Recall: Four requirements for occurrence of Deadlock

- **Mutual exclusion**
  - Only one thread at a time can use a resource.
- **Hold and wait**
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads.
- **No preemption**
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it.
- **Circular wait**
  - There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads
  - \( T_1 \) is waiting for a resource that is held by \( T_2 \)
  - \( T_2 \) is waiting for a resource that is held by \( T_3 \)
  - \( \ldots \)
  - \( T_n \) is waiting for a resource that is held by \( T_1 \)

Recall: Banker’s Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:
    - \((\text{available resources} - \#\text{requested}) \geq \text{max remaining that might be needed by any thread})
- Banker’s algorithm (less conservative):
  - Allocate resources dynamically
    - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
      \([\text{Maxnode}] - [\text{Allocnode}] \leq [\text{Avail}]\)
        for \([\text{Requestnode}] \leq [\text{Avail}]\)
  - Grant request if won’t prevent some thread from allocating its maximum and finishing
- Keeps system in a “SAFE” state:
  - There exists a sequence \( \{T_1, T_2, \ldots, T_n\} \) with \( T_1 \) requesting all remaining resources, finishing, then \( T_2 \) requesting all remaining resources, etc..

Recall: Banker’s Algorithm Example

- Banker’s algorithm with dining lawyers
  - “Safe” (won’t cause deadlock) if when try to grab chopstick either:
    - Not last chopstick
    - Is last chopstick but someone will have two afterwards

- What if k-handed lawyers? Don’t allow if:
  - It’s the last one, no one would have k
  - It’s 2nd to last, and no one would have k-1
  - It’s 3rd to last, and no one would have k-2
  - \( \ldots \)
Virtualizing Resources

• Physical Reality:
  Different Processes/Threads share the same hardware
  – Need to multiplex CPU (just finished: scheduling)
  – Need to multiplex use of Memory (starting today)
  – Need to multiplex disk and devices (later in term)
• Why worry about memory sharing?
  – The complete working state of a process and/or kernel is defined by its data in
    memory (and registers)
  – Consequently, cannot just let different threads of control use the same memory
  » Physics: two different pieces of data cannot occupy the same locations in memory
  – Probably don’t want different threads to even have access to each other’s memory if
    in different processes (protection)

Important Aspects of Memory Multiplexing

• Protection:
  – Prevent access to private memory of other processes
    » Different pages of memory can be given special behavior (Read Only, Invisible to
      user programs, etc).
    » Kernel data protected from User programs
    » Programs protected from themselves
• Translation:
  – Ability to translate accesses from one address space (virtual) to a different one
    (physical)
  – When translation exists, processor uses virtual addresses, physical memory
    uses physical addresses
  – Side effects:
    » Can be used to avoid overlap
    » Can be used to give uniform view of memory to programs
• Controlled overlap:
  – Separate state of threads should not collide in physical memory. Obviously,
    unexpected overlap causes chaos!
  – Conversely, would like the ability to overlap when desired (for communication)

Alternative View: Interposing on Process Behavior

• OS interposes on process’ I/O operations
  – How? All I/O happens via syscalls.
• OS interposes on process’ CPU usage
  – How? Interrupt lets OS preempt current thread

Question: How can the OS interpose on process’ memory accesses?
  – Too slow for the OS to interpose every memory access
  – Translation: hardware support to accelerate the common case
  – Page fault: uncommon cases trap to the OS to handle

Recall: Four Fundamental OS Concepts

• Thread: Execution Context
  – Fully describes program state
  – Program Counter, Registers, Execution Flags, Stack
• Address space (with or w/o translation)
  – Set of memory addresses accessible to program (for read or write)
  – May be distinct from memory space of the physical machine
    (in which case programs operate in a virtual address space)
• Process: an instance of a running program
  – Protected Address Space + One or more Threads
• Dual mode operation / Protection
  – Only the “system” has the ability to access certain resources
  – Combined with translation, isolates programs from each other and
    the OS from programs
THE BASICS: Address/Address Space

- What is $2^{10}$ bytes (where a byte is abbreviated as “B")?
  - $2^{10} \text{B} = 1024 \text{B} = 1 \text{ KB}$ (for memory, $1 \text{K} = 1024$, not $1000$)

- How many bits to address each byte of 4KB page?
  - $4 \text{KB} = 4 \times 1 \text{KB} = 4 \times 2^{10} = 2^{12}$ bits

- How much memory can be addressed with 20 bits? 32 bits? 64 bits?
  - Use $2^k$

Address Space, Process Virtual Address Space

- Definition: Set of accessible addresses and the state associated with them
  - $2^{32} = \sim 4 \text{ billion bytes}$ on a 32-bit machine

- How many 32-bit numbers fit in this address space?
  - $32\text{-bits} = 4 \text{ bytes}$, so $2^{32}/4 = 2^{30} = \sim 1\text{ billion}$

- What happens when processor reads or writes to an address?
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    » (Memory-mapped I/O)
  - Causes program to abort (segfault)?
  - Communicate with another program
  - …

Recall: Process Address Space: typical structure

Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address

- Application given illusion of dedicated machine by giving it reality of a dedicated machine
**Primitive Multiprogramming**

- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads
  
  - Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - Everything adjusted to memory location of program
    - Translation done by a linker-loader (relocation)
    - Common in early days (... till Windows 3.x, 95?)

