A Solution to the

Gaming Parlor

Programming Project
Scenario:
Front desk with dice (resource units)
Groups request (e.g., 5) dice (They request resources)
Groups must wait, if none available
A list of waiting groups... A “condition” variable
Dice are returned (resources are released)
The condition is signaled
The group checks and finds it needs to wait some more
The group (thread) waits...and goes to the end of the line

Problem?
The Gaming Parlor - Solution

Scenario:
Front desk with dice (*resource units*)
Groups request (e.g., 5) dice (*They request resources*)
Groups must wait, if none available
Dice are returned (*resources are released*)
A list of waiting groups... A “condition” variable
The condition is signalled
The group checks and finds it needs to wait some more
The group (thread) waits...and goes to the end of the line

Problem?
Starvation!
The Gaming Parlor - Solution

**Approach:**
Serve every group “first-come-first-served”.

**Implementation:**
Keep the thread at the front of the line separate “Leader” - the thread that is at the front of the line
Use 2 condition variables.
“Leader” will have at most one waiting thread
“RestOfLine” will have all other waiting threads
function Group (numDice: int) {
    var i: int
    for i = 1 to 5 {
        gameParlor.Acquire (numDice)
        currentThread.Yield ()
        gameParlor.Release (numDice)
        currentThread.Yield ()
    }
    endFor
} endFunction

thA.Init ("A")
thA.Fork (Group, 4)
...
The Monitor

class GameParlor
  superclass Object

fields
  monitorLock: Mutex
  leader: Condition
  restOfLine: Condition
  numberDiceAvail: int
  numberOfWaitingGroups: int

methods
  Init ()
  Acquire (numNeeded: int)
  Release (numReturned: int)
  Print (str: String, count: int)

endClass
The Release Method

```plaintext
method Release (numReturned: int)
    monitorLock.Lock ()

    -- Return the dice
    numberDiceAvail = numberDiceAvail + numReturned

    -- Print
    self.Print ("releases and adds back", numReturned)

    -- Wakeup the first group in line (if any)
    leader.Signal (&monitorLock)

    monitorLock.Unlock ()
endMethod
```
The Acquire Method

method Acquire (numNeeded: int)
    monitorLock.Lock ()
    -- Print
    self.Print ("requests", numNeeded)
    -- Indicate that we are waiting for dice.
    numberOfWaitingGroups = numberOfWaitingGroups + 1
    -- If there is a line, then get into it.
    if numberOfWaitingGroups > 1
        restOfLine.Wait (&monitorLock)
    endIf
    -- Now we're at the head of the line. Wait until there are enough dice.
    while numberDiceAvail < numNeeded
        leader.Wait (&monitorLock)
    endwhile
    ...
    ...
The Acquire Method

... 

-- Take our dice.
numberDiceAvail = numberDiceAvail - numNeeded

-- Now we are no longer waiting; wakeup some other group and leave.
numberOfWaitingGroups = numberOfWaitingGroups - 1
restOfLine.Signal (&monitorLock)

-- Print
self.Print ("proceeds with", numNeeded)

monitorLock.Unlock ()
endMethod
Virtual Memory (2)
Inverted Page Tables

Problem:
- Page table overhead increases with address space size
- Page tables get too big to fit in memory!

Consider a computer with 64 bit addresses
- Assume 4 Kbyte pages (12 bits for the offset)
- Virtual address space = $2^{52}$ pages!
- Page table needs $2^{52}$ entries!
- This page table is much too large for memory!
Inverted Page Tables

How many mappings do we need (maximum) at any time?
Inverted Page Tables

How many mappings do we need (maximum) at any time?
We only need mappings for pages that are in memory!
Inverted Page Tables

An inverted page table
- Has one entry for every resident memory page
- Roughly speaking, one for each frame of memory
- Records which page is in that frame
- Can not be indexed by page number

So how can we search an inverted page table on a TLB miss fault?
Inverted Page Tables

Given a page number (from a faulting address), do we exhaustively search all entries to find its mapping?
Inverted Page Tables

Given a page number (from a faulting address), do we exhaustively search all entries to find its mapping?

- No, that’s too slow!
- A hash table could allow fast access given a page number
- O(1) lookup time with a good hash function
Hash Tables

Data structure for associating a key with a value
- Apply hash function to key to produce a hash
- Hash is a number that is used as an array index
- Each element of the array can be a linked list of entries (to handle collisions)
- The list must be searched to find the required entry for the key (entry’s key matches the search key)
- With a good hash function the average list length will be short
Inverted page table

Traditional page table with an entry for each of the $2^{52}$ pages

Indexed by virtual page

$2^{52} - 1$

256-MB physical memory has $2^{16}$ 4-KB page frames

Indexed by hash on virtual page

Hash table

Virtual page

Page frame
Which Page Table Design is Best?

