ANNOUNCEMENTS

- Recitation for this week will cover required material (Barrier Synchronization) assigned in the reading (C. 21) of the Harmony book.

- Recitation recording will be available!
Memory Management
(3EP, Ch. 12-24)
Previously, on CS4410...
Avoiding Deadlock:
The Banker’s Algorithm

- Sum of max resources needs can exceed total available resources
- Acquiring all resources at once can be inefficient!
- Allow to parcel out resources incrementally as long as
  - there exists a schedule of loan fulfillments such that
    - all clients receive their maximal loan
    - build their house
    - pay back all the loan

E.W. Dijkstra & N. Habermann
Living dangerously: Safe, Unsafe, Deadlocked

**Safe**: For any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests. No deadlock as long as system can enforce that safe schedule!

**Unsafe**: There exists a set of (pending and future) resource requests that leads to a deadlock, independent of the schedule in which requests are processed. Unlucky set of requests can force deadlock.

**Deadlocked**: The system has at least one deadlock.
Detecting Deadlock

- 5 processes, 3 resources.

<table>
<thead>
<tr>
<th>Process</th>
<th>Holds</th>
<th>Available</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P2</td>
<td>2 0 0</td>
<td>0 0 0</td>
<td>0 0 2</td>
</tr>
<tr>
<td>P3</td>
<td>3 0 3</td>
<td>0 0 0</td>
<td>0 0 2</td>
</tr>
<tr>
<td>P4</td>
<td>2 1 1</td>
<td>0 0 0</td>
<td>0 0 2</td>
</tr>
<tr>
<td>P5</td>
<td>0 0 2</td>
<td>0 0 0</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Yes, there is a safe schedule!

but it is not a safe state!
Detecting Deadlock

5 processes, 3 resources.

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_5$</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Available:

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Pending:

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$P_5$</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Cannot determine whether the state is safe

☐ I need Max and Needs for that!

But can determine if the state has a deadlock

☐ Given the set of pending requests, is there a safe sequence? If no, deadlock
Detecting Deadlock

5 processes, 3 resources.

- Cannot determine whether the state is safe
  - I need Max and Needs for that!

- Without Max, can we avoid deadlock by delaying granting requests?
  - NO! Deadlock triggered when request formulated, not granted!
Abstraction is our Business

What I have

- A single (or a finite number) of CPUs
- Many programs I would like to run

What I want: a **Thread**

- Each program has full control of one or more virtual CPUs
Abstraction is our Business

What I have

- A certain amount of physical memory
- Multiple programs I would like to run together, they may need more than the available physical memory

What I want: **an Address Space**

- Each program has as much memory as the machine’s architecture will allow to name
- All for itself
Address Space

- Set of all names used to identify and manipulate unique instances of a given resource

- memory locations (determined by the size of the machine's word)
  - for 32-bit-register machine, the address space goes from 0x00000000 to 0xFFFFFFFF

- memory locations (determined by the number of memory banks mounted on the machine)

- phone numbers (XXX) (YYY-YYYY)

- colors: R (8 bits) + G (8 bits) + B (8 bits)
Virtual Address Space: An Abstraction for Memory

- Virtual addresses start at 0
- Heap and stack can be placed far away from each other, so they can nicely grow
- Addresses are all contiguous
- Size is independent of physical memory on the machine
Processes loaded in memory at some memory location

- virtual address 0 is not loaded at physical address 0

Multiple processes may be loaded in memory at the same time, and yet...

...physical memory may be too small to hold even a single virtual address space in its entirety

- 64-bit, anyone?
II. Memory Isolation

Step 2: Address Translation

Implement a function mapping

\( \langle \text{pid, virtual address} \rangle \) into physical address

Enables:
- Isolation
- Relocation
- Data sharing
- Multiplexing
- Non-contiguity

Virtual

\( P_i \)

Virtual Address: 0xA486D4

Physical Address: 0x5E3A07
II. Memory Isolation

Step 2: Address Translation
- Implement a function mapping (grid, virtual address) into physical address

Isolation
- At all times, functions used by different processes map to disjoint ranges — aka "Stay in your room"

Relocation
- The range of the function used by a process can change over time

Relocation
- The range of the function used by a process can change over time — "Move to a new room"

Data Sharing
- Map different virtual addresses of distinct processes to the same physical address — "Share the kitchen"

Data Sharing
- Map different virtual addresses of distinct processes to the same physical address — "Share the kitchen"

Multiplexing
- Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses — ever lived in a studio?

