CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Fall 2018

Week 7: Capabilities

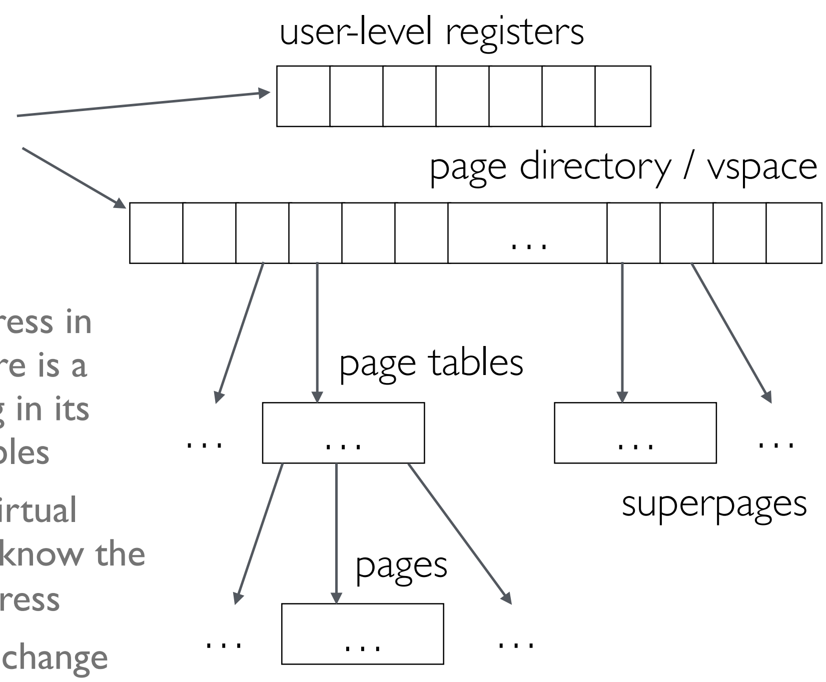
Introduction

Capabilities

- A **capability** is a "token" that grants certain rights to the holder [Dennis and Van Horn, 1966]
- Aligns with the "principle of least privilege" in computer security
- Supports fine grained access control and resource control
- Used in prior OSes and microkernels, including KeyKOS, Mach, EROS, OKL4 V2.1, and seL4
- Goals for today:
  - introduce the concepts in a simple example/framework
  - prepare for lab exercises to explore these ideas in practice

struct Process

```
struct Process {
  struct Context ctxt;
  struct Pdir*   pdir;
};
```

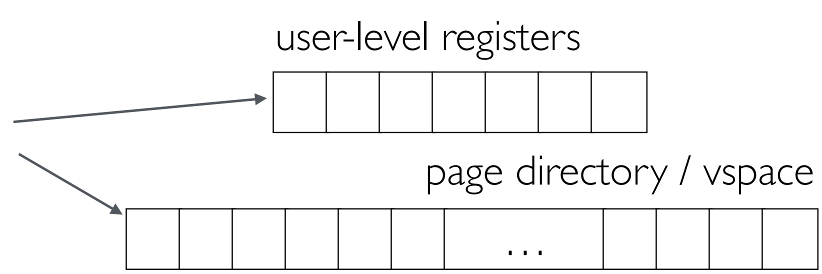


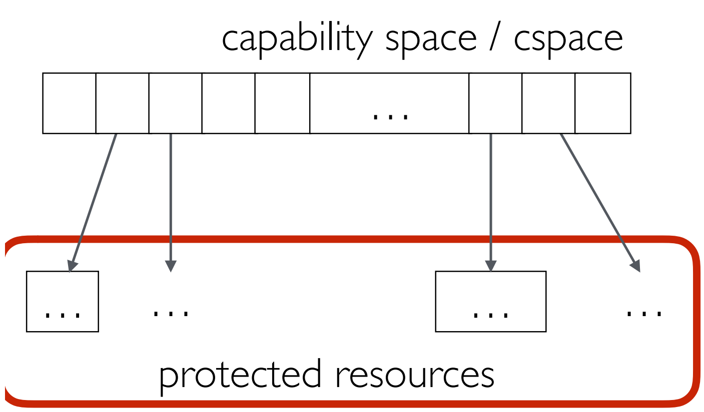
A user process:

- can only access an address in physical memory if there is a corresponding mapping in its page directory/page tables
- accesses memory via virtual addresses, and doesn't know the underlying physical address
- has no direct ability to change the page directory/page tables

Can we replicate this idea?

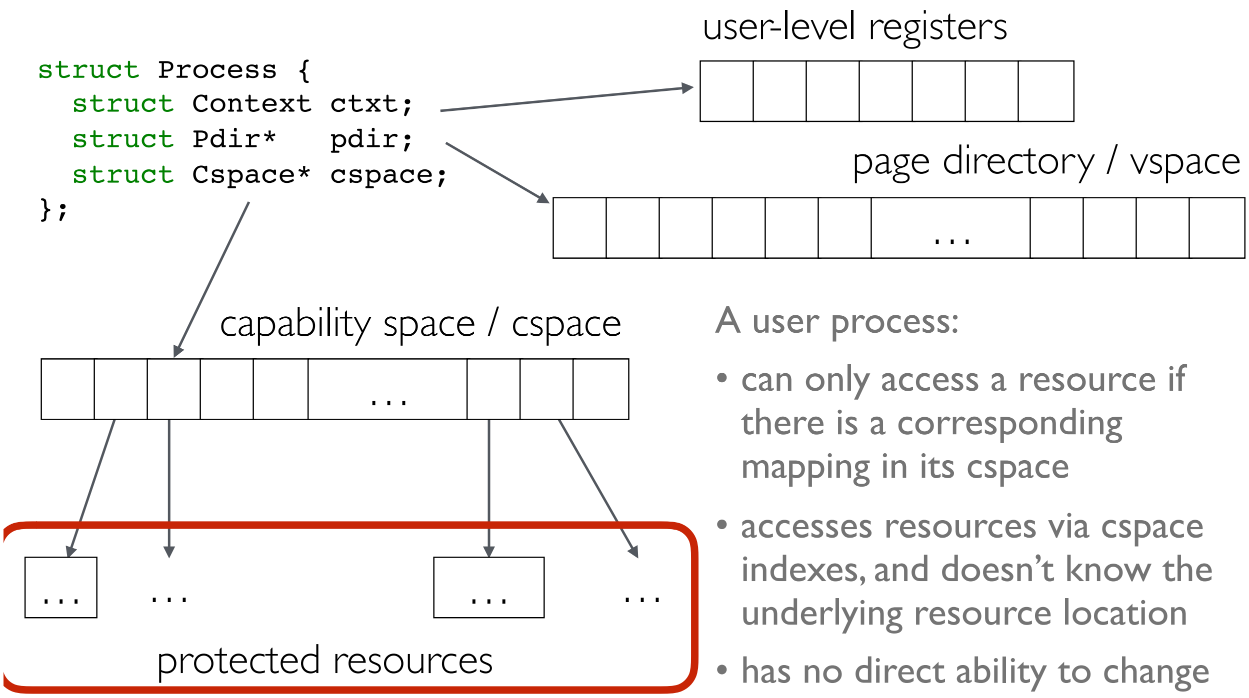
```
struct Process {
  struct Context ctxt;
  struct Pdir*   pdir;
};
```