The best choice depends on CPU architecture
64 bit systems need inverted or multi-level page tables
Some systems use a combination of regular page tables together with segmentation
Memory Protection

Protection though addressability

- If address translation only allows a process to access its own pages, it is implementing memory protection.

But what if you want a process to be able to read and execute some pages but not write them? (eg. the text segment)

Or read and write them but not execute them? (eg. the stack)

Can we implement protection based on access type?
When should the protection checking be done?
Memory Protection

How is protection checking implemented?
- Compare page protection bits with process capabilities and operation types on every load or store
- That is expensive, and requires hardware support!

How can protection checking be done efficiently?
- Use the TLB as a “protection” look-aside buffer as well as a translation lookaside buffer
- Use special segment registers
Protection Lookaside Buffer

A TLB is often used for more than just “translation”
Memory accesses need to be checked for validity
- Does the address refer to an allocated segment of the address space?
  If not: **fault!**
- Is this process allowed to access this memory segment?
  If not: **fault!**
- Is the type of access valid for this segment?
  Read, write, execute ...?
  If not: **protection fault!**
## Protection Checking With a TLB

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>
Page Grain Protection

Each page table entry needs extra bits to support page grain protection based on access type.
Segmented Address Spaces

*Traditional Virtual Address Space*
- “flat” address space (1 dimensional)
- Mapped by a single page table

*Segmented Address Space*
- Program made of several regions or “segments”
- Each segment is its own mini-address space
- Each segment could have its own base register
- Each segment could have its own page table
- The program must always say which segment it is referring to when using an address
  - either embed a segment id in an address
  - or load a value into a segment register and refer to the register
Segmented Address Spaces

Example: A compiler
Segmented Memory

- Symbol table
- Source text
- Parse tree
- Call stack
- Constants
Instruction and Data Spaces

* One address space
* Separate I and D spaces
Sharing Segments

If you run multiple instances of the same program, why have more than one copy of the instructions in memory?

**Goal:**

Share pages among “processes” (not just threads!)
- Cannot share writable pages because processes would notice each other’s effects
- But readable pages, such as the text segment, can be shared without noticing
Delayed Copying of Writeable Pages

Page sharing optimizations of process Fork

Normal usage: copy the parent’s virtual address space and immediately do an “Exec” system call

- Exec overwrites the calling address space with the contents of an executable file (ie a new program)
- So why waste all that work copying pages that are not used?

Desired Semantics:
- Pages are copied, not shared

Observations
- Copying every page in an address space is expensive!
- Processes can’t notice the difference between copying and sharing unless pages are modified!
- Delay copying until you know its necessary!
Copy-on-Write Page Sharing

Initialize new page table, but point entries to existing page frames of parent, i.e. share pages
Temporarily mark all pages “read-only”
Continue to share all pages until a protection fault occurs
Protection fault (copy-on-write fault):
  - Is this page really read only or is it writable but temporarily protected for copy-on-write?
  - If it is writable, copy the page, mark both copies writable, resume execution as if no fault occurred

This is an interesting new use of protection faults!
Sharing Writeable Pages

Why not give processes the option of sharing writeable pages if they both agree?
- Allows fast communication of data between processes
- Similar to how threads share memory

What is the difference between two processes sharing all pages, and a single address space with two threads?
- Does this amount to the same thing?
Virtual Memory Implementation

When is the virtual memory management part of the kernel invoked?

- Process creation
- Process is scheduled to run
- A fault occurs
- Process termination
Virtual Memory Implementation

Process creation
  - Determine the process size
  - Create new page table
Virtual Memory Implementation

Process is scheduled to run
- MMU is initialized to point to new page table
- TLB is flushed (unless it’s a tagged TLB)
Virtual Memory Implementation

A fault occurs
- Could be a TLB-miss fault, segmentation fault, protection fault, copy-on-write fault ...
- Determine the virtual address causing the problem
- Determine whether access is allowed, if not terminate the process
- Refill TLB (TLB-miss fault)
- Copy page and reset protections (copy-on-write fault)
- Swap an evicted page out & read in the desired page (page fault)
Virtual Memory Implementation

Process termination
- Release / free all frames (if reference count is zero)
- Release / free the page table
Handling a Page Fault

1. Hardware traps to kernel
2. Save the other registers
3. Determine the virtual address causing the problem
4. Check validity of the address
   - Determine which page table entry is needed
   - May need to kill the process if address is invalid
5. If page is not resident in memory we need to get it from disk
6. Find the frame to use to hold it (page replacement algorithm)
7. Has the page currently in the target frame been modified?
   - If so, write it out (& schedule other processes)
8. Read in the desired frame from disk
9. Update the page tables
   (continued)
Handling a Page Fault

