Announcements

- Today's office hour is 2:30–3:30pm in 2203 Watershed Sciences
- Lab Exercise 1's due date is changed to Monday, April 25

Memory Management

Goal

- Memory must be shared
  - CPU gains related to scheduling require that many processes be in memory
- Which memory management scheme to use depends on the hardware available

How Programs Interact with Memory

steps	results	addresses
write program	source code	symbolic
compile, assemble	object module	relocatable addresses (*eg*, *bigmod* + 4)
linker	load module	
loader	in-core image	bind relocatable addresses to physical addresses
execute		

Program Execution

- Uses absolute or physical addresses
- Instruction execution cycle
  - fetch instruction at address *a*
  - decode it
  - fetch operands at addresses $b_1, \ldots, b_n$
    - May not be necessary
  - execute instructions
  - store results at addresses $d_1, \ldots, d_m$
- Memory unit sees stream of addresses
  - Does not know how stream is generated
  - We just care about the sequence

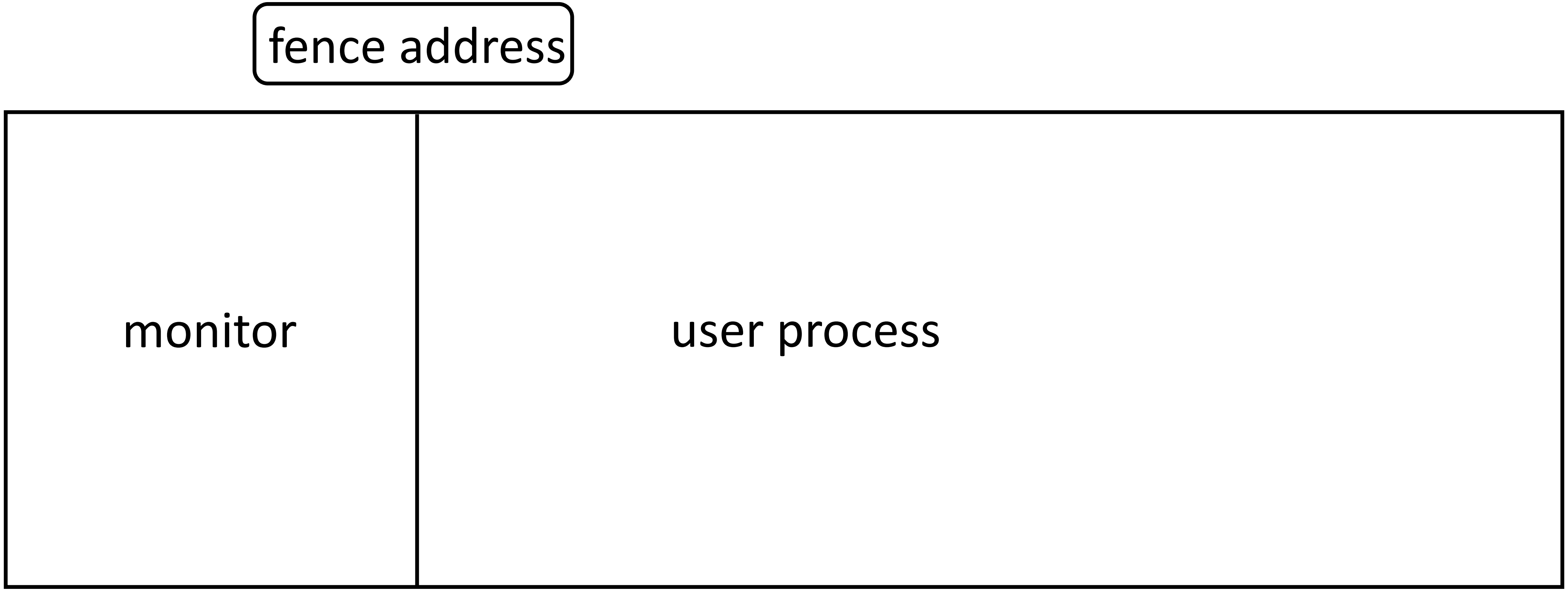
Memory Management: Bare Machine

- Used in dedicated systems where simplicity, flexibility are required
  - For example, IoT sensors
- No memory management
- No operating system software
- No special hardware
- No services
- Processes run withlout constraints beylond the physical hardware

Memory Management with Resident Monitor

- Two sections of memory
  - One for the monitor, usually in low memory because the interrupt/trap vectors are there
  - One for the user process
- Hardware protects monitor from user
  - Usually done with a fence address

Fence Register



How It Is Used

- Every user process memory reference checked against fence address
  - If it crosses into the monitor area, the user process is stopped
- Comparison not done when in monitor mode
- Good hardware design overlaps comparison with other activities
  - This reduces memory access time

Specifying Fence Address

- Build it into the hardware as a constant
  - How do you select it?
  - What happens if the monitor changes size?
- Put the fence address into a register
  - Register is called a *fence register*
  - Always used in address bounds check
  - This register loaded in monitor mode and requires a special privileged instruction

