Recall: Multithreaded Processes

Thread(s) + Shared Address Space

Multiple stacks in one address space
Recall: High-Level Sync

Semaphores – integers with restricted interface
- P() / down: Wait until non-zero then decrement
- V() / up: Increment and wake sleeping task

Monitors: Lock + Condition Variables
- Shared state determines when to wait
- Condition variables to wait "within critical section" on shared state changing
- Wait(), Signal(), Broadcast()
Thread Correctness Problems

Many famous bugs from synchronization bug:
- Therac-25 – Radiation therapy machine had race condition causing overdose and death
- Space shuttle launch aborted because of timing bug

Really hard to find synchronization bugs
- Can't test
- Nondeterministic errors
- Some tools, but very underdeveloped
- "Too much" synchronization – deadlock
Avoiding Threads/Synchronization

Multiple processes
  – Use files?
  – Still synchronization

Non-preemptive threads

Event-based programming
Avoiding Threads/Synchronization

Multiple processes
  - Use files?
  - Still have synchronization

Non-preemptive threads

Event-based programming

Can't use multiple cores
Avoiding Threads/Synchronization

Multiple processes
  - Use files?
  - Still synchronization

Non-preemptive threads

Event-based programming
Recall: Threading Models

Option A: User-level library, one kernel thread (1:N)
- Early Java, ...
- Library does thread context switch
- Kernel time slices between processes, e.g. on I/O

Option B: User-level library, multiple kernel threads (M:N)
- Solaris <9, FreeBSD <7, ...
- Kernel still time slices on I/O

Option C: Scheduler activations, kernel thread/CPU (M:N + upcalls)
- NetBSD < 5, Windows (option), ...

Option D: Kernel thread per user thread (1:1)
- Linux, OS X, Windows (default), FreeBSD >= 7, Solaris >= 7, NetBSD >= 5
Threads with explicit switching

Our problems come from **arbitrary interleavings**

Let's *only context switch when program tells us to*

Example: In a web server, when reading/writing from the network

No need for special synchronization functions
Problems with explicit switching

What if too long between switches?

What if a thread enters an infinite loop?

What if a thread blocks on IO without triggering context switching?
Alternatives to Threads

Multiple processes
  - Use files?
  - Still synchronization

Non-preemptive threads

Event-based programming
Event-driven programming

IO done while code is running
  − Function to get next external "event"

Example ATM Server:

```c
BankServer() {
    while(TRUE) {
        event = WaitForNextEvent();
        if (event == ATMRequest)
            StartOnRequest(event);
        else if (event == AcctAvail)
            ContinueRequest(event);
        else if (event == AcctStored)
            FinishRequest(event);
    }
}
```

POSIX support for this: select/poll
  − Function to wait for I/O from multiple sources
Event-driven programming

```c
BankServer() {
    while(TRUE) {
        event = WaitForNextEvent();
        if (event == ATMRequest)
            StartOnRequest(event);
        else if (event == AcctAvail)
            ContinueRequest(event);
        else if (event == AcctStored)
            FinishRequest(event);
    }
}
```

Same problems as threads with explicit switching:
- What if processing an event takes too long?
- What if processing an event does (blocking) IO?

Also less convenient to code:
- "next" line of code in another event handler
BankServer() {
    while (TRUE) {
        event = WaitForNextEvent();
        if (event == ATMRequest)
            StartOnRequest(event);
        else if (event == AcctAvail)
            ContinueRequest(event);
        else if (event == AcctStored)
            FinishRequest(event);
    }
}

But few synchronization problems, possibly lower overhead (Why?)

Pattern common in:
- Graphical user interface programming
- Some high-performance servers
Logistics

Homework 0 grades out – please check them
  – Tell us by *next week* if there's something wrong

Project 1 Design Reviews (today)

Project 1 Checkpoint 1 – Tommorrow 11:59PM

Homework 1 Due Wednesday 11:59PM
Break
Virtualizing Resources

Run multiple processes/thread
- Multiplex CPUs: scheduling/threads – just finished
- Multiplex memory (today) via address translation
- Multiplex devices (like disk) – later
Virtualizing Resources

Run multiple processes/thread
- Multiplex CPUs: scheduling/threads – just finished
- **Multiplex memory (today) via address translation**
- Multiplex devices (like disk) – later
Recall: Multithreaded Processes

Thread(s) + **Shared Address Space**

Multiple stacks in one address space
Memory Multiplexing Goals

Protection
- Private memory of other processes/OS remains private
- Even from *malicious* processes

Controlled overlap
- Want processes, OS to be able to use shared memory to communicate
- Or to save space

Uniform view of memory
- No need to change addresses in a program to run two copies of it
- No need to change a program to let it use more memory
Address Translation

Translation

- Program addresses are **virtual**
- Kernel-controlled mapping from virtual to physical
- Implemented by hardware

Preferred mechanism for memory multiplexing
Living Without Translation
Recall: Loading
Process view of memory

data1:   dw  32
            ...
start:   lw  r1,0(data1)
               jal  checkit
loop:    addi  r1, r1, -1
               bnz  r1, loop  ...
checkit: ...

