User-memory-based Kernel Objects

Bryan Ford

Department of Computer Science
University of Utah

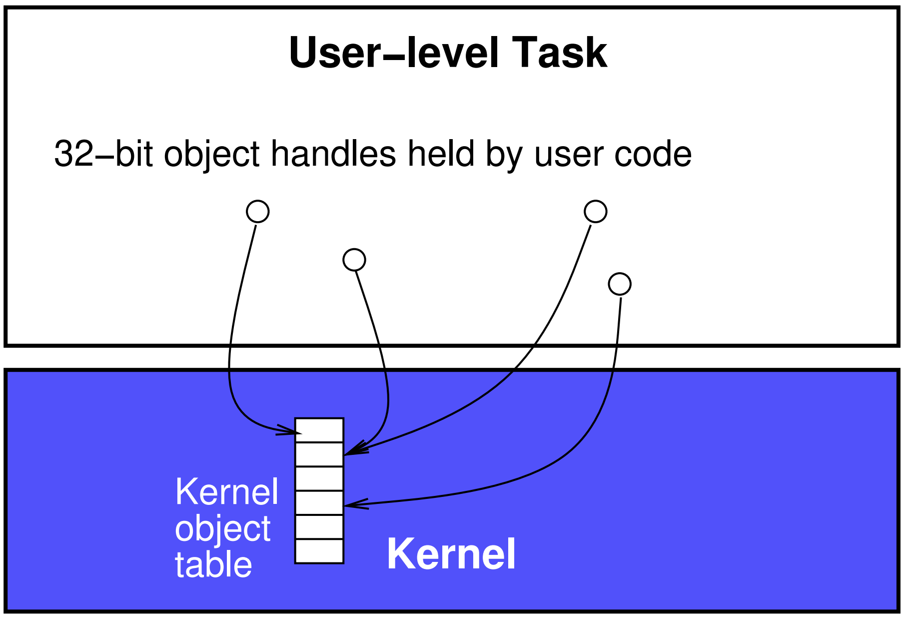
http://www.cs.utah.edu/~baford/

Traditional Kernel Objects

Kernel objects are traditionally presented to user code through "handles" of some kind, e.g.:

```
int fd = open(...);
read(fd, ...);
```

The actual kernel object the handle refers to is stored "somewhere" in the kernel, invisible to user code.

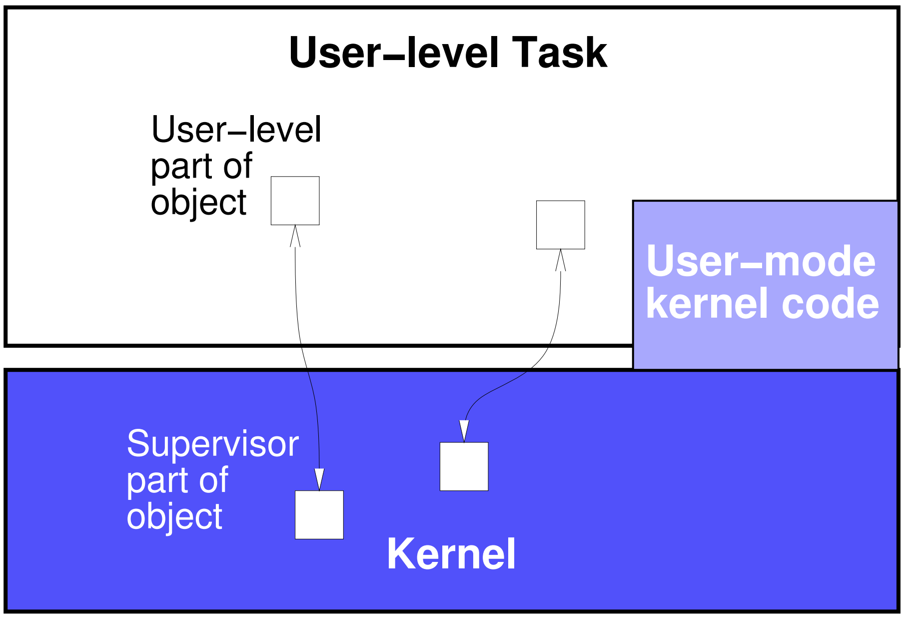


User-memory-based Objects

Suppose instead user code provides a small region of its own memory for the kernel to use as it desires while the object is active:

```
struct file_descriptor fd;
open(&fd, ...);
read(&fd, ...);
```

Kernel objects then have two parts: a “user-mode part” and a “supervisor-mode part”.



Object Invocation

1. User code jumps to a well-known address at which kernel code is mapped executable by user code, and passes the address of the user-mode part of the object.

2. If user-mode kernel code needs access to privileged state, it enters supervisor mode and uses the contents of the user-mode part of the object to find the supervisor-mode part.

Advantages of this model

Performance is no worse in worst case.

Many supervisor-mode calls can be optimized away transparently in common cases.

Examples:

- Uncontended mutex locks/unlocks

- Context switches between threads in the same task (like Scheduler Activations)

Not just a library veneer!

- Kernel knows the location of the user-mode part of every object; can update it when necessary.

- Kernel can provide special interrupt/fault semantics to kernel code running in user mode. (like RAS but more flexible)

Status

Initial prototype kernel running, using special rollback/roll-forward interruption semantics to optimize critical paths.