Lecture #14

• Final next week!!
  – Thursday May 4th, 7-9pm WWH 109
  – Exam is closed book/notes; calculators allowed

• Review
  – This is a broad overview. See the Study Guide for more detail of covered material
  – Exam questions will be based on reading and class material

• Study Guide is on the web site

• Xen Project/presentation
  – http://www.cl.cam.ac.uk/Research/SRG/netos/xen/
  – http://xen.sf.net
What is an OS and why do we need it?

An Operating System is:

- A resource allocator and scheduler
- An intermediary between programs and the hardware
- An abstraction
- A multiplexor
- An event manager
Steps in Making a System Call

There are 11 steps in making the system call `read (fd, buffer, nbytes)`
The Process

- User process
- User process
- kernel

Stack:
- text/code
- data/heap
- gap
- stack

Kernel stack: 0xFFFFFFFF
User stack: 0x0
The Stack Pointer

BEFORE: push eax
The Stack Pointer

AFTER: push eax
Invoking a System Call: execve

glibc-2.3/sysdeps/unix/sysv/linux/execve.c

int
execve (file, argv, envp)
    const char *file;
    char *const argv[];
    char *const envp[];
{
    ...

LOADARGS_3
movl 3, %eax
int 0x80
RESTOREARGS_3
}
How do System Calls Really Work: simplified

```c
system_call( int sys_call_num, sys_call_args )

    SAVE_ALL; /* save process registers */

    if (sys_call_num >= NR_syscalls)
        errno = -ENOSYS;
    else {
        errno=(*sys_call_table[sys_call_num])(sys_call_args);
    }

    /* skip code to handle nesting */

    if (need_resched)
        schedule(); /* returns when *this* process is reactivated */

    ...

    RESTORE_ALL; /* restore process registers */
```
Chapter 2: Processes & Threads
Processes

- A program in execution
- Every process has an address space
- CPU is timeshared between processes
- States: ready, running, blocked, ...
- Processes can communicate (IPC, signals)
- OS maintains a process table
The Process

- Kernel
- User process
- User process
- Kernel stack
- Stack
- Gap
- Data/heap
- Text/code

0x0
0xFFFFFFFF
kernel stack
# Fields of a Process Table Entry

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Threads = lightweight processes

Multithreading: allowing multiple threads in the same process
# Threads: shared & private data

<table>
<thead>
<tr>
<th><strong>Per process items</strong></th>
<th><strong>Per thread items</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>
Advantages of Threads

• Space saving
• Switching between threads less expensive
  – Because threads in the same process share so much state, switching between them is much less expensive than switching between separate processes.
Mutual Exclusion

Definition: A way to ensure that if one process is using a shared variable (data), then other processes will be excluded from using the same variable.
IPC: Race Condition

Process A

Process B

Spooler directory

4  abc
5  prog.c
6  prog.n
7

out = 4

in = 7
Race Condition

```
next_free_slot = in
in = in + 1
spool_directory[next_free_slot] ← data
```

```
in=7
next_free_slot = 7
```

```
in=8
next_free_slot = in
in=in+1
```

```
in=8
next_free_slot = 7
```

```
in=8
```
Code run by processes A & B

\[
\begin{aligned}
\{ & \text{next\_free\_slot} = \text{in} \\
& \text{in} = \text{in} + 1 \\
\} \\
\text{spool\_directory}[ \text{next\_free\_slot} ] \leftarrow \text{data}
\end{aligned}
\]
Critical Regions

Four conditions to provide mutual exclusion:

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region
Semaphores

A semaphore consists of:

- A variable \( v \)
- A thread list \( L \) \textit{(list of waiting threads)}
- An Initialize function
- A function \( P \) (down/sleep/wait operation)
- A function \( V \) (up/wakeup/signal operation)

\[
\text{Initialization}(val) \\
\{ \\
\quad v = val; \\
\quad L = \text{Null}; \\
\}
\]

\[
\text{\underline{DOWN}} \\
\quad v--; \\
\quad \text{if}(v < 0) \\
\quad \quad \{ \\
\quad \quad \quad \text{add thread to } L; \\
\quad \quad \quad \text{block}; \\
\quad \quad \} \\
\]

\[
\text{\underline{UP}} \\
\quad v++; \\
\quad \text{if}(v \leq 0) \\
\quad \quad \{ \\
\quad \quad \quad \text{Remove a thread } T \text{ from } L \\
\quad \quad \quad \text{Unblock}(T); \\
\quad \quad \} \\
\]
#define N 100
typedef int semaphore; /* semaphores are a special kind of int */
semaphore mutex = 1;  /* controls access to critical region */
semaphore empty = N;  /* counts empty buffer slots */
semaphore full = 0;   /* counts full buffer slots */

void producer(void)
{
  int item;

  while (TRUE) {
    item = produce_item(); /* TRUE is the constant 1 */
    down(&empty); /* generate something to put in buffer */
    down(&mutex); /* decrement empty count */
    insert_item(item);
    up(&mutex); /* put new item in buffer */
    up(&full); /* leave critical region */
    up(&full); /* increment count of full slots */
  }
}

void consumer(void)
{
  int item;

  while (TRUE) {
    /* infinite loop */
    down(&full); /* decrement full count */
    down(&mutex); /* enter critical region */
    item = remove_item(); /* take item from buffer */
    up(&mutex); /* leave critical region */
    up(&empty); /* increment count of empty slots */
    consume_item(item); /* do something with the item */
  }
}
Scheduling
Schedulers

Short Term

Long Term

Medium Term

Unblock is done by another task (a.k.a. wakeup, release, allocate, V)
Block is a.k.a sleep, request, P
Scheduling Algorithm Goals

All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems
- Response time - respond to requests quickly
- Proportionality - meet users’ expectations

Real-time systems
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems
Preemptive vs. Non-preemptive scheduling

• Non-preemptive scheduling:
  − Each process completes its full CPU burst cycle before the next process is scheduled.
  − No time slicing or CPU stealing occurs.
  − Once a process has control of the CPU, it keeps control until it gives it up (e.g. to wait for I/O or to terminate).
  − Works OK for batch processing systems, but not suitable for time sharing systems.

• Preemptive scheduling:
  − A process may be interrupted during a CPU burst and another process scheduled. (E.g. if the time slice of the first process expires).
  − More expensive implementation due to process switching.
  − Used in all time sharing and real time systems.
Costs of Preemptive Scheduling

• Preemptive scheduling leads to some problems that the OS must deal with:

• Problem 1: inconsistent data:
  – Suppose process 1 is updating data when preempted by process 2.
  – Process 2 may then try to read the data, which is in an inconsistent state.
  – The OS needs mechanisms to coordinate shared data.

• Problem 2: Kernel preemption:
  – Suppose the kernel is preempted while updating data (e.g. I/O queues) used by other kernel functions. This could lead to chaos.
  – UNIX solution: Wait for the system call to complete or have an I/O block take place if in kernel mode.
  – Problem with UNIX solution: Not good for real time computing.
Multilevel Queues

Queue headers

Priority 4
Priority 3
Priority 2
Priority 1

Runnable processes
(Highest priority)

(Lowest priority)

highest priority
system processes
interactive processes
interactive editing processes
batch processes
student processes

lowest priority
Optimization Criteria

• The scheduling criteria are optimization problems. We would like to maximize or minimize each.

• Question: Maximize or Minimize?
  – CPU utilization:
  – throughput:
  – turnaround time:
  – waiting time:
  – response time:

• Can all criteria be optimized simultaneously?