Can we replicate this idea?

```
struct Process {
  struct Context ctxt;
  struct Pdir*   pdir;
  struct Cspace* cspace;
};
```

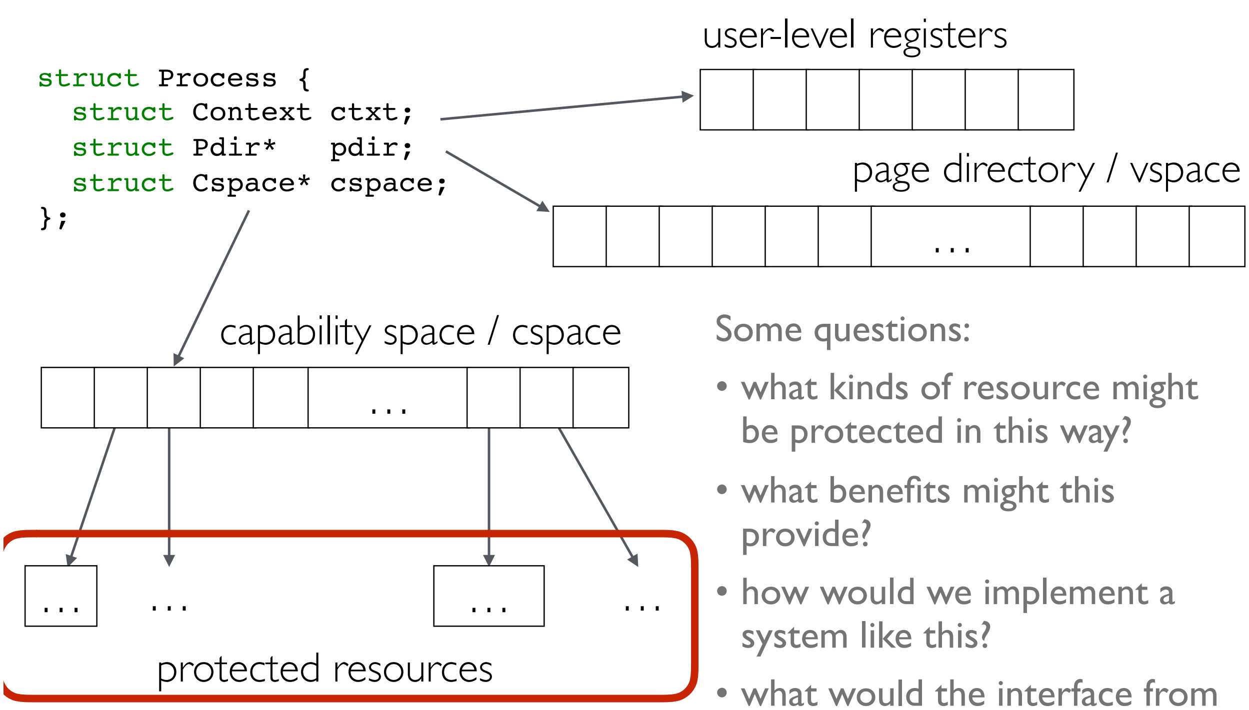


A user process:

- can only access a resource if there is a corresponding mapping in its cspace
- accesses resources via cspace indexes, and doesn't know the underlying resource location
- has no direct ability to change the protected resources

Can we replicate this idea?

```
struct Process {
  struct Context ctxt;
  struct Pdir*   pdir;
  struct Cspace* cspace;
};
```



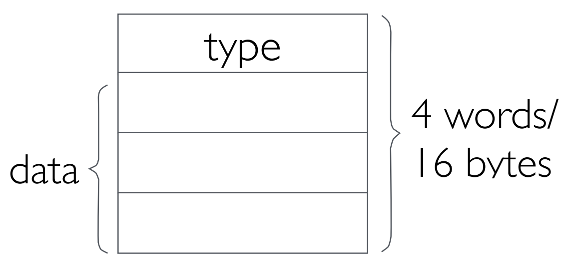
Some questions:

- what kinds of resource might be protected in this way?
- what benefits might this provide?
- how would we implement a system like this?
- what would the interface from user level programs look like?

A “Simple” Implementation

struct Cap and the Null Capability

```
struct Cap {
  enum Captype type;
  unsigned     data[3];
};
```




```
enum Captype {
  NullCap = 0,
  ...
};
```

(If necessary, we could “pack” multiple data items into a single word; e.g., a Captype could fit in ~5 bits; a pointer to a page directory only requires 20 bits; etc…)

```
static inline unsigned isNullCap(struct Cap* cap) {
  return cap->type==NullCap;
}
```

test

```
static inline void nullCap(struct Cap* cap) {
  cap->type = NullCap;
}
```

set

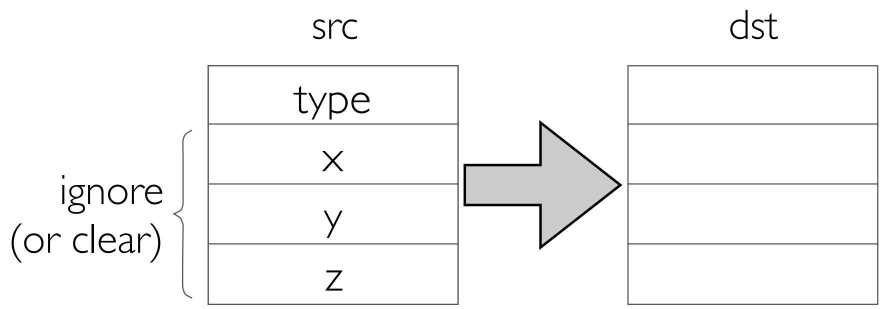
Moving a capability

```
static inline
void moveCap(struct Cap* src, struct Cap* dst, unsigned copy) {
  dst->type    = src->type;
  dst->data[0] = src->data[0];
  dst->data[1] = src->data[1];
  dst->data[2] = src->data[2];
  if (copy==0) {
    nullCap(src);
  }
}
```

0 = move
1 = copy

transfer components

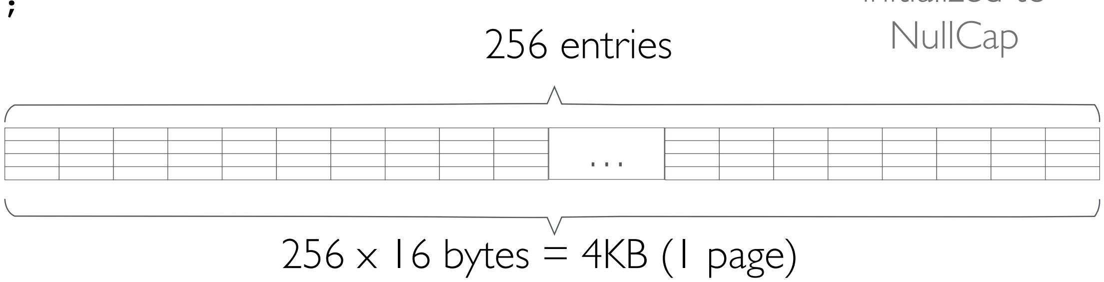
if this is a move, then clear the source



Capability spaces (struct Cspace)

```
#define CSPACEBITS 8
#define CSPACESIZE (1 << CSPACEBITS)

struct Cspace {
  struct Cap caps[CSPACESIZE];
};
```