Assume 4byte words
0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

Physical addresses
Assume 4byte words
0x0300 0000 0000
           ...
0x0900 8C2000C0
0x0904 0C000280
0x0908 2021FFFF
0x090C 14200242
           ...
0x0A00
Instructions and Data in Memory

Process view of memory

data1:    dw 32
          ...
start:   lw r1,0(data1)
jal checkit
loop:    addi r1, r1, -1
bnz r1, loop  ...
checkit: ...

Assume 4byte words
0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

Physical addresses
Assume 4byte words
0x0300 0000 0000 0000
0x0900 8C20000C0
0x0904 0C000280
0x0908 2021FFFF
0x090C 14200242
0x0A00
Second copy?

Process view of memory

```
data1: dw 32
...  
start: lw r1,0(data1)
  jal checkit
loop:  addi r1, r1, -1
      bnz r1, r0, loop
...  
checkit: ...
```

Physical addresses

```
0x300  00000020  
...    ...        
0x900  8C2000C0  
0x904  0C000280  
0x908  2021FFFF  
0x90C  14200242  
...    ...        
0x0A00
```

Need address translation!
Recall: Generating Executables

Steps:
- Compile ("gcc")
- Link/Load ("ld")
- Execute (dynamic linker)

Addresses chosen at all of these steps

Want to avoid work at runtime...
Relocatable Libraries

Dynamic libraries – final linking postponed

Code compiled with only relative addresses
- MIPS: can't use jal

Table of pointers to library routines

Statically linked stubs
- Initial entries in table
- Locate dynamic library routine
- Replace entry in table
Recall: Uniprogramming

No translation and no protection
Application can access any physical address
Application always at same location (owns machine)

Reality of dedicated machine
Primitive Multiprogramming

No translation

Programs need to be at different addresses

Use linker-loader
  - Loader adjusts memory locations (links) the program
    - (Maybe program compiled to make this easier)

Common on early OSs
  - MacOS <= 9; Windows 3.x

No protection
Still no translation

Hardware support: BaseAddr + LimitAddr registers
- Access outside range → error (except in kernel mode)
- Change from kernel mode only

Kernel loads from PCB during switches
Address Translation

Each process, kernel have different address space
- address space = all the addresses a process can touch and their state

Two views of memory:
- View from the CPU – what the program sees (virtual)
- View from physical memory (physical)
- MMU is the hardware that converts between the two
Implementing **protection** with translation
- no virtual address names other process's private memory

Later:
- **Controlled sharing** (some addresses in common)
- **Uniform view** (same address every time)
Recall: General Address Translation

Translation Map 1

Translation Map 2

Physical Address Space
Recall: Simple Base and Bounds

Actually what the Cray-1 did

Add **base** to every address

Error if any address is greater than **bound**
Recall: Simple Base and Bounds

![Diagram of address translation]

Illusion of **dedicated machine** with memory starting at 0
and ending a limit

**No relocation (relinking)** at load-time
B&B: External Fragmentation

Not every process is the same size

Over time, memory *fragmented*

Example: To start process 11,

- **copy** process 9 + 10 to make space
- even though enough memory is free
B&B: Internal Fragmentation

Traditional MIPS memory layout:

Big hole where stack/dynamic data grows
  - Allocate 2GB to every process?

Would like to have unallocated space
B&B: Sharing

Want to share code to save space
  – Example: OS libraries

Want to share memory for communication
Virtual memory address specifies a **segment**
- E.g. code, data, stack, for memory sharing, ...

Each segment has base + bound
### Implementation of Segmentation

**Segment map** in processor

- Segment extracted from program address
- Base added to generate physical address
- Bound checked → error if out of range
Implementation of Segmentation

As many segments as entries

Segment number from:

- Top bits of virtual address (Multics), or
- Part of instruction (x86): mov [es:bx], ax
  - More later
Implementation of Segmentation

V/N = Valid/Not Valid

- (since can't have negative bound)
### Example: Four Segments

#### Virtual Address Format

<table>
<thead>
<tr>
<th>Seg ID</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

#### Segments Overview

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
<td>Y</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
<td>Y</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
<td>Y</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
<td>Y</td>
</tr>
</tbody>
</table>