10. Load the new page table entry into the TLB
11. Back up the current instruction (i.e., the *faulting instruction*)
12. Schedule the faulting process to run again
13. Reload registers and resume execution of faulting process
Locking Pages in Memory

Virtual memory and I/O interact, requiring *pinning* of pages

*Example:*
- One process does a read system call
  - This process suspends during I/O
- Another process runs
  - It has a page fault
  - Some page is selected for eviction
  - The frame selected contains the page involved in the read

*Solution:*
Each frame has a flag: “Do not evict me”. Must always remember to un-pin the page!
Managing the Swap Area on Disk

Approach #1:
A process starts up
   Assume it has N pages in its virtual address space
A region of the swap area is set aside for the pages
There are N pages in the swap region
The pages are kept in order
For each process, we need to know:
   Disk address of page 0
   Number of pages in address space
Each page is either...
   In a memory frame
   Stored on disk
Approach #1

Main memory

Pages

0  3
4  6

Page table

Disk

Swap area

7
5
2
1
Approach #2

What if more pages are allocated and the virtual address space grows during execution?

Approach #2
Store the pages in the swap space in a random order
View the swap file as a collection of free swap frames
Need to evict a frame from memory?
  - Find a free swap frame
  - Write the page to this place on the disk
  - Make a note of where the page is
  - Use the page table entry? (Just make sure the valid bit is still zero!)
Next time the page is swapped out, it may be written somewhere else.
Approach #2

This picture uses a separate disk map data structure to tell where pages are stored on disk rather than using the page table.

Some information, such as protection status, could be stored at segment granularity.
Approach #3

Swap to a file
- Each process has its own swap file
- File system manages disk layout of files
Approach #4

Swap to an external pager process (object)
A user-level external pager determines policy
  - Which page to evict
  - When to perform disk I/O
  - How to manage the swap file

When the OS needs to read in or write out a page it sends a message to the external pager
  - Which may even reside on a different machine
Approach #4

1. Page fault

2. Needed page

3. Request page

4. Page arrives

5. Here is page

6. Map page in

User process

External pager

Fault handler

MMU handler

Main memory

Disk

User space

Kernel space
Mechanism vs Policy

Kernel contains
- Code to interact with the MMU
  - This code tends to be *machine dependent*
- Code to handle page faults
  - This code tends to be *machine independent* and may embody generic operating system policies
Paging Performance

Paging works best if there are plenty of free frames.
If all pages are full of dirty pages we must perform 2 disk operations for each page fault.
- This doubles page fault latency.

It can be a good idea to periodically write out dirty pages in order to speed up page fault handling delay.
Paging Daemon

Paging daemon
- A kernel process
- Wakes up periodically
- Counts the number of free page frames
- If too few, run the page replacement algorithm...
  - Select a page & write it to disk
  - Mark the page as clean
- If this page is needed later... then it is still there
- If an empty frame is needed then this page is evicted
Spare Slides
Paged Segments in MULTICS

Each segment is divided up into pages. Each segment descriptor points to a page table.
# Paged Segments in MULTICS

Each entry in segment table

<table>
<thead>
<tr>
<th>18</th>
<th>9</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
</table>
| Main memory address of the page table | Segment length (in pages) | Page size:  
0 = 1024 words  
1 = 64 words  
0 = segment is paged  
1 = segment is not paged  
Miscellaneous bits  
Protection bits |
Paged Segments in MULTICS

Conversion of a 2-part MULTICS address into a main memory address
The MULTICS TLB

<table>
<thead>
<tr>
<th>Comparison field</th>
<th>Segment number</th>
<th>Virtual page</th>
<th>Page frame</th>
<th>Protection</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>Read/write</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>Read only</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>Read/write</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>Execute only</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>Execute only</td>
<td>9</td>
</tr>
</tbody>
</table>

Simplified version of the MULTICS TLB
Existence of 2 page sizes makes actual TLB more complicated
Conversion of a (selector, offset) pair to a linear address
Pentium Segmentation & Paging

Pentium segment descriptor
## Backing up the PC

Consider a multi-word instruction. The instruction makes several memory accesses. One of them faults. The value of the PC depends on when the fault occurred. How can you know what instruction was executing?

### MOVE.L #6(A1), 2(A0)

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>First operand</th>
<th>Second operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>MOVE</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

The instruction is 16 bits long.
Solution

Lots of architecture-specific code in the kernel

Hardware support (precise interrupts)
  - Dump internal CPU state into special registers
  - Make special registers accessible to kernel
Complications

What if you swapped out the page containing the first operand in order to bring in the second one?