A Solution to the

Gaming Parlor

Programming Project
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
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
The Threads

```plaintext
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

```java
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

```plaintext
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 (3)
Page Sharing

In a large multiprogramming system some users run the same program at the same time
- Why have more than one copy of each page in memory?

**Goal:**
Share pages among “processes” (not just threads!)
- Cannot share writable pages
- If writable pages were shared processes would notice each other’s effects
- Text segment can be shared
Process 1
address space

Process 1
page table

Physical memory

Process 2
address space

Process 2
page table

Data (rw)

Stack (rw)

Instructions (rx)
Page Sharing in Fork System Call

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)

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!
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!
Page Management System Calls

Goal: Allow some processes more control over paging!

System calls added to the kernel
- A process can request a page before it is needed
  - Allows processes to grow (heap, stack etc)
- Processes can share pages
  - Allows fast communication of data between processes
  - Similar to how threads share memory
... so what is the difference?
Unix Processes

- Stack Pages
- Data Pages
- Text Pages
- Not allocated to the virtual address space
Unix Processes

Stack Pages

Not allocated to the virtual address space

Data Pages

Text Pages

Page Zero: Invalid to catch null pointer dereferences; can be used by OS.
Unix Processes

The stack grows;
Page requested here

- Stack Pages
- Data Pages
- Text Pages
- Not allocated to the virtual address space
Unix Processes

The stack grows;
Page requested here
A new page is allocated
and process continues
Unix Processes

The stack grows;
Page requested here
A new page is allocated
and process continues
Unix Processes

The heap grows;
Page requested here

Stack Pages

Not allocated to the virtual address space

Data Pages

Text Pages
Unix Processes

The heap grows;
Page requested here
A new page is allocated
and process continues
Unix Processes

The heap grows;
Page requested here
A new page is allocated
and process continues
Virtual Memory Implementation

When is the kernel involved?
Virtual Memory Implementation

When is the kernel involved?

- 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 (PC and SR are saved on stack)
2. Save the other registers
3. Determine the virtual address causing the problem
4. Check validity of the address
   - Determine which page is needed
   - May need to kill the process if address is invalid
5. Find the frame to use (page replacement algorithm)
6. Is the page in the target frame dirty?
   - If so, write it out (& schedule other processes)
7. Read in the desired frame from swapping file
8. Update the page tables
   
   (continued)
Handling a Page Fault

9. Back up the current instruction (ie., the *faulting instruction*)
10. Schedule the faulting process to run again
11. Return to scheduler

... 
12. Reload registers
13. Resume execution
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)

16 Bits

1000  MOVE
1002  6
1004  2

{ Opcode
{ First operand
{ Second operand
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?
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
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