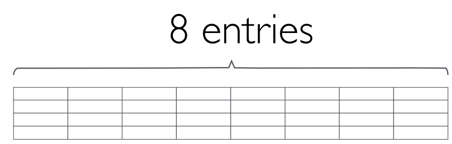
All entries initialized to NullCap




```
typedef unsigned Cptr;      // identifies  a slot in a cspace

static inline Cptr cptr(unsigned w) {
  return maskTo(w, CSPACEBITS);
}
```

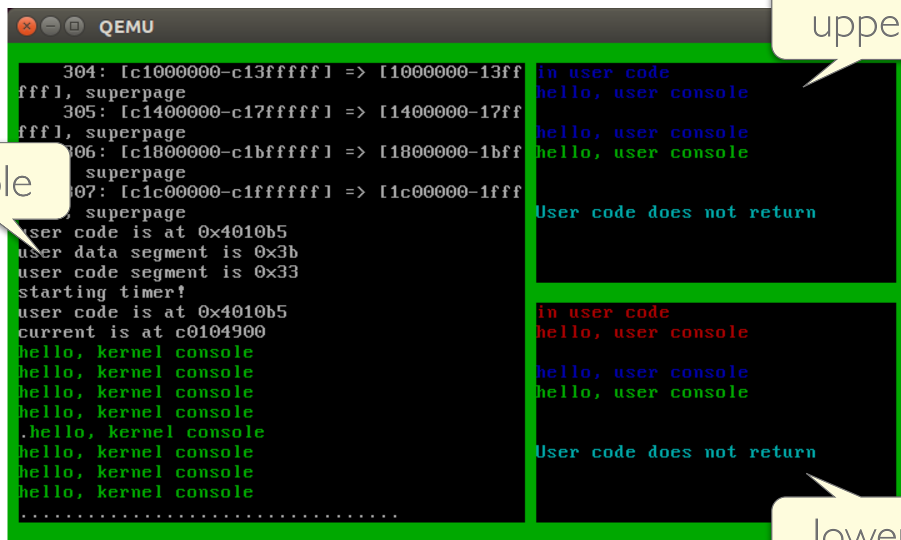
Capability spaces, in practice

- Capabilities and capability spaces are stored in kernel memory, and **must not** be accessible from user-level code
- In practice:
  - We may not need 256 slots for simple applications
  - We may need a lot more than 256 slots for complex applications
  - We could use variable-length nodes and a multi-level tree structure to represent a cspace as a sparse array (much like a page directory/page table structure)
- To simplify this presentation:
  - I'll typically draw a cspace as:



A First Application

What shall we protect today?



The (unprotected) kputc system call

```
void kputc_imp() {
  struct Context* ctxt = &current->ctxt;

  putchar(ctxt->regs.eax);

  ctxt->regs.eax = 0;

  switchToUser(ctxt);
}
```

find registers

output character in console window

set return code

return to caller

Any user program can write to the console window by calling kputc()

Can we limit access to programs that have been given an explicit capability for console access?

Steps to implement a new capability type

1. Define a new capability type
   - pick a new capability type code, determine structure, and add test/set methods (in kernel/caps.h)
   - for debugging purposes, update showCap() to display capability (in kernel/caps.c)
2. Rewrite system call(s) to use the new capabilities (in kernel/syscalls.c)
3. Install capabilities in the appropriate user-level capability spaces (in kernel/kernel.c)
4. Add user-level interface/system calls (in user/syscalls.h, user/userlib.s)

1. Define a console access capability type

```
enum Captype { …, ConsoleCap = 1, … };

struct ConsoleCap {
  enum Captype type;      // ConsoleCap
  unsigned      unused[3];
};

static inline struct ConsoleCap* isConsoleCap(struct Cap* cap) {
  return (cap->type==ConsoleCap) ? (struct ConsoleCap*)cap : 0;
}

static inline void consoleCap(struct Cap* cap) {
  struct ConsoleCap* ccap = (struct ConsoleCap*)cap;
  printf("Setting console cap at %x\n", ccap);
  ccap->type = ConsoleCap;
}
```

capability type

capability structure

capability test

capability set

2. A capability-protected version of kputc

Inputs:

- eax: character to output
- ecx: console capability

Output:

- eax: "thread id"

for illustration only: not really appropriate for kputc :-)

```
void kputc_imp() {
  struct Context*    ctxt = &current->ctxt;
  struct ConsoleCap* cap  = isConsoleCap(current->cspace->caps +
                                 cptr(ctxt->regs.ecx));

  if (cap) {
    putchar(ctxt->regs.eax);
    ctxt->regs.eax = (unsigned)current;
  } else {
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```

requires capability

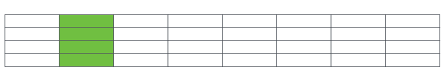
capability lookup

current provides a unique token for the process, but there is no user-level access to that address

3. Install capabilities

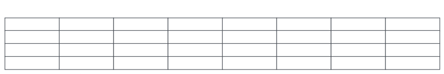
```
// Configure proc[0]:
initProcess(proc+0, hdrs[7], hdrs[8], hdrs[9]);
consoleCap(proc[0].cspace->caps + 1);
showCspace(proc[0].cspace);
```

console access



```
Capability space at c040b000
  0x01 ==> ConsoleCap
1 slot(s) in use
```

```
// Configure proc[1]:
initProcess(proc+1, hdrs[7], hdrs[8], hdrs[9]);
showCspace(proc[1].cspace);
```



```
Capability space at c0109000
0 slot(s) in use
```

no console access

4. User level access to the console

```
#define CONSOLE  1
extern unsigned kputc(unsigned cap, unsigned ch);
```

user/syscalls.h

```
void kputs(unsigned cap, char* s) {
  while (*s) {
    kputc(cap, *s++);
  }
}

void cmain() {
  unsigned myid = kputc(CONSOLE, '!');
  printf("My process id is %x\n", myid);
  kputs(CONSOLE, "hello, kernel console\n");
}
```

user/user.c

```
        # System call to print a character in the
        # kernel's window:
        #
        #     | retn | cap  | ch   |
        #     | 0    | 4    | 8    |
        .globl  kputc
kputc:  movl    4(%esp), %ecx
        movl    8(%esp), %eax
        int     $128
        ret
```

user/userlib.s

Protected access to the console

- A console access capability is a "token" that grants the holder the ability to write output on the console window
- User level processes have access to the console … but only if they have an appropriate capability installed in their cspace
- The kernel can add or remove access at any time
- No capability, no access …
- … and no way for a user-level process to "fake" a capability

- But how can a user distinguish kernel output in the console window from output produced by a capability-holding user-level process?

Badged Capabilities:
Identity and Permissions

A badged capability type for console access

```
enum Captype { …, ConsoleCap = 1, … };

struct ConsoleCap {
  enum Captype type;      // ConsoleCap
  unsigned     attr;      // attribute for display
  unsigned     unused[2];
};

static inline struct ConsoleCap* isConsoleCap(struct Cap* cap) {
  return (cap->type==ConsoleCap) ? (struct ConsoleCap*)cap : 0;
}

static inline void consoleCap(struct Cap* cap, unsigned attr) {
  struct ConsoleCap* ccap = (struct ConsoleCap*)cap;
  printf("Setting console cap at %x\n", ccap);
  ccap->type = ConsoleCap;
  ccap->attr = attr;
}
```

video attribute

Using the attribute badge