• Usually try to optimize average times (although sometimes optimize minimum or maximum)
Round Robin

- First process in the Ready queue is given the CPU
- A timer is set to $q$ (quantum) time units
- When the timer expires the process is placed back in the Ready queue
## Round Robin

Processes arrive in the order: A, B, C, D

<table>
<thead>
<tr>
<th>Process</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
</tr>
</tbody>
</table>

q = 3

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>15</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Wait times are: A=0+9; B=3; C=6; D=9; Avg=6.75 (FCFS: 9)
Turnaround times are: A=9+9; B=3+3; C=6+3; D=9+3; Avg=11.25 (FCFS: 13.5)

- Preemptive FCFS
- As q gets larger RR approaches FCFS
- As q gets smaller RR approaches Processor Sharing
Chapter 3: Deadlock
Deadlock Modeling

A

Assigned-to

B
Sample Deadlock

A

Waiting-on

B

Assigned-to
Deadlock Modeling

- Modeled with directed graphs

- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U
Deadlocks

• Formal definition:
  *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause*

• Usually the event is release of a currently held resource

• None of the processes can …
  – run
  – release resources
  – be awakened
Four **Necessary** Conditions for Deadlock

1. **Mutual exclusion condition**
   - Each resource assigned to 1 process or is available

2. **Hold and wait condition**
   - Process holding resources can request additional

3. **No preemption condition**
   - Previously granted resources cannot forcibly taken away

4. **Circular wait condition**
   - Must be a circular chain of 2 or more processes
   - Each is waiting for resource held by next member of the chain
## Summary of Approaches to Deadlock Prevention

<table>
<thead>
<tr>
<th>Condition</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual exclusion</td>
<td>Spool everything</td>
</tr>
<tr>
<td>Hold and wait</td>
<td>Request all resources initially</td>
</tr>
<tr>
<td>No preemption</td>
<td>Take resources away</td>
</tr>
<tr>
<td>Circular wait</td>
<td>Order resources numerically</td>
</tr>
</tbody>
</table>
Deadlock Prevention
Attacking the Mutual Exclusion Condition

• Some devices (such as printer) can be spooled
  – only the printer daemon uses printer resource
  – thus deadlock for printer eliminated

• Not all devices can be spooled

• Principle:
  – avoid assigning resource when not absolutely necessary
  – as few processes as possible actually claim the resource
Attacking the Hold and Wait Condition

• Require processes to request ALL resources before starting
  – a process never has to wait for what it needs

• Problems
  – may not know required resources at start of run
  – also ties up resources other processes could be using

• Variation:
  – process must give up all resources
  – then request all immediately needed
Attacking the No Preemption Condition

• This is not a viable option
• Consider a process given the printer
  – halfway through its job
  – now forcibly take away printer
  – !!??
Attacking the Circular Wait Condition

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

• Normally ordered resources
• A resource graph
Banker's Algorithm for Multiple Resources

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Resources assigned

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Resources still needed

Existing (E) = (6342)
Possessed (P) = (5322)
Available (A) = (1020)
Chapter 4:
Memory Management
Swapping

Memory allocation changes as
- processes come into memory
- leave memory

Shaded regions are unused memory
Relocation and Protection (in multiprogramming)

• Cannot be sure where program will be loaded in memory
  – address locations of variables, code routines cannot be absolute
  – must keep a program out of other processes’ partitions

• Use base and limit values
  – address locations added to base value to map to physical address
  – address locations larger than limit value is an error
CPU utilization as a function of number of processes in memory

![Graph showing CPU utilization (in percent) as a function of degree of multiprogramming for different I/O wait percentages (20%, 50%, 80%).]
Chapter 4

1. A computer system has enough room to hold four programs in its main memory. These programs are idle waiting for I/O half the time. What fraction of the CPU time is wasted?