  - With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

---

**Binding of Instructions and Data to Memory**

### Process view of memory

```
start: lw r1,0(data1)
      jal checkit
loop:  addi r1, r1, -1
       bnez r1, loop
checkit: ...
```

### Physical addresses

```
0x00000000
0x00000002
...
```

---

**Second copy of program from previous example**

### Process view of memory

```
start: lw r1,0(data1)
      jal checkit
loop:  addi r1, r1, -1
       bnez r1, loop
checkit: ...
```

### Physical addresses

```
0x0000 00000020
0x0800 00000020
0x0900 08C000C0
0x0904 08C08280
0x0908 2021FFFF
0x090C 14200242
...
```

---

**Need address translation!**

- Assume 4-byte words
- $0x300 = 4 \times 0x0C0$
- $0x0C0 = 0000\text{1100} 0000$
- $0x300 = 0011\text{0000} 0000$

---

**Physical Memory**

- App X

---

**Physical Memory**

- $0xFF00$

---

**Physical Memory**

- $0xFFFF$

---

**Physical Memory**

- $0x0000$

---

**Physical Memory**

- $0x0000$

---

**Physical Memory**

- $0x0000$

---

**Physical Memory**

- $0x0000$

---

**Physical Memory**

- $0x0000$
Second copy of program from previous example

From Program to Process

- Preparation of a program for execution involves components at:
  - Compile time (i.e., "gcc")
  - Link/Load time (UNIX "ld" does link)
  - Execution time (e.g., dynamic libs)

- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system

- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code (i.e. the stub), locates appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine

Administivia

- Midterm 2: Thursday 3/14 from 8-10PM
  - A week from tomorrow!!!
  - All material up to Lecture 16 technically in bounds
  - Closed book: with two double-sided handwritten sheets of notes

- Homework 4 coming out
  - Released tomorrow, Wednesday 3/6

- Project 2 design document due this Friday!

- Starting next week – will have an opportunity to get extra credit participation points by attending lecture
  - Details to follow

Administivia (Con’t)

- You need to know your units as CS/Engineering students!
- Units of Time: "s": Second, "min": 60s, "h": 3600s, (of course)
  - Millisecond: 1ms \(\Rightarrow 10^{-3}\) s
  - Microsecond: 1us \(\Rightarrow 10^{-6}\) s
  - Nanosecond: 1ns: \(\Rightarrow 10^{-9}\) s
  - Picosecond: 1ps

- Integer Sizes: "b" \(\Rightarrow \text{bit}, \ B \Rightarrow \text{byte} = 8\text{ bits, } W \Rightarrow \text{word} = ? \) (depends. Could be 16b, 32b, 64b)

- Units of Space (memory), sometimes called the "binary system"
  - Kilo: 1KB \(\Rightarrow 1024\) bytes \(= 2^{10}\) bytes \(= 1024 = 1.0 \times 10^3\)
  - Mega: 1MB \(\Rightarrow (1024)^2\) bytes \(= 2^{20}\) bytes \(= 1,048,576 = 1.0 \times 10^6\)
  - Giga: 1GB \(\Rightarrow (1024)^3\) bytes \(= 2^{30}\) bytes \(= 1,073,741,824 = 1.1 \times 10^9\)
  - Tera: 1TB \(\Rightarrow (1024)^4\) bytes \(= 2^{40}\) bytes \(= 1,099,511,627,776 = 1.1 \times 10^{12}\)
  - Peta: 1PB \(\Rightarrow (1024)^5\) bytes \(= 2^{50}\) bytes \(= 1,125,899,068,424,624 = 1.1 \times 10^{15}\)
  - Exa: 1EB \(\Rightarrow (1024)^6\) bytes \(= 2^{60}\) bytes \(= 1,152,921,504,606,846,976 = 1.2 \times 10^{18}\)

- Units of Bandwidth, Space on disk/etc. Everything else..., sometimes called the “decimal system”
  - Kilo: 1KB/s \(\Rightarrow 10^3\) bytes/s, 1KB \(\Rightarrow 10^3\) bytes
  - Mega: 1MB/s \(\Rightarrow 10^6\) bytes/s, 1MB \(\Rightarrow 10^6\) bytes
  - Giga: 1GB/s \(\Rightarrow 10^9\) bytes/s, 1GB \(\Rightarrow 10^9\) bytes
  - Tera: 1TB/s \(\Rightarrow 10^{12}\) bytes/s, 1TB \(\Rightarrow 10^{12}\) bytes
  - Peta: 1PB/s \(\Rightarrow 10^{15}\) bytes/s, 1PB \(\Rightarrow 10^{15}\) bytes
  - Exa: 1EB/s \(\Rightarrow 10^{18}\) bytes/s, 1EB \(\Rightarrow 10^{18}\) bytes
Multiprogramming with Protection