```
void kputc_imp() {
  struct Context*    ctxt = &current->ctxt;
  struct ConsoleCap* cap  = isConsoleCap(current->cspace->caps +
                                 cptr(ctxt->regs.ecx));
  if (cap) {
    setAttr(cap->attr);
    putchar(ctxt->regs.eax);
    setAttr(7);
    ctxt->regs.eax = (unsigned)current;
  } else {
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```

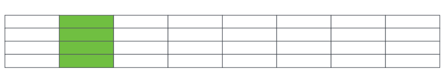
set video attribute

restore video attribute

Setting the video attribute

```
// Configure proc[0]:
initProcess(proc+0, hdrs[7], hdrs[8], hdrs[9]);
consoleCap(proc[0].cspace->caps + 1, 0x2e);
showCspace(proc[0].cspace);
```

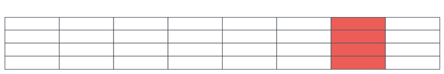
PSU Green



```
Capability space at c040b000
  0x01 ==> ConsoleCap, attr=2e
1 slot(s) in use
```

```
// Configure proc[1]:
initProcess(proc+1, hdrs[7], hdrs[8], hdrs[9]);
consoleCap(proc[1].cspace->caps + 6, 4);
showCspace(proc[1].cspace);
```

Red



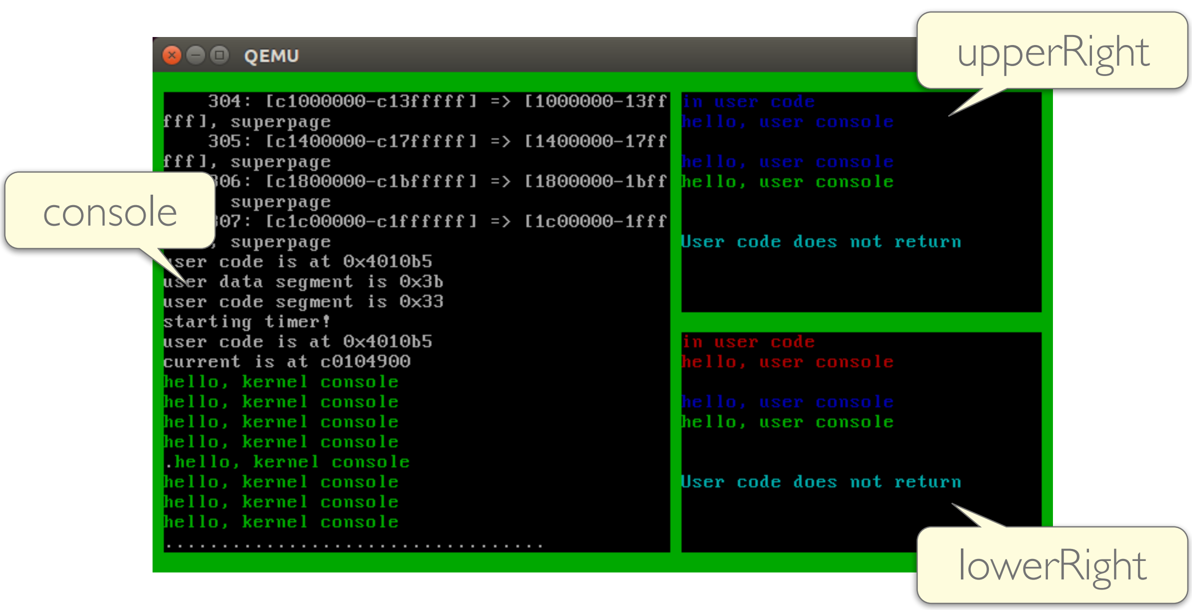
```
Capability space at c0109000
  0x06 ==> ConsoleCap, attr=4
1 slot(s) in use
```

Prevents user code from "spoofing" kernel output!

Badged capabilities

- A badged capability stores extra information in the capability
- Different capabilities for an object may have different badges
- There is no (a priori) way for the holder of a capability to determine or change the value of its “badge”
- A common practical application scenario:
  - Server process receives requests from clients via a read-only capability to a communication channel
  - Clients hold write-only capabilities to the same communication channel, each “badged” with a unique identifier so that the server can distinguish between them

Capability permissions/rights



Capabilities to Windows

```
enum Captype { …, WindowCap = 2, … };

struct WindowCap {
  enum Captype   type;       // WindowCap
  struct Window* window;     // Pointer to the window
  unsigned       perms;      // Perms (CAN_{cls,setAttr,putchar})
  unsigned       unused[1];
};

#define CAN_cls       0x4    // confers permission to clear screen
#define CAN_setAttr   0x2    // confers permission to set attribute
#define CAN_putchar   0x1    // confers permission to putchar
```

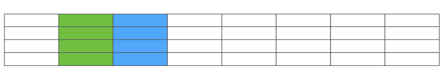
protected resource

permissions (badge)

permission flags

Installing a capability to a Window

```
// Configure proc[0]:
initProcess(proc+0, hdrs[7], hdrs[8], hdrs[9]);
consoleCap(proc[0].cspace->caps + 1, 4);
windowCap(proc[0].cspace->caps  + 2,
          &upperRight,
          /*CAN_cls|*/CAN_setAttr|CAN_putchar);
showCspace(proc[0].cspace);
```



```
Capability space at c040b000
  0x01 ==> ConsoleCap, attr=4
  0x02 ==> WindowCap, window=c01069c0, perms=3
2 slot(s) in use
```

System calls using Window capabilities

```
struct WindowCap* getWindowCap() {
  return isWindowCap(current->cspace->caps
                    + cptr(current->ctxt.regs.ecx));
}

void capputchar_imp() {
  struct WindowCap* wcap = getWindowCap();

  if (wcap &&
      (wcap->perms & CAN_putchar)) {

    wputchar(wcap->window, current->ctxt.regs.eax);

  }

  switchToUser(&current->ctxt);
}
```