CPU Utilization = 1 - p^n,

where n is the degree of multi programming, and p is fraction of time a process sends waiting for I/O.

p^n is the probability that all n processes are waiting for I/O. The chance that all 4 programs are idle is

p^n = (1/2)^4 = 1/16 = 0.0625, 6.25%.
Memory Management with Bit Maps

- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list
Memory Management

• Bitmap
  – Simple +
  – Small(?)/fixed size +
    • Mapping unit too small -> large bitmap
    • Mapping unit too large -> internal fragmentation
  – Slow to search –

• Linked List
  – Little more complex –
  – Dynamic size +
Memory Allocation

- First fit
- Next fit
- Best fit
- Worst fit
- Quick fit
Page Tables

Internal operation of MMU with 16 4 KB pages
Chapter 4

8. Using the page table of Fig. 4-10 (p. 204), give the physical address corresponding to each of the following virtual addresses:

(a) 20
(b) 4100
(c) 8300
• 20: is on page-0 (starting at va=0), offset=20
  – Page-0 maps to frame-2, frame-2 begins at 2*4096=8192, + 20 = 8212

• 4100: is on page-1 (starting at va=4096), offset= (4100 – 1*4096)= 4
  – Page-1 maps to frame-1, frame-1 begins at 1*4096=4096, + 4 = 4100

• 8300: is on page-2 (starting at va=8192), offset= (8300 – 2*4096) = 108
  – Page-2 maps to frame-6, frame-6 begins at 6*4096=24576, + 108 = 24684
Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page

- Modified page must first be saved
  - unmodified just overwritten

- Better not to choose an often used page
  - will probably need to be brought back in soon
Locality

• **Temporal locality**: If a word is referenced now, it is *likely* to be referenced in the near future.
  – This argues for *caching* referenced words, i.e. keeping the referenced word near the processor for a while.

• **Spatial locality**: If a word is referenced now, nearby words are *likely* to be referenced in the near future.
  – This argues for *prefetching* words around the currently referenced word.
Optimal Page Replacement Algorithm

• Replace page needed at the farthest point in future
  – Optimal but unrealizable

• Estimate by …
  – logging page use on previous runs of process
  – although this is impractical
Simulating LRU in Software

- The aging algorithm simulates LRU in software
- Note 6 pages for 5 clock ticks, (a) – (e)
Gottlieb Practice #4d

6 1 2 7 3 7 2 1 6 3
F F F F F H H H F F
6 6 6 6 1 1 1 3 7 2
1 1 1 2 2 3 7 2 1
2 2 7 3 7 2 1 6
7 3 7 2 1 6 3
# Review of Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximation to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
<tr>
<td>Working set</td>
<td>Somewhat expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good efficient algorithm</td>
</tr>
</tbody>
</table>
Page Fault Frequency (PFF)

- Weighted Page fault avg
- At ‘B’ the process becomes a potential page frame donor
Load Control

• Despite good designs, system may still thrash

• When PFF algorithm indicates
  – some processes need more memory
  – but no processes need less

• Solution:
  Reduce number of processes competing for memory
  – swap one or more to disk, divide up pages they held
  – reconsider degree of multiprogramming
Medium Term Scheduling

Unblock is done by another task (a.k.a. wakeup, release, allocate, V)
Block is a.k.a sleep, request, P
Small Page Size

• Advantages
  – less internal fragmentation
  – better fit for various data structures, code sections
  – less unused program in memory

• Disadvantages
  – programs need many pages, larger page tables
Implementation Issues
Operating System Involvement with Paging

Four times when OS involved with paging
1. Process creation
   - determine program size
   - create page table
2. Process execution
   - MMU reset for new process
   - TLB flushed
3. Page fault time
   - determine virtual address causing fault
   - swap target page out, needed page in
4. Process termination time
   - release page table, pages
Page Fault Handling (1)

1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address, seeks page frame
5. If selected frame is dirty, write it to disk
Page Fault Handling (2)

6. OS brings schedules new page in from disk
7. Page tables updated
   - Faulting instruction backed up to when it began
6. Faulting process scheduled
7. Registers restored
   - Program continues
Chapter 5: I/O

- Memory-Mapped I/O
- Programmed I/O
- DMA
- Software Drivers and Buffering
- Disks, RAID
- Disk access time
DMA Transfer

1. CPU programs the DMA controller
2. DMA requests transfer to memory
3. Data transferred
4. Ack

CPU

DMA controller

Address
Count
Control

Drive

Disk controller

Buffer

Main memory

Interrupt when done
Device Driver interface

Note: This picture is excerpted from Write a Linux Hardware Device Driver, Andrew O'Shaughnessy, Unix world
Circular Buffering

- Two buffer may not be sufficient
- Recall Producer/Consumer
- Linux sk_buf
Is Buffering Always Good?

