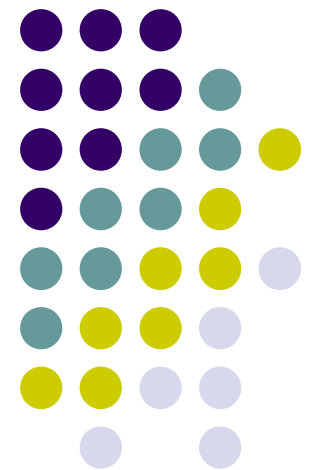


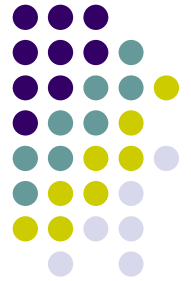
# CS241

# System Programming

---

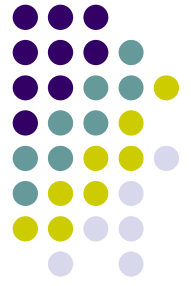
Discussion Section 9  
April 3 – April 6





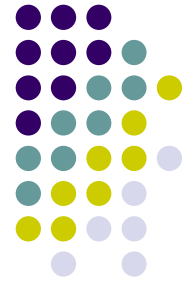
# Outline

- UNIX Security
  - Access Control Lists
  - Capabilities
- Memory Management
  - Fragmentation
  - Storage Placement Algorithms
  - malloc revisited



# POSIX ACL

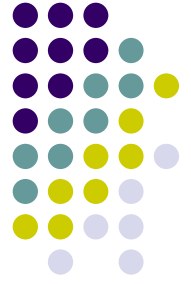
- POSIX standard (POSIX 1003.1e, POSIX 1003.2c)
  - Currently withdrawn
  - Many implementations follow this standard
    - Basic file commands (`cp`, `mv`, `ls`, etc) support ACLs
- ACL Type
  - Default ACL
    - Associated with directories
    - Initial access ACL to objects under the directory
  - Access ACL
    - Associated with objects
    - Initialized when `creat`, `mkdir`, `mknod`, `mkfifo`, or `open` function



# POSIX ACL (continued)

- ACL Format
  - ACL Entry
    - Entry Tag Type
      - ACL\_USER\_OBJ, ACL\_USER, ACL\_GROUP\_OBJ, ACL\_GROUP, ACL\_MASK, ACL\_OTHER
    - Entry Tag Qualifier (optional)
    - Set of permissions
  - Example (long text form)

```
user::rw-          # owner
user:lisa:rw-      # named user
group::r--         # owning group
group: toolies:rw- # named group
mask::r--         # mask
other::r--        # other
```



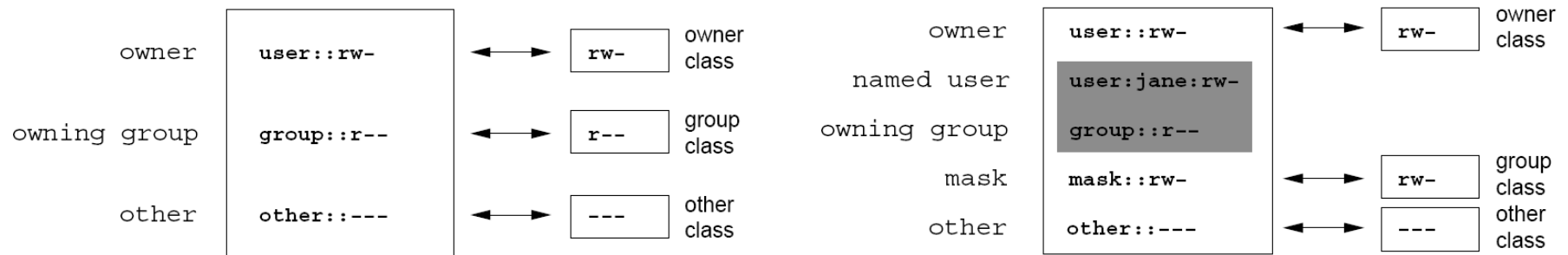
# POSIX ACL (continued)

- Valid ACL
  - Should contain exactly one entry with ACL\_USER\_OBJ, ACL\_GROUP\_OBJ, ACL\_OTHER tag types.
  - ACL\_USER, ACL\_GROUP is optional.
    - Should have ACL\_MASK if ACL has one of these
  - User ID qualifiers or group ID qualifiers must be unique among ACL\_USER or ACL\_GROUP.