lookup

protected object

permission check

underlying operation

The capio library

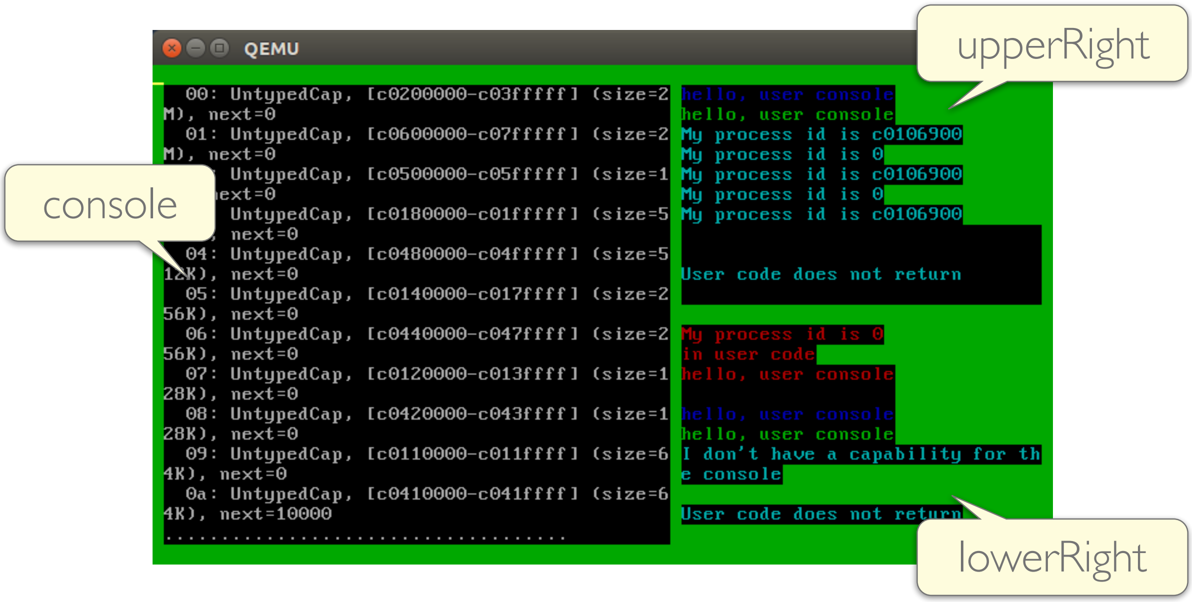
```
/*-------------------------------------------------------------------------
 * capio.h: A version of the simpleio library using capabilities.
 * Mark P Jones, Portland State University
 *-----------------------------------------------------------------------*/
#ifndef CAPIO_H
#define CAPIO_H

// General operations that allow us to specify a window capability.
extern void capsetAttr(unsigned cap, int a);
extern void capcls(unsigned cap);
extern void capputchar(unsigned cap, int c);
extern void capputs(unsigned cap, char* s);
extern void capprintf(unsigned cap, const char *format, ...);

// By default, we assume that our window capability is in slot 2.
#define DEFAULT_WINDOW_CAP   2

#define setAttr(a)          capsetAttr(DEFAULT_WINDOW_CAP, a)
#define cls()               capcls(DEFAULT_WINDOW_CAP)
#define putchar(c)          capputchar(DEFAULT_WINDOW_CAP, c)
#define puts(s)             capputs(DEFAULT_WINDOW_CAP, s)
#define printf(args...)     capprintf(DEFAULT_WINDOW_CAP, args)

#endif
/*-----------------------------------------------------------------------*/
```

C idiom to avoid repeated includes

general form

“easy” defaults

You have no “right” to clear the screen!



Organizing Capability Spaces

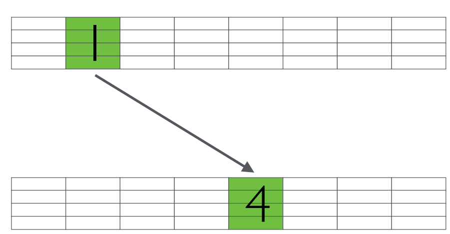
Capability space layout

- We're used to having certain memory regions at known addresses:
  - Video RAM at 0xb8000
  - KERNEL_SPACE at 0xc000_0000
  - …
- We're developing a "default" layout for capability spaces:
  - Console access in slot 1
  - Window access in slot 2
  - …
- Should user level programs have the ability to rearrange/remap their capability space?

A move/copy capability system call

```
void capmove_imp() {
  struct Context* ctxt = &current->ctxt;
  struct Cap*     caps = current->cspace->caps;
  struct Cap*     src  = caps + cptr(ctxt->regs.esi);
  struct Cap*     dst  = caps + cptr(ctxt->regs.edi);
  if (isNullCap(dst) && !isNullCap(src)) {
    printf("  Before:\n");
    showCspace(current->cspace);
    moveCap(src, dst, ctxt->regs.eax);
    printf("  After:\n");
    showCspace(current->cspace);
    ctxt->regs.eax = 1;
  } else {
    printf("  Invalid capmove\n");
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```

debugging output



Wait a minute! Shouldn't this kind of operation be protected using capabilities?

Capabilities to capability spaces

This should be looking quite familiar by now!

```
enum Captype { …, CspaceCap = 3, … };

struct CspaceCap {
  enum Captype   type;      // CspaceCap
  struct Cspace* cspace;    // Pointer to the cspace
  unsigned       unused[2];
};

static inline struct Cspace* isCspaceCap(struct Cap* cap) {
  return (cap->type==CspaceCap) ? ((struct CspaceCap*)cap)->cspace : 0;
}
```

capability test

```
static inline
struct CspaceCap* cspaceCap(struct Cap* cap, struct Cspace* cspace) {
  struct CspaceCap* ccap = (struct CspaceCap*)cap;
  ccap->type   = CspaceCap;
  ccap->cspace = cspace;
  return ccap;
}
```

capability set

Capability slot references

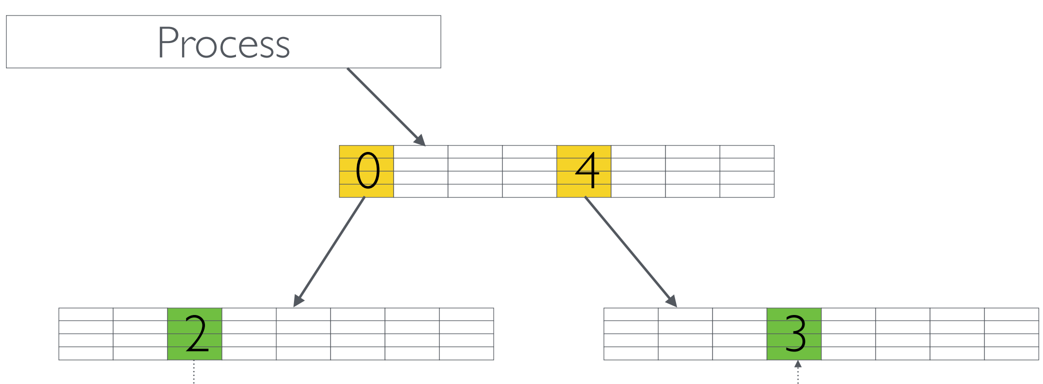
- The src and dest arguments contain 4 bytes each

-	-	index	cptr

index to a CspaceCap in the cspace of the calling process

offset within that cspace

- Example: move from 0x00_02 to 0x04_03:



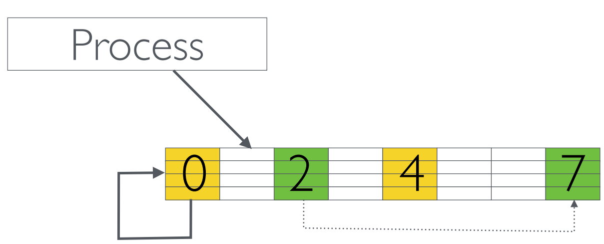
Capability slot lookup

```
static inline Cptr index(unsigned w) {
  return maskTo(w >> CSPACEBITS, CSPACEBITS);
}

struct Cap* getCap(unsigned slot) {
  struct Cspace* cspace = isCspaceCap(current->cspace->caps
                                 + index(slot));
  return cspace ? (cspace->caps + cptr(slot)) : 0;
}

void capmove_imp() {
  struct Context* ctxt = &current->ctxt;
  struct Cap*     src  = getCap(ctxt->regs.esi);
  struct Cap*     dst  = getCap(ctxt->regs.edi);
  if ((dst && src && isNullCap(dst) && !isNullCap(src))) {
    moveCap(src, dst, ctxt->regs.eax);
    ctxt->regs.eax = 1;
  } else {
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```

But now: how can a process change the capabilities in its own cspace?