Multiplexing
- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

Multiplexing
- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

More Multiplexing
- At different times, different processes can map part of their virtual address space into the same physical memory — (change tenants)

Multiplexing
- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

More Multiplexing
- At different times, different processes can map part of their virtual address space into the same physical memory — (change tenants)

Non) Contiguity
- Contiguous virtual addresses can be mapped to non-contiguous physical addresses

Non) Contiguity
- …and non-contiguous virtual addresses can be mapped to contiguous physical addresses

The Power of Mapping
Address Translation, Conceptually

CPU

Virtual Address

Translation

Invalid

Raise Exception

Valid

Physical Memory

Physical Address

Data

Who does this?
Memory Management Unit (MMU)

Hardware device
- Maps virtual addresses to physical addresses

User process
- deals with virtual addresses
- never sees the physical address

Physical memory
- deals with physical addresses
- never sees the virtual address
The Identity Mapping

- Map each virtual address onto the identical physical address
  - Virtual and physical address spaces have the same size
  - Run a single program at a time
    - OS can be a simple library
    - very early computers
- Friendly amendment: leave some of the physical address space for the OS
  - Use loader to relocate process
    - early PCs
More sophisticated address translation

How to perform the mapping efficiently?

☐ So that it can be represented concisely?

☐ So that it can be computed quickly?

☐ So that it makes efficient use of the limited physical memory?

☐ So that multiple processes coexist in physical memory while guaranteeing isolation?

☐ So that it decouples the size of the virtual and physical addresses?

☐ Ask hardware for help!
Base & Bound

Goal: let multiple processes coexist in memory while guaranteeing isolation

Needed hardware
- two registers: Base and Bound (a.k.a. Limit)
- Stored in the PCB

Mapping
- \( pa = va + \text{Base} \)
  - as long as \( 0 \leq va \leq \text{Bound} \)
- On context switch, change B&B (privileged instruction)
**Base & Bound**

- $P_1$: Base = 1000; Bound = 300
- $P_2$: Base = 500; Bound = 400

**Memory Exception**

- **CPU**
  - Virtual address
  - $\leq$ yes
  - Bound Register
  - $+$ no
  - Base Register

**Physical address**

- MAX$_{sys}$
  - 1300
  - 1000
  - 0
Base & Bound

- $P_1$: Base = 1000; Bound = 300
- $P_2$: Base = 500; Bound = 400
**Base & Bound**

- $P_1$: Base = 1000; Bound = 300
- $P_2$: Base = 500; Bound = 400

---

**Diagram:**
- CPU
- Virtual address
- Base Register
- Bound Register
- Memory Exception
- Physical address
- $P_1$: Base = 1000; Bound = 300
- $P_2$: Base = 500; Bound = 400
**Base & Bound**

- **P₁**: Base = 1000; Bound = 300
- **P₂**: Base = 500; Bound = 400

---

Memory Exception

Virtual address

\[ \leq \]

no

yes

1000

300

Boundary Register

1150

Physical address

1300

MAX_{sys}

Context Switch

Base & Bound saved in P₁’s PCB
Base & Bound

- \( P_1 \): Base = 1000; Bound = 300
- \( P_2 \): Base = 500; Bound = 400

Memory Exception

Virtual address

Bound Register

Physical address

Context Switch

\[ 400 \]

\[ 500 \]
On Base & Bound

Contiguous Allocation

- contiguous virtual addresses are mapped to contiguous physical addresses

But mapping entire address space to physical memory

- is wasteful
  - lots of free space between heap and stack...
  - makes sharing hard

- does not work if the address space is larger than physical memory
  - think 64-bit registers...
E Pluribus Unum

- An address space comprises multiple segments
  - contiguous sets of virtual addresses, logically connected
    - heap, code, stack, (and also globals, libraries...)
  - each segment can be of a different size
Segmentation: Generalizing Base & Bound