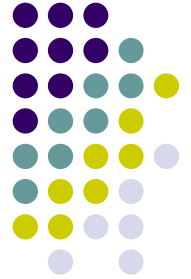


# ACL and file permission bits

- ACL permissions are a superset of permissions by file permission bits
  - ACL\_USER\_OBJ corresponds to the file owner
  - ACL\_GROUP\_OBJ (or ACL\_MASK) corresponds to the file group
  - ACL\_OTHER corresponds to the other class
- Modification of one results in modification of the other.



# Manipulating file permission bits

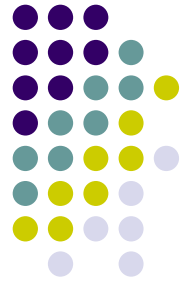


- **Commands**

- `chmod`: changes file permission bits
- `chown`: changes file owner and group
- `chgrp`: changes group ownership

- **Examples**

- `chmod go-w file1`
- `chmod go+rw file1 file2`
- `chmod ugo+rwx file1`
- `chmod 644 file1`
- `chown roger file1 file2`



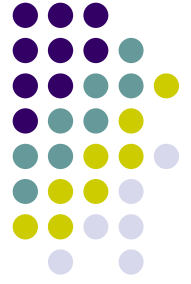
# Access Check Algorithm

- Upon a read, write, or execute request of a file object
  - Case 1: user ID matches the file object owner
    - ACL\_USER\_OBJ entry should contain the requested permission
  - Case 2: user ID matches ACL\_USER qualifier
    - Matching ACL\_USER and ACL\_MASK entry should contain the requested permission
  - Case 3: group ID matches file group or ACL\_GROUP qualifier
    - If ACL contains ACL\_MASK, matching ACL\_GROUP\_OBJ or ACL\_GROUP entry should contain the requested permission
    - Otherwise, ACL\_GROUP\_OBJ entry should contain the requested permission
  - Case 4: Otherwise
    - ACL\_OTHER should contain the requested permission
- In other cases, the access is denied.



# ACL functions

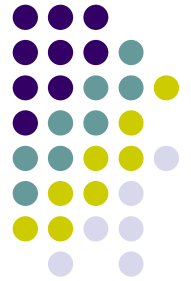
- `#include <sys/acl.h>`
- ACL storage management
  - `acl_dup`, `acl_free`, `acl_init`
- ACL entry manipulation
  - `acl_copy_entry`, `acl_create_entry`, `acl_delete_entry`, `acl_get_entry`, `acl_valid`
  - `acl_add_perm`, `acl_calc_mask`, `acl_clear_perms`, `acl_delete_perm`, `acl_get_permset`, `acl_set_permset`
  - `acl_get_qualifier`, `acl_get_tag_type`, `acl_set_qualifier`, `acl_set_tag_type`
- ACL manipulation on an object
  - `acl_delete_def_file`, `acl_get_fd`, `acl_get_file`, `acl_set_fd`, `acl_set_file`
- ACL format translation
  - `acl_copy_entry`, `acl_copy_ext`, `acl_from_text`, `acl_to_text`, `acl_size`



# POSIX Capabilities

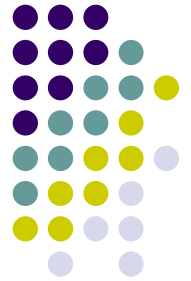
- No standards govern capabilities
  - Linux implementation is based on POSIX 1003.1e
- Somewhat different concept from capability list
- Motivation
  - POSIX capabilities supports fine-grained root privileges
    - Dividing privileges of superuser into distinct units

# POSIX Capabilities (continued)

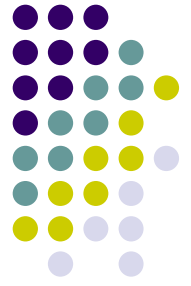


- Process Capabilities
  - Each process has 3 capability sets
    - Effective: capabilities used by the kernel to perform permission checks
    - Permitted: capabilities that the process may assume
    - Inherited: capabilities preserved across an `execve`
  - A child created via `fork` inherits its parent's capability sets

# POSIX Capabilities (continued)

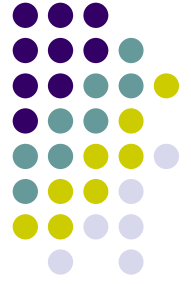


- Current Implementations
  - The kernel checks whether the process has the required capability for all privileged operations
  - The kernel provides system calls to change or retrieve the capability sets of a process
- Future Implementations
  - File system support for attaching capabilities to an executable file, so that a process can gain the capabilities while the file is executed