Relocation

- Monitor located in low address space
- Program’s addresses all begin with address 0
  - Need to relocate it to another address
  - So map address 0 into the first address after the fence address

When to Map Addresses

- At compile time
  - Need to know fence address
  - Problem: if fence address changes, code must be recompiled
- At load time
  - Computer generates relocatable code
  - Problem: if fence address changes, code must be reloaded

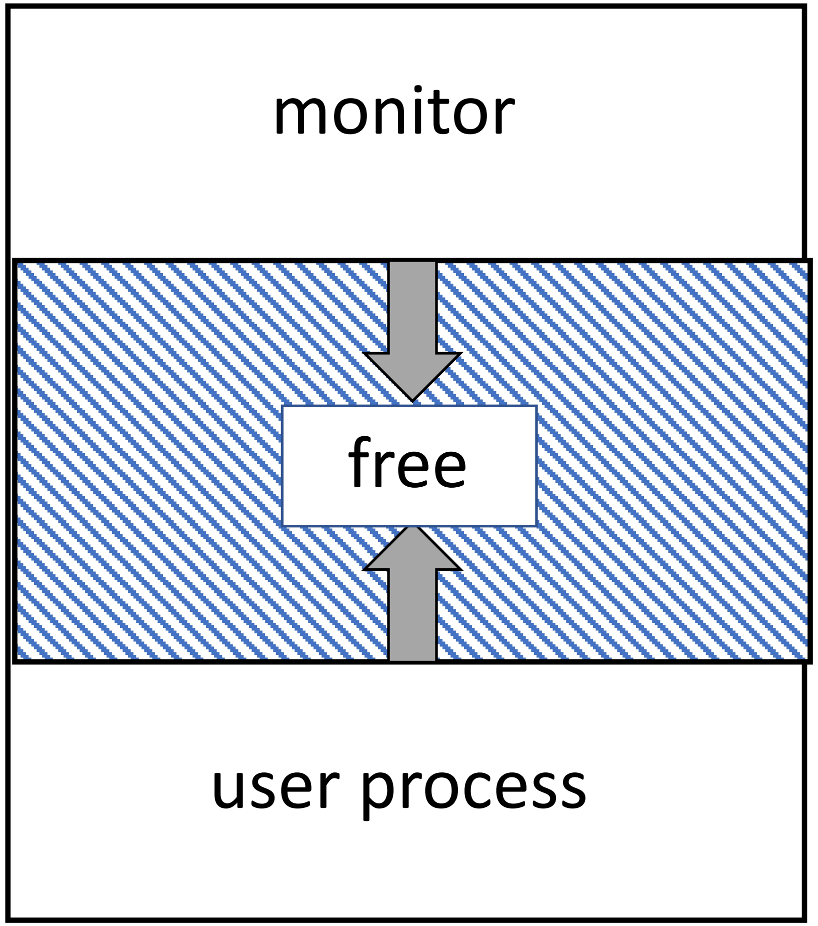
An Assumption

- Fence register does not change during execution
  - So it cannot change when any program is running
- Problem: transient monitor code
  - As code loaded into monitor, it grown
  - As code removed from monitor, it shrinks
  - Very useful when some monitor routines are rarely used
- But neither of the two previous methods allow this

How to Get Around This

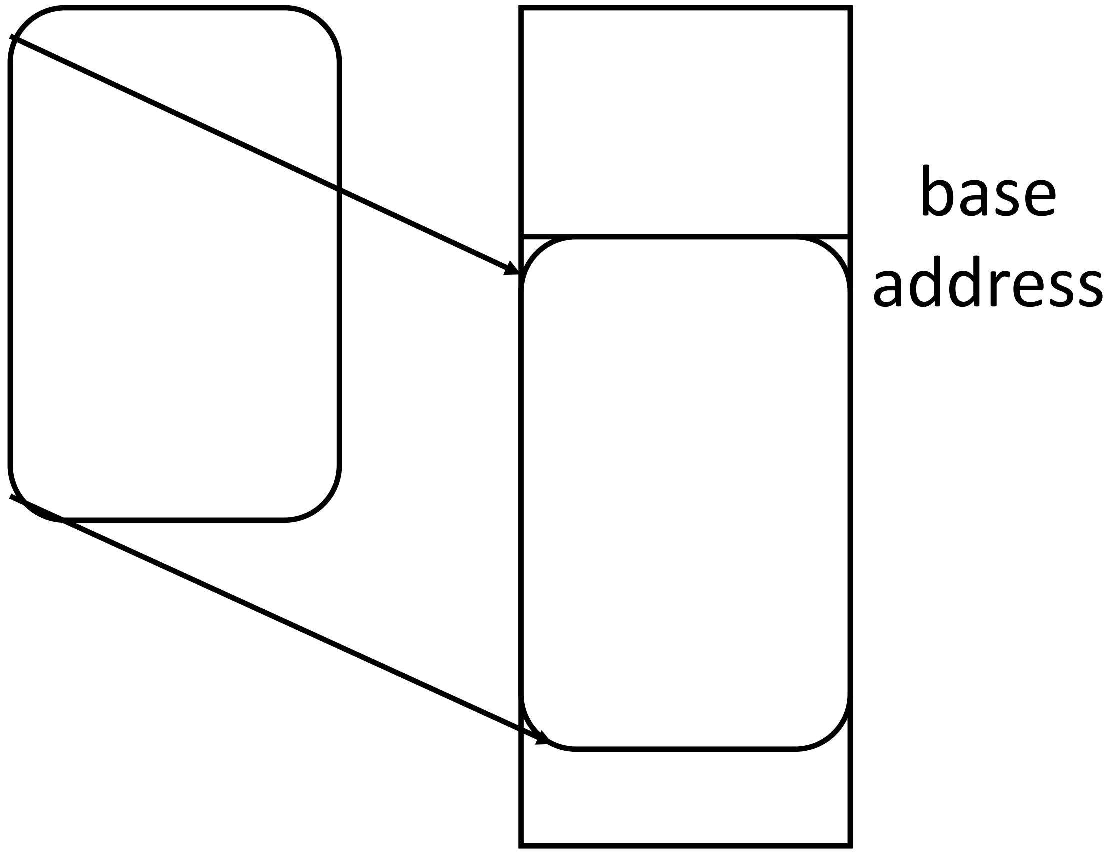
- Load user program in high memory so it grows down towards the fence register
- Monitor, user program can use space between