Slot zero

- A process can have access to its own cspace if, and only if it has a capability to its cspace
- Slot zero is a convenient place to store this capability
- Example: move from 0x00_02 to 0x00_07 (same as 2 to 7):



- The kernel can create a loop like this using:

```
static inline
void cspaceLoop(struct Cspace* cspace, unsigned w) {
   cspaceCap(cspace->caps + w, cspace);
}
```

What have we accomplished?

- Controlled access to cspace objects
- For processes that have the slot zero capability:
  - the ability to reorganize the entries in the process' cspace using simple slot numbers
- For all processes:
  - the ability to manipulate and move entries between multiple cspaces, given the necessary capabilities
  - the ability to access and use more than 256 capabilities at a time by using multiple cspaces
- But how can a process ever get access to multiple cspaces?

Memory Allocation: Using Capabilities for Resource Management

A system call to extend an address space

- **Problem**: a user level process needs more memory
- **Solution**: the process decides where it wants the memory to be added, and then asks the kernel to map an unused page of memory at that address
- **Implementation**:

```
void kmapPage_imp() {
▶ struct Context* ctxt = &current->ctxt;
  unsigned         addr = ctxt->regs.esi;
  unsigned*        page;
  if (!isMapped(current->pdir, addr) && (page=allocPage())) {
    mapPage(current->pdir, addr, toPhys(page));
    ctxt->regs.eax = 1;
  } else {
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```

This isn't actually correct: we'll see why soon ...

Example use:

- Program:

```
unsigned stomp = 0x700000;
for (int j=0; j<8; j++) {
  kmapPage(stomp);
  *((unsigned*)stomp) = stomp;
  stomp += (1<<12);
}
```

write to new location

- Resulting: page directory/page table structure:

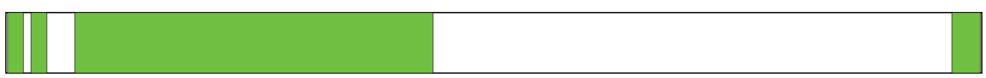
```
Page directory at c040c000
  [400000-7fffff] => page table at c040e000 (physical 40e000):
    0: [400000-400fff] => [40d000-40dfff] page
    1: [401000-401fff] => [40f000-40ffff] page
    2: [402000-402fff] => [108000-108fff] page
    300: [700000-700fff] => [10d000-10dfff] page
    301: [701000-701fff] => [10e000-10efff] page
    302: [702000-702fff] => [10f000-10ffff] page
    303: [703000-703fff] => [410000-410fff] page
    304: [704000-704fff] => [411000-411fff] page
    305: [705000-705fff] => [412000-412fff] page
    306: [706000-706fff] => [41b000-41bfff] page
    307: [707000-707fff] => [41c000-41cfff] page
  300: [c0000000-c03fffff] => [0-3fffff], superpage
  301: [c0400000-c07fffff] => [400000-7fffff], superpage
  302: [c0800000-c0bfffff] => [800000-bfffff], superpage
  303: [c0c00000-c0ffffff] => [c00000-ffffff], superpage
  304: [c1000000-c13fffff] => [1000000-13fffff], superpage
  305: [c1400000-c17fffff] => [1400000-17fffff], superpage
  306: [c1800000-c1bfffff] => [1800000-1bfffff], superpage
  307: [c1c00000-c1ffffff] => [1c00000-1ffffff], superpage
```

What's wrong with this?

- No protection against "denial of service" attacks (intentional or otherwise):
  - There is nothing to prevent one process from allocating all of the available memory, or even just enough memory to prevent another process from doing useful work
- Requires a kernel-based memory allocator:
  - Complicates the kernel …
  - Works against the microkernel philosophy of providing mechanisms but otherwise remaining "policy free"
- Ideally, the kernel would perform initial allocation of memory at boot time, but then delegate all subsequent allocation to user-level processes

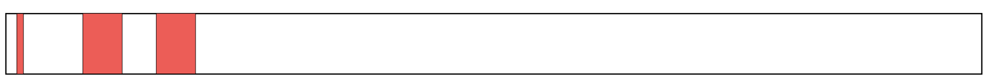
Back to boot time …

mmap



(installed physical memory)

- hdrs

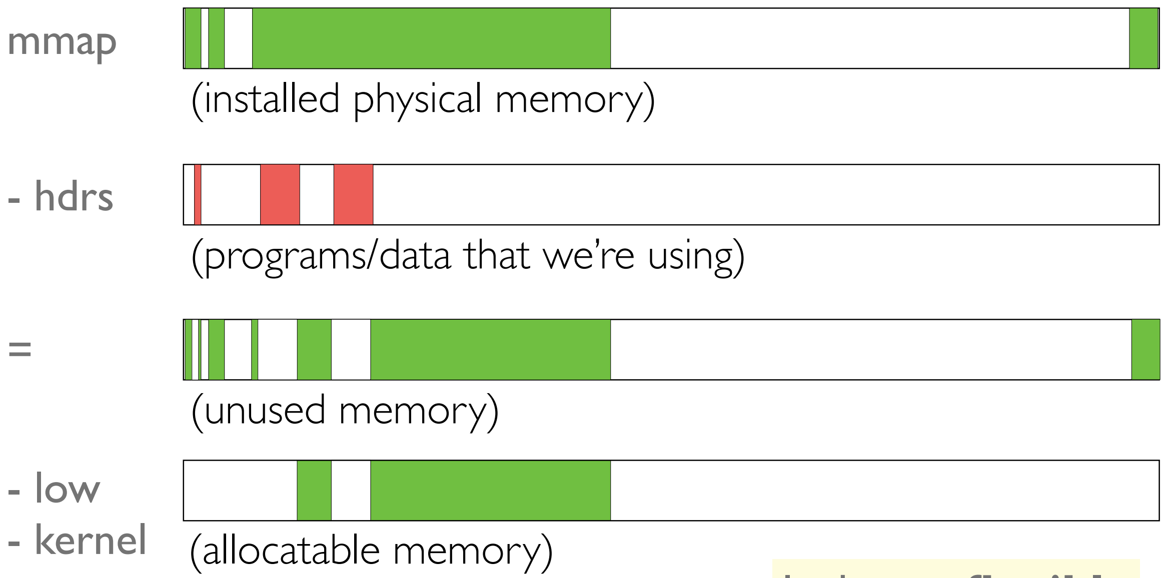


(programs/data that we're using)

Example

```
Headers:
 header[0]: [1000-3fff], entry ffffffff
 header[1]: [100000-104d63], entry 100000
 header[2]: [400000-40210b], entry 4010b5
Memory map:
 mmap[0]: [0-9fbff]
 mmap[1]: [9fc00-9ffff]
 mmap[2]: [f0000-fffff]
 mmap[3]: [100000-1ffdfff]
 mmap[4]: [1ffe000-1ffffff]
 mmap[5]: [fffc0000-ffffffff]
```

Back to boot time …



Example
```
[108000-3fffff] size=3040K
[403000-1ffffff] size=28660K
```

Is there a **flexible** way to manage this memory?

Splitting memory into flexpages

```
[108000-3fffff] size=3040K
[403000-1ffffff] size=28660K
```

```
Flexpages for [0x00108000-0x003fffff]:
  c0108000 (15, 32K)
  c0110000 (16, 64K)
  c0120000 (17, 128K)
  c0140000 (18, 256K)
  c0180000 (19, 512K)
  c0200000 (21, 2M)