# Capability functions

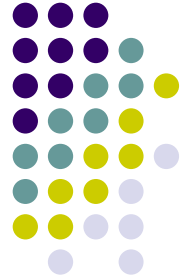
- `#include <sys/capability.h>`
- capability storage management
  - `cap_dup, cap_free, cap_init`
- capability manipulation
  - `cap_get_proc, cap_set_proc`
- capability format translation
  - `cap_copy_int, cap_copy_ext, cap_from_text, cap_to_text, cap_size`



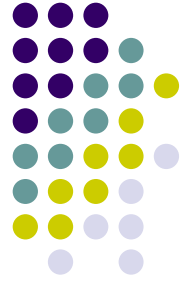
# Fragmentation

- External Fragmentation
  - Free space becomes divided into many small pieces
  - Caused over time by allocating and freeing the storage of different sizes
- Internal Fragmentation
  - Result of reserving space without ever using its part
  - Caused by allocating fixed size of storage

# Storage Placement Algorithms

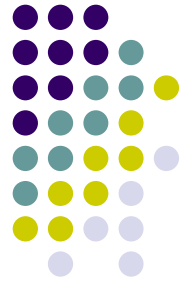


- Best Fit
- First Fit
- Next Fit
- Worst Fit



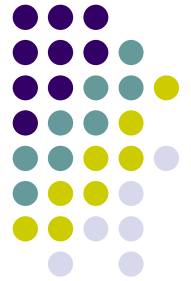
# Exercise

- Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of (a) 12KB, (b) 10KB, (c) 9KB for
  - First Fit?
  - Best Fit?
  - Worst Fit?
  - Next Fit?



# malloc revisited

- Free storage is kept as a list of free blocks
  - Each block contains a size, a pointer to the next block, and the space itself
- When a request for space is made, the free list is scanned until a big-enough block can be found
  - Which storage placement algorithm is used?
- If the block is found, return it and adjust the free list. Otherwise, another large chunk is obtained from the OS and linked into the free list

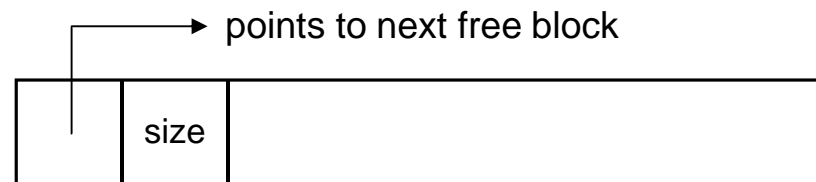


# malloc revisited (continued)

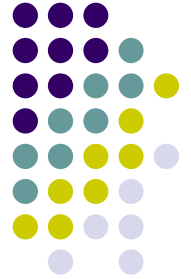
```
typedef long Align;      /* for alignment to long */

union header {          /* block header */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;     /* size of this block */
    } s;
    Align x;               /* force alignment of blocks */
};

typedef union header Header;
```



# malloc revisited (continued)

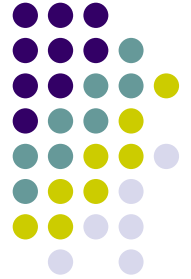


```
static Header base;
static Header *freep = NULL;

void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes + sizeof(Header) - 1) / sizeof(Header) + 1;
    if ((prevp = freep) == NULL) {
        base.s.ptr = freep = prevp = &base;  base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {           /* big enough */
            if (p->s.size == nunits)       /* exact size */
                prevp->s.ptr = p->s.ptr;
            else {                          /* allocate tail-end */
                p->s.size -= nunits;  p += p->s.size;  p->s.size = nunits;
            }
            freep = prevp;
            return (void *) (p+1);
        }
        if (p == freep)                    /* not big enough */
            if ((p = morecore(nunits)) == NULL) /* ask for more space */
                return NULL;
    }
}
```

# malloc revisited (continued)



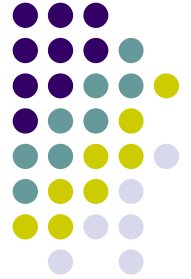
```
#define NALLOC 1024

static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *)-1)
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *) (up+1));
    return freep;
}
```

- `sbrk`: system call that increments the program's data space by the specified size

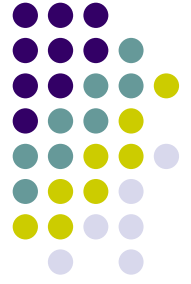
# free revisited



```
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;
    if (bp + bp->s.size == p->s.ptr) {
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

- scans the free list, looking for the place to insert the free block starting at `freep`.



# Summary

- UNIX Security
  - POSIX ACLs
  - POSIX Capabilities
- Memory Management
  - Fragmentation
    - External Fragmentation vs. Internal Fragmentation
  - Storage Placement Algorithms
    - First Fit, Best Fit, Worst Fit, Next Fit
  - malloc revisited