consistent before upcalls

Expendable tasks
Layer-specific cleanup via system procedures
Timer or User to detect infinite loop
Solutions to Recursive Calls

1. Put variables in consistent state before upcall and reevaluate them on return

2. Prohibit recursive downcalls so that variables can be left locked or inconsistent when upcalling

3. Downcall queues the work for later execution

4. Downcall is restricted in its actions

5. Downcall is replaced by extra arguments or it is replaced by a second upcall to query
Conclusions

Methodology is useful for operating systems
- Upcalls & multi-task modules are efficient
- useful for network protocols and I/O
- typesafe language gives efficient protection in shared address space

Process per layer model is inefficient

Parallel systems need shared memory for efficient communication between processes