#### Seg ID = 0

- Seg ID: 0
- Offset: 0x0000
- Space for Other Apps
- Might be shared

#### Seg ID = 1

- Seg ID: 1
- Offset: 0x4000
- Space for Other Apps
- Might be shared

#### Physical Address Space

- Offset: 0x0000
- Offset: 0x4800
- Offset: 0x5C00
- Offset: 0xF000
- Shared with Other Apps
External fragmentation:
- free space between allocations is not all together
- expensive moving

Internal fragmentation
- unused gaps within allocated chunks
- can't fix without relocating program

No sharing – waste space with extra copies

*Which of these problems does segmentation solve?*
Segmentation vs Base and Bounds

B&B problem: No sharing
- Segmentation can share
- Avoids extra copies required under base and bounds
- But only whole segments (how many libraries does a program use? will we run out of segments?)

![Physical Address Space Diagram]

- 0x0000
- 0x4000
- 0x4800
- 0x5C00
- 0xF000

Might be shared
Shared with Other Apps
Segmentation vs Base and Bounds

B&B problem: External Fragmentation
- Still a problem, may need to rearrange chunks in memory
- But smaller – less work
B&B problem: Internal fragmentation

- Segmentation can have large "holes" that don't use physical memory
- But not anywhere – still have some wasted space
- Example: stack grows to lower addresses from 0xE000
Swapping

Problem: Not all processes fit in memory
  – Breaks the *illusion of dedicated machine*

Extreme context switch:
  – when process is not running, move it to disk
Swapping with Segmentation: Out

Can we do better than moving whole process to disk?

What if process isn't using some segment?
  - Copy it to disk
  - Mark it as invalid in segment map
  - Give the space to some other process

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
<td>Y</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
<td>Y</td>
</tr>
<tr>
<td>2 (temp)</td>
<td>0xC000</td>
<td>0x1000</td>
<td>N</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
<td>Y</td>
</tr>
</tbody>
</table>
Swapping with Segmentation: In

What about when process uses segment again?

- Handle segmentation fault (valid = N)
- Make room
- Copy segment in
- Update segment map

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
<td>Y</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
<td>Y</td>
</tr>
<tr>
<td>2 (temp)</td>
<td>0xB800</td>
<td>0x1000</td>
<td>Y</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
<td>Y</td>
</tr>
</tbody>
</table>
Swapping: Discussion

Disks are **slow**
- SSDs are faster, but *not that much faster*

For modern systems, swapping performance is usually unacceptable

Why do we talk about it?
- Big motivation for **paging**
- Program loading uses this mechanism (load from disk on fault)
A better solution: paging

Eliminate external fragmentation with fixed sized chunks

- Called *pages*
- All holes are multiples of a page size
- All allocations are whole pages

Minimize internal fragmentation by making pages small

- Typical size: 4096 bytes
Paging overview

Page table: one per process (address space)
- Resides in **physical memory**
- Physical page #, permissions for each virtual page #
- Index into table is virtual page #
- Trigger **fault** (like bounds failure) if permissions wrong
Paging: Address mapping

Offset: copied from virtual addr to physical addr
- Example: 10-bit offset → 1024-byte pages

Virtual page # is all remaining bits
- 32-bit, 1024-byte pages: 22 bits → 4 M page table entries

Physical page # from table **replaces** virtual page # to form physical addr
- May be different length
Address Translation: Overhead

Base and Bound
- Two registers

Segmentation
- One segment table entry per segment
- Registers for segment table?

Paging
- Extra memory access for every memory access
- 4K pages $\rightarrow$ 1M entries for 32-bit address space!
Summary: Event-driven programming

Key idea: loop calling function to "get next event"

Need to break up "straight" threaded code into many chunks
  - If chunks block – trouble
  - If chunks take too long – trouble

But:
  - No synchronization – correctness easier
  - Maybe less overhead (no extra stacks)
Summary: Address Translation

Goals

Protection
- programs can only access physical memory they are allowed to

Controlled Sharing
- programs can share memory to communicate or to save resources

Uniform view
- programs have the same virtual addresses every time

Avoid fragmentation
- allow efficient allocation/deallocation
Summary: Base and Bound

Address translation with two special registers:
- Base – added to every virtual address
- Bound – checked against every virtual address

Protection

Controlled sharing

Uniform view (???)
- Not if program is resized

Avoid fragmentation
Summary: Segmentation

Address translation with segment table
  - Each entry has base, bound
  - Segment number (index into table) extracted from address (or elsewhere)

Protection

Controlled sharing
  - but limited number of segments to share

Uniform view

Avoid fragmentation
Summary: Paging

Address translation with page table
- Each entry has *small*, fixed size
- Page frame number (index into table) extracted from address (or elsewhere)

*Protection*

*Controlled sharing*

*Uniform view*

*Avoid fragmentation*