Comp 204: Computer Systems and Their Implementation

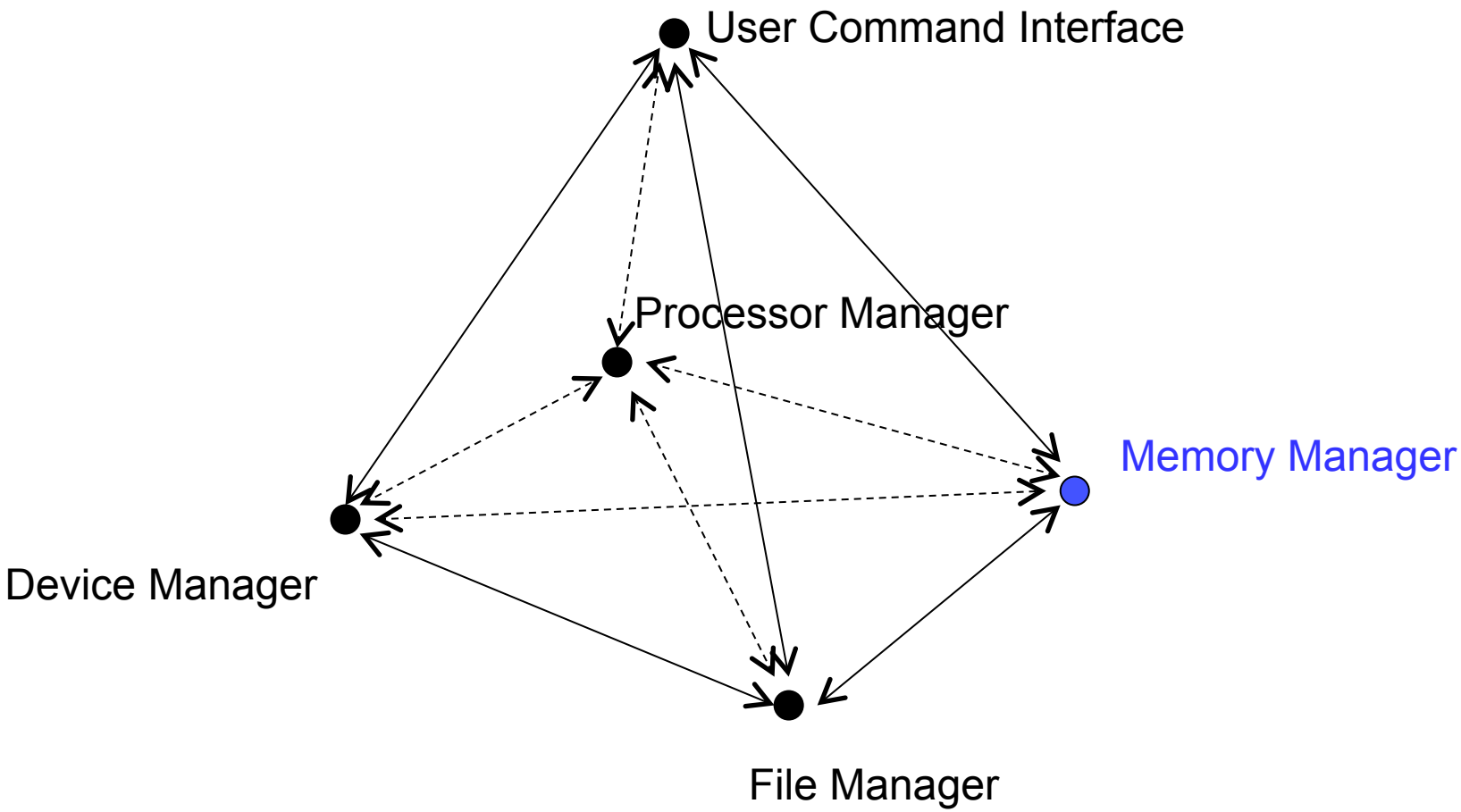
Lecture 13: Simple Memory Management Systems

Today

- Memory management
  - Number systems and bit manipulation
  - Addressing

- Simple memory management systems
  - Definition
  - Issues
  - Selection policies

Operating System – An Abstract View



Recap - Binary

- Starting from right, each digit is a power of two
  - (1,2,4,8,16…)
    - 110110 = 2+4+16+32 = 54
- To convert decimal to binary, keep subtracting largest power of 2
  - e.g. 37 (sub. 32, sub. 4, sub 1) = 100101
- n bits can represent 2^n numbers
  - range 0 to $2^n - 1$
- Conversely, to represent n numbers, need $\log_2(n)$ bits

