• Threads assignment

Why did we see an improvement in throughput for our user level threads?
IO blocks right? No, the IO goes to a communication buffer and can run so IO
blocking is minimal

• Deadlock

Deadlock is cyclic waiting in critical section (CS).

How can we deal with deadlock?
1. Detect and Recover
   a. Detect by looking for dependencies; such as with a graph, look for
cycles
   b. Recovery is hard. We don’t want to leave deranged data, like a
   linked list that has pointer pointing wildly.
   c. Transaction (logging) make recovery easy but are very expensive.
2. Prevention
   a. Before you grant, check graph and if you complete a cycle don’t
   grant the access to the CS.
3. avoidance
   a. Make sure you never get into deadlock by some policy.
   b. One example of a policy is a strict ordering on lock acquisition so
that you can’t acquire a lock in random order, but you need to get
M1 before you get M2, this guarantees that nobody is every
waiting for M1 and already has M2 locked.
   c. This is hard at an operating system level, or any large project level,
where the lock order isn’t defined universally. It becomes too
much to manage for certain jobs.

• Memory

Memory abstraction –
   - Virtual memory – mechanism
   - Replacement – policy

Virtual Memory –
   Allow the program the view of ~ infinite memory.
   pluses    - Don’t worry if a program can fit in memory
   - program can run on two machines with different memory sizes
   minuses   - leads to sloppy programming
- makes performance difficult to measure based on HD speed and other processes. Ie unpredictability.

What are the paradigms to mapping virtual mem and real mem?
How do you split up address space?

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Chunks</th>
<th>Fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>Variable</td>
<td>Big (whole program)</td>
<td>External</td>
</tr>
<tr>
<td>Paging</td>
<td>Fixed</td>
<td>Small (several pages/program)</td>
<td>Internal</td>
</tr>
</tbody>
</table>

Typically these days, machines use both segmentation and paging, paging on top of segmentation. So we break the memory as a whole into segments, and break each segment into different pages.

What is the mechanism for address translation?

1. In Segments
   a. A segment contains a base, bound, permissions and a valid? bit.
   b. A segment table is a reference to each of the segments and looks as follows

<table>
<thead>
<tr>
<th>Mem address</th>
<th>Base</th>
<th>Bound</th>
<th>Perm</th>
<th>Valid? bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A1</td>
<td>Size</td>
<td>Permissions</td>
<td>0/1</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

   Where A would point to memory location 0, and then return all relevant information about that particular segment.

   The above table would need to be in very fast memory because it may need to be used every load, and ms waiting times are unacceptable.

2. In Paging
   a. You have a page table of the form

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
<th>Permission</th>
<th>Valid?</th>
<th>Dirty bit</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   Here the dirty bit is to tell if the page was modified and needs to be written back to mem or not.

   For this scheme, you will send a page number and an offset and you will get back a frame # and an offset.

   This page table can become huge with many things running or large things running, so we can’t keep this in our small but fast memory, so we create something new and fast, a translation look-aside buffer, or TLB.
The PID in the table is for access, we don’t want other proc’s trying to address our page.

The purpose of the page table is to make memory look bigger, but it actually makes it smaller because it eats up some of our memory that could have been used in other ways. This brings about the problem of finding the cost/return ratio that is optimal for each system.

Typically most of the page table is stored in VM, but we need some base stored in memory so that we can access the rest of it in the VM.

When we try an address in a page table and it isn’t there, we call this a page fault. A page fault looks as follows –
   i. Trap into the kernel to handle this
   ii. Look in the page table in mem
   iii. Possible another page fault if the page is in VM
   iv. Pull page into mem

When you begin, you have a set amount of address space, not the max, so it doesn’t need to keep pointers for it. When you address something out of your fixed space, you do a system call and get more VM space.

Translation table is all hardware controlled until you get a TLB miss, then the kernel gets invoked and brings a page into memory.

What is the cost of a page fault?
   Several ms for disk access.
So we don’t want page faults very often, that is why page faults are handled in software, so that we may manage the faults more efficiently.

Demand paging is only bringing a page in when it is needed via a page fault. But the first few actions will all cause page faults because nothing is in mem yet.

Eviction – when me is full and we get a page fault…who do we kick out?
   LFU
   LRU
   FIFO

If you can predict the systems behavior, then the optimal is to kick out the one who will be used farthest in the future.

LRU best approximates the optimal based on temporal locality. This isn’t always the case, but often it is. Some cases LRU can be the worst.

How do you implement LRU?
List – If a page gets accessed move it to the front of the list.
    If we get a page fault then remove the last element from the list.

    But this takes 3 memory references when you only want to do 1.
    Sometimes kicking out randomly is just as good because of all the
    memory access you are doing.

Clock – Second Chance – both the same name for another method.
    Picture a clock, with pages at each of the hour markers.
    The hour hand is what decides who gets kicked out at a page fault.
    At a page fault, the clock hand starts moving.

    Rules –
        - If you reference a page, turn a clock bit to 1.
        - If clock hand gets to you and you are a 1, then change the clock bit
to 0.
        - If clock hand gets to you and you are a 0, you get kicked out.

    This isn’t a strict LRU, but a very close approximation.

N-Chance has n-bits and can allow a build up to save yourself several times.

This clock representation can still use up a lot of memory.

In all of this we are trying to cut evictions to get a max throughput for the system.
But this may choke out little processes because a bigger job is running and a
smaller process is taking as many page faults.

Thrashing is a concept where all you are doing is page faulting, and switching
memory, and doing expensive IO operations to VM.

This is why we have the idea of a working set, which is relative to the amount of
page faulting you. It is a way to keep the number of page faults down based on
the systems resources.

The working set only allows some set of the active process to be running at a
given time. Then when some finish others may jump in.

Our working set is reasonable if we get (x # of page faults)/(some time period)
and it is a sufficiently small enough number.

If it isn’t sufficiently small then we may need to stop a few processes.
Mach Paper –

Developed a general segmentation model, showed how it fits with a paging model, and how to implement it easily.

Memory object – segment, shared segment, is a general encapsulation as anything that can be stored in mem. Files, etc.

Micro-kernel provides a way for process to talk to each other.

Mach can be backed by any server via messages. The policies can be managed outside of the server.

Copy-on-write – shared mem-mapping until someone writes, only then is an expensive copy operation performed.

TLB is a problem on multiproc, should be coherent. Invalidation on all TLBs is hard.

Opal Paper –

Persistent address across memory for mult procs. Sharing becomes hard though, because of pointer consistency.

Why private VM, Opal makes public address space but protects that space through other means.

A process is a protection domain with space and memory.

Because pointers can be kept constant, we can save pointers easier on disk and we can save data structures easier. This makes sharing nice too.