Used in early operating systems for the PDP-11



How to Get Around This

- Bind virtual addresses to physical ones at execution time
- Fence register (here, called a *base* or *relocation* register) value added to every address reference
- Example: base register contains address 36980, so when address x is referenced, it is translated into the physical address x + 36980
- Used in CDC 6600 sequence of computers



Advantages

- User process never sees the physical addresses
- If base register changed, user memory need only be moved to the correct locations relative to the new base register
- User process sees addresses as 0, …, *max*, but physical addresses are *b*, …, *b*+*max* where *b* is the value in the base register
- Note all information given to the program or operating system for use as memory addresses *must* be relocated
  - Example: buffers for I/O
- Concept of logical address space being mapped to separate physical address space is *central* to proper memory management

How to Get Around This

- Use a resident monitor
- One process resident at a time, all others moved to secondary storage (*swap device*)
  - Most useful when not enough memory to have multiple processes in memory
- Later generalized to many resident processes
  - Called *swapping*
  - System needs a swap device with enough space to hold copies of memory images for all user processes and provide quick access to them

Process Execution

- CPU calls dispatcher, which selects next process to execute
- If dispatcher finds process in memory, it runs the process
- If dispatcher doesn't find process in memory:
  - It swaps out the resident process
  - It swaps in the selected process
  - It loads registers as normal
  - Begin yo execute!

Swap Time

- Swapping greatly increases time for context switching
- Make execution time per process long relative to swap time
- For the following, assume:
  - Process is 20,000 words of memory
  - Swap device has 10ms   rotational latency
  - transfer rate is 363,000 words
- Time to move a process into or out of memory:

$$10\text{ms} + (20000/363000)\text{ sec} = 10\text{ms} + 55.1\text{ ms} = 65.1\text{ ms}$$

or 130ms to replace a process with another

Optimizations

- Swap only part of the in-memory process
  - Processes must keep monitor informed of changes in memory requirements
- Speed up secondary storage performance
  - Example: hard drive has transfer rate of 150MB/sec, rotational latency 4ms

    4ms + 20000/150000000sec = 4ms + 0.13 ms = 4.13 ms
  - Example: SSD has transfer rate of 520 MB/sec, latency 0.25ms

    0.25ms + 20000/520000000sec = 0.25ms + 0.04ms = 0.29ms

Optimizations

- Overlap swapping with process execution:
    1. move contents of user process area to swap-out buffer
    2. move contents of swap-in buffer to user area
    3. begin I/O to write swap-out buffer to swap device
    4. begin I/O to read next process being swapped in to swap-in buffer
    5. execute user process

Optimizations

- Only completely idle processes can be swapped; so if process blocked on I/O and I/O operations access the process buffers directly, can't swap processes
- Solutions:
  - Never swap a process with I/O pending
  - Have all I/O operations move data into or out of operating system buffers only and then transfer the data to or from disk

Simple Memory Management

- Arose from desire to avoid swapping
- Define multiple partitions of memory
  - Multiple processes stored in memory simultaneously, each at a different location
  - Issue: how to allocate memory so processes need not be swapped out
- Each process placed in contiguous memory
  - Multiple contiguous fixed partition allocation (MFT)
  - Multiple contiguous variable partition allocation (MVT)

Hardware Requirements

- Hardware prevents access outside assigned messages
- Bounds registers
  - Keep track of uppermost, lowermost physical addresses of partition
- Base and limit registers
  - Keep track of lowest physical address (base) and highest logical address (limit)

Fixed Size Regions (MFT)

Regions are of fixed size and do not change size