Flexpages for [0x00403000-0x01ffffff]:
  c0403000 (12, 4K)
  c0404000 (14, 16K)
  c0408000 (15, 32K)
  c0410000 (16, 64K)
  c0420000 (17, 128K)
  c0440000 (18, 256K)
  c0480000 (19, 512K)
  c0500000 (20, 1M)
  c0600000 (21, 2M)
  c0800000 (23, 8M)
  c1000000 (24, 16M)
```

Splitting memory into flexpages

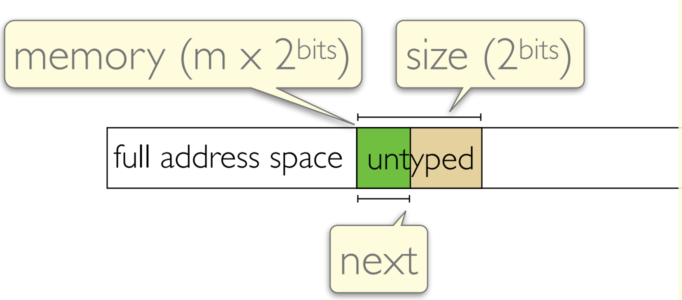
```
[108000-3fffff] size=3040K
[403000-1ffffff] size=28660K
```

sorted (largest flexpages first)

```
Available untyped(s) [17]
  00: [c1000000-c1ffffff] (size=16M)
  01: [c0800000-c0ffffff] (size=8M)
  02: [c0200000-c03fffff] (size=2M)
  03: [c0600000-c07fffff] (size=2M)
  04: [c0500000-c05fffff] (size=1M)
  05: [c0180000-c01fffff] (size=512K)
  06: [c0480000-c04fffff] (size=512K)
  07: [c0140000-c017ffff] (size=256K)
  08: [c0440000-c047ffff] (size=256K)
  09: [c0120000-c013ffff] (size=128K)
  0a: [c0420000-c043ffff] (size=128K)
  0b: [c0110000-c011ffff] (size=64K)
  0c: [c0410000-c041ffff] (size=64K)
  0d: [c0108000-c010ffff] (size=32K)
  0e: [c0408000-c040ffff] (size=32K)
  0f: [c0404000-c0407fff] (size=16K)
  10: [c0403000-c0403fff] (size=4K)
```


Capabilities to Untyped memory

```
enum Captype { …, UntypedCap = 4, … };

struct UntypedCap {
  enum Captype type;   // UntypedCap
  void*        memory;// pointer to an fpage of size bits
  unsigned     bits;   // log2 of size in bytes
  unsigned     next;   // offset to next free location within fpage
};
```



- Untyped memory objects represent pools of allocatable memory
- A capability to untyped memory confers the ability to allocate from that area

Allocating from untyped memory

Strict left to right allocation, flexpages only, padding as necessary:

Allocating from untyped memory

```
void* alloc(struct UntypedCap* ucap, unsigned bits) {
  unsigned len   = 1<<bits;
  unsigned mask  = len-1;
  unsigned first = (ucap->next + mask) & ~mask;
  unsigned last  = first + mask;

  if (ucap->next<=first && last<=((1<<ucap->bits)-1)) {
    unsigned* object = (unsigned*)(ucap->memory + first);
    for (unsigned i=0; i<bytesToWords(len); ++i) {
      object[i] = 0;
    }

    ucap->next = last+1;

    return (void*)object;
  }

  return 0; // Allocation failed: not enough room
}
```

find addresses of first and last bytes of new object

zero memory for new object

update capability

return pointer to new object

Complication: restrictions on copying

```
void capmove_imp() {
  struct Context* ctxt = &current->ctxt;
  struct Cap*     src  = getCap(ctxt->regs.esi);
  struct Cap*     dst  = getCap(ctxt->regs.edi);
  unsigned        copy = ctxt->regs.eax;
 if ((dst && src && isNullCap(dst) && !isNullCap(src)) &&
      (!copy || src->type!=UntypedCap)) {
    moveCap(src, dst, ctxt->regs.eax);
    ctxt->regs.eax = 1;
  } else {
    printf("  Invalid capmove\n");
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```


Complication: restrictions on copying

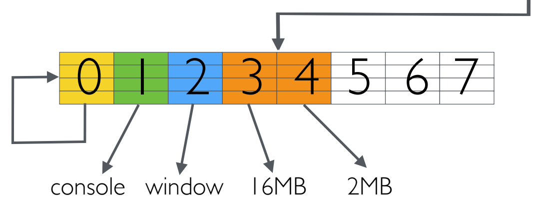
```
void capmove_imp() {
  struct Context* ctxt = &current->ctxt;
  struct Cap*     src  = getCap(ctxt->regs.esi);
  struct Cap*     dst  = getCap(ctxt->regs.edi);
  unsigned        copy = ctxt->regs.eax;
 if ((dst && src && isNullCap(dst) && !isNullCap(src)) &&
      (!copy || src->type!=UntypedCap)) {
    moveCap(src, dst, ctxt->regs.eax);
    ctxt->regs.eax = 1;
  } else {
    printf("  Invalid capmove\n");
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```

we **MUST NOT** allow duplication of a capability to untyped memory!

Overall strategy

- At boot time:
  - partition unallocated memory into a collection of untyped memory areas
  - allocate individual pages from the end of the list of untyped memory areas
  - donate remaining untyped memory to user-level processes
- User-level processes are responsible for all subsequent allocation decisions

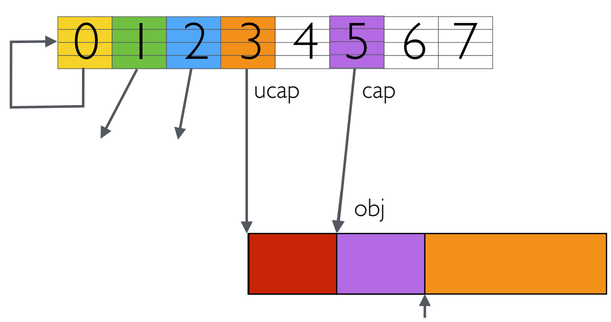
```
Available untyped(s) [17]
  00: [c1000000-c1ffffff] (size=16M)
  01: [c0800000-c0ffffff] (size=8M)
  02: [c0200000-c03fffff] (size=2M)
  03: [c0600000-c07fffff] (size=2M)
  04: [c0500000-c05fffff] (size=1M)
  05: [c0180000-c01fffff] (size=512K)
  06: [c0480000-c04fffff] (size=512K)
  07: [c0140000-c017ffff] (size=256K)
  08: [c0440000-c047ffff] (size=256K)
  09: [c0120000-c013ffff] (size=128K)
  0a: [c0420000-c043ffff] (size=128K)
  0b: [c0110000-c011ffff] (size=64K)
  0c: [c0410000-c041ffff] (size=64K)
  0d: [c0108000-c010ffff] (size=32K)
  0e: [c0408000-c040ffff] (size=32K)
  0f: [c0404000-c0407fff] (size=16K)
  10: [c0403000-c0403fff] (size=4K)
```



Example: system call to allocate a cspace

ecx holds cap for untyped memory

