# Advanced Programming Languages

## Providing programming support for safety, modularity, and efficiency

*Peter Lee*

**Carnegie Mellon University**

**January, 1995**

Carnegie Mellon

# From yesterday's talks on OS design...

blah... blah... blah..

blah... blah... blah... blah...

blah... blah... blah... blah...

blah...

blah...

blah... blah... blah...

Carnegie Mellon

# From yesterday's talks on OS design...

**<u>P. Dale (OSF):</u>** blah... blah... blah..

blah... blah... ***<u>Modularity!</u>*** blah... blah...

blah... blah... blah... blah...

***<u>Crucial for delivery and maintenance!</u>***

blah... **Performance?** ***<u>Can be better due to</u>***

blah... ***<u>specification of more constraints.</u>***

blah... blah... blah...

Carnegie Mellon

# From yesterday's talks on OS design...

**J. Lepreau (Utah):** blah... blah... blah..

blah... blah... ***Modularity!*** blah... blah...

blah... blah... blah... blah...

***Specified interfaces! Adaptive specialization!***

blah... **How?** ***Use a Scheme interpreter!***

blah... ***(Rich set of optimization opportunities)***

blah... blah... blah...

# From yesterday's talks on OS design...

**S. Lucco (CMU):** blah... blah... blah..

blah... blah... ***Modularity!*** blah... blah...

blah... blah... blah... blah...

***But VM is too generic and too slow!***

blah... **Philosophy?** ***RISC! Shift attention***

blah... ***to compiler optimizations.***

blah... blah... blah...

# For this panel:

**P. Lee (CMU):** blah... blah... blah..

blah... blah... ***Modularity!*** blah... blah...

blah... blah... blah... blah...

***Because everyone else is right!***

blah... **Performance? Safety? Philosophy?**

blah... ***Use a type-safe programming language!***

blah... blah... blah...

# Modularity in OS Design

## Software modularity has emerged as an important principle in structuring systems.

- System structuring motivations:
  - composability and customization of services
  - safety, extensibility, reliable reconfiguration
- Performance motivations:
  - avoid the overhead of hardware safety mechanisms
  - earlier binding of properties
- Software engineering motivations:
  - maintenance and team development
  - code reuse

# Customization, Reconfiguration, Composability

**To take advantage of information from the application, basic services must be customizable.**

- **software modularity is the key**
- **but there is a tension between:**
  - **providing good properties with sufficient expressiveness...**
  - **...and efficiency**

**Much of the research in OS/network design has to do with achieving *modularity* without sacrificing *performance*.**

- **x-kernel, SPIN, SFI,*etc.***

# Composable Protocols in the FoxNet

## In the FoxNet, protocols can be composed safely and efficiently to build standard protocol stacks:

```
structure Dev = Dev();

structure Eth = Eth (structure Lower = Dev);

structure Arp = Arp (structure Lower = Eth);

structure Ip = Ip (structure Lower = Arp);

structure Icmp = Icmp (structure Lower = Ip);

structure Tcp = Tcp (structure Lower = Icmp
                     val checksum = true);
```

## Essentially a form of “dynamic” linking.

# Composable Protocols, cont'd

**The composable protocols provide "mix-and-match" composability, allowing *ad hoc* protocols for efficiency:**

```
structure Len = Len_Protocol (structure Lower = Arp);

structure Glue = Pseudo_Ip_Protocol (structure Lower = Len);

structure Tcp_over_Eth = Tcp_Protocol (structure Lower = Glue
                                       val checksum = false);
```

- **The requirement for the "glue" protocol is automatically enforced at link-time by the type system of the Standard ML programming language.**

Carnegie Mellon

# Specializing Modules

**Some of the cost of modularity can be eliminated by *specializing* modules when they are composed.**

```
structure Tcp = Tcp (structure Lower = Icmp
                     val checksum = true);
```

**In a language such as SML, a partial evaluator might automate such specializations.**

# Partial Evaluation

$$P(x, y) = z$$

$$\mathrm{mix}(P, x) = P_x \qquad P_x(y) = z$$

$$\mathrm{mix}(\mathrm{mix}, P) = \mathrm{mix}_P \qquad \mathrm{mix}_P(x) = P_x$$

$$\mathrm{mix}(\mathrm{mix}, \mathrm{mix}) = \mathrm{mix}_{\mathrm{mix}} \qquad \mathrm{mix}_{\mathrm{mix}}(P) = \mathrm{mix}_P$$

# Partial Evaluation for Link-time Specialization

```
structure Tcp = Tcp (structure Lower = Icmp
                     val checksum = true);
```

- **mix (Tcp,struct Lower=Icmp  checksum=true end)  = $\mathbf{Tcp_{Icmp}}$**

- **(comp o mix) (Tcp, struct Lower=Icmp  checksum=true end) =  $\mathbf{Tcp_{Icmp}}$**

Carnegie Mellon

# Partial Evaluation for RTCG

Consider **send** `: connection -> message -> unit`

- `mix` (**send**, **connection**) = $\textbf{send}_{\textbf{conn}}$
- **(comp o mix) (send, connection)** = $\textbf{send}_{\textbf{conn}}$
- `mix` (**comp o mix**, **send**) = $\textbf{comp o mix}_{\textbf{send}}$
- **(comp o mix) (comp o mix, send)** = $\textbf{comp o mix}_{\textbf{send}}$

Carnegie Mellon

# Lessons, Conclusions

## A well-designed programming language provides

- programmer support for modularity, safety, resource management
- expressive power to define and manipulate modules and composable systems
- opportunities for compile-time, link-time, and even run-time transformations to achieve efficiency and modularity at the same time

## The many conflicting demands of modern systems are best solved by putting more power in the programmer's hands.

Carnegie Mellon