- Can we protect programs from each other without translation?
  - Yes: Base and Bound!
  - Used by, e.g., Cray-1 supercomputer

Recall: Base and Bound (No Translation)

- Still protects OS and isolates program
- Requires relocating loader
- No addition on address path

Recall: General Address translation

- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (Memory Management Unit or MMU) converts between two views
- Translation ⇒ much easier to implement protection!
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space

Recall: Base and Bound (with Translation)

- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?
### Issues with Simple B&B Method

- **Fragmentation problem over time**
  - Not every process is same size → memory becomes fragmented over time
  - Fragmentation: wasted space both **external** (between blocks) and **internal** (inside blocks)
- **Missing support for sparse address space**
  - Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc)
- **Hard to do inter-process sharing**
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

### More Flexible Segmentation

- **Logical View**: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

### Implementation of Multi-Segment Model

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: mov [es:bx],ax.
- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well

### Intel x86 Special Registers

- **Typical Segment Register**
  - Current Priority is RPL of Code Segment (CS)
  - Segmentation can’t be just "turned off"
  - What if we just want to use paging?
  - Set base and bound to all of memory, in all segments
Let's simulate a bit of this code to see what happens (PC=0x240):
1. Fetch 0x0240 (0 0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 → $a0, Move PC+4 → PC

2. Fetch 0x0244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

3. Fetch 0x0360. Translated to Physical=0x4360. Get "li $v0, 0"
   Move 0x0364 → $t0, Move PC+4 → PC

Example of Segment Translation (16bit address)
Observations about Segmentation

- Translation on every instruction fetch, load or store
- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
- When it is OK to address outside valid range?
  - This is how the stack (and heap?) allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
- What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called “swapping”)

What if not all segments fit in memory?

- Extreme form of Context Switch: Swapping
  - To make room for next process, some or all of the previous process is moved to disk
  - Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- What might be a desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory

Problems with Segmentation

- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don’t need all memory within allocated chunks

Recall: General Address Translation

Prog 1
Virtual Address Space 1

Translation Map 1

Physical Address Space

Prog 2
Virtual Address Space 2

Translation Map 2
Paging: Physical Memory in Fixed Size Chunks

• Solution to fragmentation from segments?
  – Allocate physical memory in fixed size chunks ("pages")
  – Every chunk of physical memory is equivalent
    » Can use simple vector of bits to handle allocation:
      \[0110011100011010\] \[110010\]
    » Each bit represents page of physical memory
      \(1 \Rightarrow \text{allocated}, \ 0 \Rightarrow \text{free}\)

• Should pages be as big as our previous segments?
  – No: Can lead to lots of internal fragmentation
    » Typically have small pages (1K-16K)
  – Consequently: need multiple pages/segment

How to Implement Simple Paging?

Virtual Address:

- Page Table (One per process)
  – Resides in physical memory
  – Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
- Virtual address mapping
  – Offset from Virtual address copied to Physical Address
  » Example: 10 bit offset \(\Rightarrow 1024\)-byte pages
  – Virtual page # is all remaining bits
  » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
  – Physical page # copied from table into physical address
  – Check Page Table bounds and permissions

Simple Page Table Example

Example (4 byte pages)

Virtual Address: (Process A):

What about Sharing?

Virtual Address: (Process B):

Shared Page

This physical page appears in address space of both processes
Where is page sharing used?

- The "kernel region" of every process has the same page table entries
  - The process cannot access it at user level
  - But on U→K switch, kernel code can access it AS WELL AS the region for THIS user
    » What does the kernel need to do to access other user processes?
- Different processes running same binary!
  - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
  - Can actually share objects directly between processes
    » Must map page into same place in address space!
  - This is a limited form of the sharing that threads have within a single process

Some simple security measures

- Address Space Randomization
  - Position-Independent Code → can place user code anywhere in address space
    » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
  - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
  - Don’t map whole kernel space into each process, switch to kernel page table
  - Meltdown→map none of kernel into user mode!

Memory Layout for Linux 32-bit (Pre-Meltdown patch!)

Summary: Paging
### Summary: Paging

- **Virtual memory view**
  - Data
  - Code
  - Stack
  - Heap

- **Physical memory view**
  - Data
  - Code
  - Stack
  - Heap

- **Page Table**
  - Page number
  - Offset

### Conclusion

- **Segment Mapping**
  - Segment registers within processor
  - Segment ID associated with each access
    - Often comes from portion of virtual address
    - Can come from bits in instruction instead (x86)
  - Each segment contains base and limit information
    - Offset (rest of address) adjusted by adding base

- **Page Tables**
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory

- **Next Time: Multi-Level Tables**
  - Virtual address mapped to series of tables
  - Permit sparse population of address space