- Base & Bound registers to each segment
  - each segment independently mapped to a set of contiguous addresses in physical memory
    - no need to map unused virtual addresses

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>10K</td>
<td>2K</td>
</tr>
<tr>
<td>Stack</td>
<td>28</td>
<td>2K</td>
</tr>
<tr>
<td>Heap</td>
<td>35K</td>
<td>3K</td>
</tr>
</tbody>
</table>
Segmentation

Goal: Supporting large address spaces (while allowing multiple processes to coexist in memory)

Needed hardware

- two registers (Base and Bound) per segment
  - values stored in the PCB
- if many segments, a segment table, stored in memory, at an address pointed to by a Segment Table Register (STBR)
  - process’ STBR value stored in the PCB
Segmentation: Mapping

How do we map a virtual address to the appropriate segment?

- Read VA as having two components
  - $s$ most significant bits identify the segment
    - at most $2^s$ segments
  - $o$ remaining bits identify offset within segment
    - each segment’s size can be at most $2^o$ bytes

$k = s + o$ bits
Segment Table

- Use \( s \) bits to index to the appropriate row of the segment table

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Bound (Max 4K)</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>32K</td>
<td>2K</td>
<td>Read/Execute</td>
</tr>
<tr>
<td>Heap</td>
<td>34K</td>
<td>3K</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Stack</td>
<td>28K</td>
<td>3K</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>

- Segments can be shared by different processes
  - Use protection bits to determine if shared Read only (maintaining isolation) or Read/Write (if shared, no isolation)
  - Processes can share code segment while keeping data private
Implementing Segmentation

Segment table generalizes Base & Bound

CPU

Segment Table Base Register

STBR

Logical addresses

Bound

≤ 512

Memory exception

no

yes

Base

40K

Physical addresses

40K

0

MAXsys

MAXsys

0

33
Revisiting `fork()`
Revisiting fork()

- Copying an entire address space can be costly...
  - especially if you proceed to obliterate it right away with exec()!
Revisiting fork():
Segments to the Rescue

Instead of copying entire address space, copy just segment table (the VA->PA mapping)

<table>
<thead>
<tr>
<th>Code</th>
<th>Base</th>
<th>Bound</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32K</td>
<td>2K</td>
<td>RX</td>
</tr>
<tr>
<td>Heap</td>
<td>34K</td>
<td>3K</td>
<td>RW</td>
</tr>
<tr>
<td>Stack</td>
<td>28K</td>
<td>3K</td>
<td>RW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Base</th>
<th>Bound</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Child</td>
<td>32K</td>
<td>2K</td>
<td>RX</td>
</tr>
<tr>
<td>Heap</td>
<td>34K</td>
<td>3K</td>
<td>RW</td>
</tr>
<tr>
<td>Stack</td>
<td>28K</td>
<td>3K</td>
<td>RW</td>
</tr>
</tbody>
</table>

but change all writeable segments to Read only
Revisiting fork():
Segments to the Rescue

Instead of copying entire address space, copy just segment table (the VA->PA mapping)

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Bound</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>32K</td>
<td>2K</td>
<td>RX</td>
</tr>
<tr>
<td>Heap</td>
<td>34K</td>
<td>3K</td>
<td>R</td>
</tr>
<tr>
<td>Stack</td>
<td>28K</td>
<td>3K</td>
<td>R</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Bound</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>32K</td>
<td>2K</td>
<td>RX</td>
</tr>
<tr>
<td>Heap</td>
<td>34K</td>
<td>3K</td>
<td>R</td>
</tr>
<tr>
<td>Stack</td>
<td>28K</td>
<td>3K</td>
<td>R</td>
</tr>
</tbody>
</table>

but change all writeable segments to Read only

Segments in VA spaces of parent and child point to same locations in physical memory
Copy on Write (COW)

- When trying to modify an address in a COW segment:
  - exception!
    - exception handler copies just the affected segment, and changes both the old and new segment back to writeable

- If `exec()` is immediately called, only stack segment is copied!
  - it stores the return value of the `fork()` call, which is different for parent and child
Managing Free space