- Networking may involves many copies
- Copying reduces system performance
A salesman claims ... their disk driver used the elevator algorithm and also queued multiple requests within a cylinder in sector order. A student, Harry Hacker, was impressed and bought one. He took it home and wrote a program to randomly read 10,000 blocks spread across the disk. To his amazement, the performance that he measured was identical to what would be expected from first-come first-served. Was the salesman necessarily lying?

The salesman is not necessarily lying. A program that reads 10,000 blocks issues the requests one at a time. So the driver sees requests one at a time and can only process them in arrival order. Harry should try starting many processes at the same time to see if the elevator algorithm works.
Disk Access Time

calculation of disk access time for WD18300 (see Tanenbaum p. 301 for specs.)

access_time = seek_time + rotational_delay + transfer_time + other_delays

seek_time = 6.9ms

rotational_delay = rotation_time/2 = 8.33/2 = 4.165ms

transfer_rate (Kbytes/ms) = (281 * 512) bytes/track * rotation_time
=> 143.8 (KB/track) * 1/8.33 (track/ms) = 17.26 KB/ms

Now, let's say we wanted to transfer 10K bytes, assume other_delays=0

the transfer_time (ms) = 1/17.26 (ms/KB) * 10 (KB) = 0.57ms

so, access_time = 6.9ms + 4.16ms + 0.57 ms = 11.63ms

This shows us that about 90% of the transfer_time (for the first track) is needed just to reach the track (seek and rotational delay)!
Chapter 6: File Systems

• File structure
• Directory structure
• Metadata
• UNIX File System
• Log Structured File System
#17 Consider the i-node shown in Fig. 6-15 (p.405). If it contains 10 direct address of 4 bytes each and all disk blocks are 1024 KB, what is the largest possible file?
#17 Consider the i-node shown in Fig. 6-15 (p.405). If it contains 10 direct address of 4 bytes each and all disk blocks are 1024 Bytes*, what is the largest possible file?

The indirect block holds : 1024 / 4 = 256 block addresses.
Add the 10 direct addresses for
Total of 266 block addresses
266 * 1024 = 266 KB (max file size)

* Book erroneously has KB as units
#19 Two computer science students, Carolyn and Elinor, are having a discussion about i-nodes. Carolyn maintains that memories have gotten so large and so cheap that when a file is opened, it is simpler and faster just to fetch a new copy of the i-node into the i-node table, rather than search the entire table to see if it is already there. Elinor disagrees. Who is right?

Elinor is right. Having two copies of the same i-node in the OS open i-nodes table could easily lead to inconsistencies.
Access Control Matrix

- **Access control list (ACL)**
  - Store column of matrix with the resource
- **Capability**
  - User holds a “ticket” for each resource
  - Two variations
    - store row of matrix with user
    - Un-forgable ticket in user space

<table>
<thead>
<tr>
<th></th>
<th>File 1</th>
<th>File 2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>read</td>
<td>write</td>
<td>-</td>
</tr>
<tr>
<td>User 2</td>
<td>write</td>
<td>write</td>
<td>-</td>
</tr>
<tr>
<td>User 3</td>
<td>-</td>
<td>-</td>
<td>read</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User m</td>
<td>read</td>
<td>write</td>
<td>write</td>
</tr>
</tbody>
</table>

Access control lists are widely used, often with groups
Some aspects of capability concept are used in Kerberos, ...