```
void allocCspace_imp() {
  struct Context*     ctxt = &current->ctxt;
  struct UntypedCap*  ucap = getUntypedCap();
  struct Cap*         cap  = getCap(ctxt->regs.edi);
  void*               obj;

  if (ucap &&                              // valid untyped capability

      cap  && isNullCap(cap) &&            // empty destination slot

      (obj=alloc(ucap, PAGESIZE))) {       // object allocation succeeds

    cspaceCap(cap, (struct Cspace*)obj);

    ctxt->regs.eax = 1;
  } else {
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```



Example: allocating untyped memory

```
void allocUntyped_imp() {
  struct Context*     ctxt = &current->ctxt;
  struct UntypedCap* ucap = getUntypedCap();
  struct Cap*         cap  = getCap(ctxt->regs.edi);
  unsigned            bits = ctxt->regs.eax;
  void*               obj;
  printf("allocUntyped: bits %d from ucap=%x, slot=%x\n",
           bits, ucap, cap);
  if (ucap &&                         // valid untyped capability
      cap  && isNullCap(cap) &&       // empty destination slot
      validUntypedSize(bits) &&       // bit size in legal range
      (obj=alloc(ucap, bits))) {      // object allocation succeeds
    untypedCap(cap, obj, bits);
    ctxt->regs.eax = 1;
  } else {
    ctxt->regs.eax = 0;
  }
  switchToUser(ctxt);
}
```

It would be nice if there was a single system call that could allocate multiple types of objects ... (retype)

No dynamic allocation in the kernel

- Once it has been initialized, the kernel must not allocate any memory on behalf of user level processes
  - This is a key feature of seL4: it simplifies the kernel and also prevents memory allocation denial of service attacks
  - Instead, any system call that might need memory for a new kernel data structure will require a capability to untyped memory as an input
- Concretely, there must not be any calls to `allocPage()` in code that is used after the kernel is initialized
  - This includes *anything* that depends on `allocPage()`: `allocPdir()`, `mapPage()`, `initProcess()`, etc.
  - This applies to *all* interrupt and system call handlers

Can we enforce this requirement?

- If we are disciplined, understand the restriction, and keep it in mind at all times, then perhaps our code will be ok
- It we don't trust ourselves, we can insert code to check for violations at runtime
  - This has a (small) impact on performance
  - Worse: we might not discover bugs until code is shipped
- Can we use a programming language that:
  - Uses types to indicate that certain procedures/functions cannot be used after initialization?
  - Allows us to check for violations at compile time?
- Examples like this are not uncommon in low-level code (e.g., we must not sleep or block in an interrupt handler)

But how can we implement kmapPage()?

- The original `kmapPage()` system call *might* require allocation of as many as two new pages:
  - one for the page itself, and another for the page table.
- We must expose this level of detail to user-level programs:
  - Two new capability types: `PageCap` for page objects, and `PageTableCap` for page table objects
  - Two new allocator system calls

```
unsigned allocPage(unsigned ucap, unsigned slot);
unsigned allocPageTable(unsigned ucap, unsigned slot);
```

  - Two new mapping system calls

```
unsigned mapPage(unsigned cap, unsigned addr);
unsigned mapPageTable(unsigned cap, unsigned addr);
```

Example

```
allocPage(3,      /*slot*/12);
allocCspace(3,    /*slot*/14);
stomp = 0x80000000;                // Let's allocate a page here
allocPageTable(3, /*slot*/21);     // allocate a page table
mapPageTable(21, stomp);           // map it into the address space
mapPageTable(21, stomp+0x800000); // and again, 8MB further
allocPage(3,     /*slot*/20);
mapPage(20,      stomp);
```

```
Page directory at c0406000
  [400000-7fffff] => page table at c0408000 (physical 408000):
    0: [400000-400fff] => [407000-407fff] page
    1: [401000-401fff] => [409000-409fff] page
    2: [402000-402fff] => [40a000-40afff] page
  [80000000-803fffff] => page table at c1002000 (physical 1002000):
    0: [80000000-80000fff] => [1003000-1003fff] page
  [80800000-80bfffff] => page table at c1002000 (physical 1002000):
    0: [80800000-80800fff] => [1003000-1003fff] page
  ...

Capability space at c040b000
  0x00 ==> CspaceCap, cspace=c040b000
  0x01 ==> ConsoleCap, attr=4
  0x02 ==> WindowCap, window=c01069c0, perms=3
  0x03 ==> UntypedCap, [c1000000-c1ffffff] (size=16M), next=4000
  0x0c ==> PageCap, page=c1000000
  0x0e ==> CspaceCap, cspace=c1001000
  0x14 ==> PageCap, page=c1003000
  0x15 ==> PageTableCap, ptab=c1002000
8 slot(s) in use
```


Example

```
allocPage(3,      /*slot*/12);
allocCspace(3,    /*slot*/14);
stomp = 0x80000000;                // Let's allocate a page here
allocPageTable(3, /*slot*/21);     // allocate a page table
mapPageTable(21, stomp);           // map it into the address space
mapPageTable(21, stomp+0x800000); // and again, 8MB further
allocPage(3,     /*slot*/20);
mapPage(20,      stomp);
```

```
Page directory at c0406000
  [400000-7fffff] => page table at c0408000 (physical 408000):
    0: [400000-400fff] => [407000-407fff] page
    1: [401000-401fff] => [409000-409fff] page
    2: [402000-402fff] => [40a000-40afff] page

Capability space at c040b000
  0x00 ==> CspaceCap, cspace=c040b000
  0x01 ==> ConsoleCap, attr=4
  0x02 ==> WindowCap, window=c01069c0, perms=3
  0x03 ==> UntypedCap, [c1000000-c1ffffff] (size=16M), next=4000
```

What have we accomplished now?

- User-level code:
  - can construct its own address space
  - is responsible for allocating any pages and page tables that it requires for this
  - is limited by the amount of memory it has been assigned via capabilities to untyped memory
- The kernel:
  - ensures validity of mapping operations (no mappings in kernel space, no overlapping mappings, ...)
  - updates the underlying page directory and page table structures as necessary
  - does not perform any dynamic memory allocation!

Advanced feature "wish list"

- Capabilities for page directories:
  - Allow user level code to manage multiple address spaces
- Capability faults:
  - Our system calls report an error code if the requested capability is invalid/does not exist
  - A more flexible strategy is to invoke a "capability fault handler" (analogous to a page fault handler for virt. mem.)
- Capability delegation and revocation
  - How do we find all the copies of a capability if the original is deleted?
- Object deletion:
  - Can we reclaim memory for an object when the last capability for the object is deleted?

Other kinds of capabilities

- Capabilities for Thread Control Blocks
  - likely including system calls to:
    - configure address space, scheduling params, etc.
    - start/suspend new threads
    - read/write thread registers
- Capabilities for "Endpoints":
  - threads read from and write to endpoints to support IPC
  - each endpoint holds a queue of threads that are blocked, waiting for a communication partner
- Capabilities for IO ports (or other hardware features):
  - each capability can provide access to a range of IO ports, with separate permissions for in and out instructions

Summary

- Capabilities support:
  - Fine-grained access control
  - A novel approach to resource management: no dynamic memory allocation in the kernel; shifts responsibility to user level
- The implementation described here is a "toy", but is enough to demonstrate key concepts for a capability-based system
- The seL4 microkernel is a real-world system built around the use of capabilities
- A very powerful and important abstraction: don't be put off by implementation complexities!