- Many segments, different processes, different sizes

- OS tracks free memory blocks ("holes")
  - Initially, one big hole

- Many strategies to fit segment into free memory (think "assigning classrooms to courses")
  - First Fit: first big-enough hole
  - Next Fit: Like First Fit, but starting from where you left off
  - Best Fit: smallest big-enough hole
  - Worst Fit: largest big-enough hole
External Fragmentation

Over time, memory can become full of small holes
- Hard to fit more segments
- Hard to expand existing ones

Compaction
- Relocate segments to coalesce holes
External Fragmentation

- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones

Compaction
  - Relocate segments to coalesce holes
External Fragmentation

- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones

Compaction
- Relocate segments to coalesce holes
  - Copying eats up a lot of CPU time!
    - if 4 bytes in 10ns, 8 GB in 20s!

But what if a segment wants to grow?
Eliminating External Fragmentation: Swapping

- Preempt processes and reclaim their memory

- Move images of suspended processes to backing store

Diagram:
- Ready queue
- Running
- Waiting
- Suspended queue
- Semaphores/condition queues
- OS
- p1
- p2
- swap in
- swap out
Eliminating External Fragmentation: Tiling Memory

Virtual \((P_1)\)

Physical
Virtual ($P_1$)
Virtual (P₁)
Virtual (P₁)

Physical

Tiling Memory
Tiling Memory

Virtual (P₁)

Physical

page

frame
Tiling Memory

Physical

<table>
<thead>
<tr>
<th>3</th>
<th>43</th>
<th>81</th>
<th>44</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>6</td>
<td>42</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>82</td>
<td>83</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Tiling Memory

<table>
<thead>
<tr>
<th>Physical</th>
<th>83</th>
<th>84</th>
<th>2</th>
<th>3</th>
<th>81</th>
<th>44</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>43</td>
<td>81</td>
<td>44</td>
<td>45</td>
<td>85</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>5</td>
<td>85</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>6</td>
<td>42</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>47</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### P2

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
</tr>
</tbody>
</table>

### P1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>81</td>
<td>82</td>
<td>83</td>
</tr>
</tbody>
</table>
Eliminating External Fragmentation: Paging

- Allocate VA & PA memory in chunks of the same, fixed size (pages and frames, respectively)

- Adjacent pages in VA (say, within the stack) need not map to contiguous frames in PA!
  - Free frames can be tracked using a simple bitmap
    - 001111100111101110000 one bit/frame
  - No more external fragmentation!
  - But now internal fragmentation (you just can't win...)
    - when memory needs are not a multiple of a page
    - typical size of page/frame: 4KB to 16KB
How can I reference a byte in VA space?
Virtual address

- Interpret VA as comprised of two components
  - page: which page?
  - offset: which byte within that page?
Virtual address

Interpret VA as comprised of two components

- **page**: which page?
  - no. of bits specifies no. of pages are in the VA space
- **offset**: which byte within that page?
Virtual address

- Interpret VA as comprised of two components
  - **page**: which page?
    - no. of bits specifies no. of pages are in the VA space
  - **offset**: which byte within that page?
    - no. of bits specifies size of page/frame
Virtual address

To access a byte

- extract page number
- map that page number into a frame number using a page table
  - Note: not all pages may be mapped to frames
- extract offset
- access byte at offset in frame
Basic Paging

The Page Table
- lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR value saved/restored in PCB on context switch

CPU

Page Table

Frame

Physical Memory

PTBR

The Page Table too needs to live in memory!
Basic Paging

The Page Table
- lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR value saved/restored in PCB on context switch

The Page Table too needs to live in memory!

Helps implement mapping...
## Page Table Entries

- **Frame number**
- **Present (Valid/Invalid) bit**
  - Set if entry stores a valid mapping. If not, and accessed, page fault
- **Referenced bit**
  - Set if page has been referenced
- **Modified (dirty) bit**
  - Set if page has been modified
- **Protection bits (R/W/X)**

### Diagram

![Page Table Entries Diagram](image)

<table>
<thead>
<tr>
<th>Frame</th>
<th>Protection bits (R/W/X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Physical Memory

<table>
<thead>
<tr>
<th>Protection</th>
<th>Modified</th>
<th>Referenced</th>
<th>Other</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Present</td>
<td>11</td>
<td>2</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

---

59