Recap - Octal and Hex

- Octal is base 8 (digits 0-7)
  - 134 (oct) = (1*64) + (3*8) + (1*4) = 92
- To convert between octal and binary
  - think in groups of 3 bits (since 8 = 2^3)
  - 134(oct) = 001 011 100
  - 10111010 = 272 (oct)
- Hex is base 16 (A-F = 10-15)
  - B3 (hex) = (11*16) + (3*1) = 179
- Conversion to/from binary
  - using groups of 4 bits (since 16 = 2^4)
  - B3 (hex) = 1011 0011
  - 101111 = 2F (hex)

Recap - Bit Manipulation

- Use AND (&) to mask off certain bits
  - x = y & 0x7 // put low 3 bits of y into x
- Use left and right shifts as necessary
  - x = (y & 0xF0) >> 4 // put bits 4-7 of y into bits 0-3 of x
- Can also test if a bit is set
  - if (x & 0x80)… // if bit 7 of x is set…

('0x' states that a number is in hexadecimal)

Recap - Bit Manipulation

- Can switch a bit off
  - x = x & 0x7F // unset bit 7 of x (assume x is only 8 bits)
- Use OR (|) to set a bit
  - x = x | 0x80 // set bit 7 of x
- A right shift is divide by 2; left shift is multiply by 2
  - 6 << 1 = 0110 << 1 = 1100 = 12
  - 6 >> 1 = 0110 >> 1 = 0011 = 3

Memory Management

- A large-scale, multi-user system may be represented as a set of sequential processes proceeding in parallel
- To execute, a process must be present in the computer's memory
- So, memory has to be shared among processes in a sensible way

Memory

- Memory: a large array of words or bytes, each with its own address
- The value of the program counter (PC) determines which instructions from memory are fetched by the CPU
  - The instructions fetched may cause additional loading/storage access from/to specific memory addresses
- Programs usually reside on a disk as a binary executable file
- In order for the program to be executed it must be brought into memory and placed within a process
- When the process is executed it accesses data and instructions from memory, then upon termination its memory space is declared available

Address Binding – Compile Time

- Programs often generate addresses for instructions or data, e.g.

```
START:    CALL FUN1
                .
                .
          LOAD NUM
          JUMP START
```

- Suppose assemble above to run at address 1000, then jump instruction equates to JUMP #1000
- Consider what happens if we move program to another place in memory
- Obvious disadvantage for multiprogrammed systems
- Fixed address translation like this is referred to as compile-time binding

Load-Time Binding

- Ideally, would like programs to run anywhere in memory
- May be able to generate position-independent code (PIC)
  - aided by various addressing modes, e.g.
    PC-relative: JUMP +5
    Register-indexed: LOAD (R1) #3
- If not, binary program file can include a list of instruction addresses that need to be initialised by loader

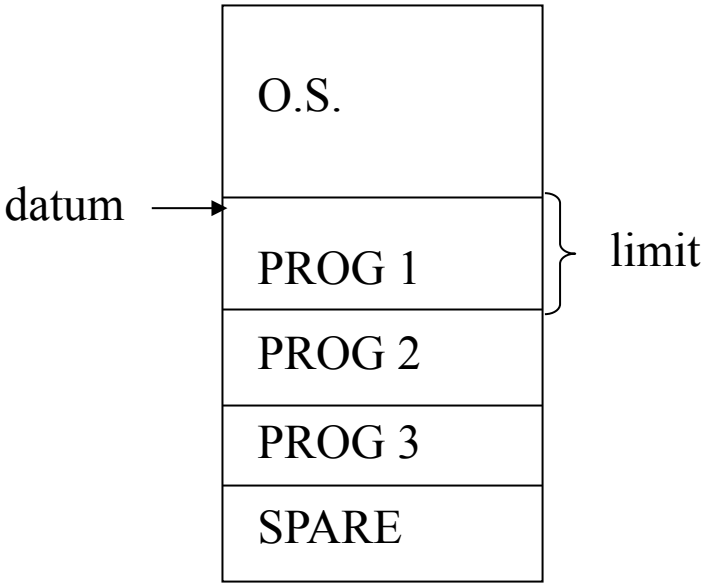
Dynamic (Run-Time) Binding

- Used in modern systems
- All programs are compiled to run at address zero
- For program with address space of size N, all addresses are in range 0 to N-1
- These are logical (virtual) addresses
- Mapping to physical addresses handled at run-time by CPU's memory management unit (MMU)
- MMU has relocation register (base register) holding start address of process
  - Contents of registers are added to each virtual address

Logical and Physical Addresses

- Addresses generated by the CPU are known as logical (virtual) addresses
  - The set of all logical addresses generated by a program is known as the logical address space
- The addresses seen by the MMU are known as physical addresses
  - The set of all physical addresses corresponding to the logical addresses is known as the physical address space
- The addresses generated by compile-time binding and load-time binding result in the logical and the physical addresses being the same
- Run-time binding results in logical and physical addresses that are different

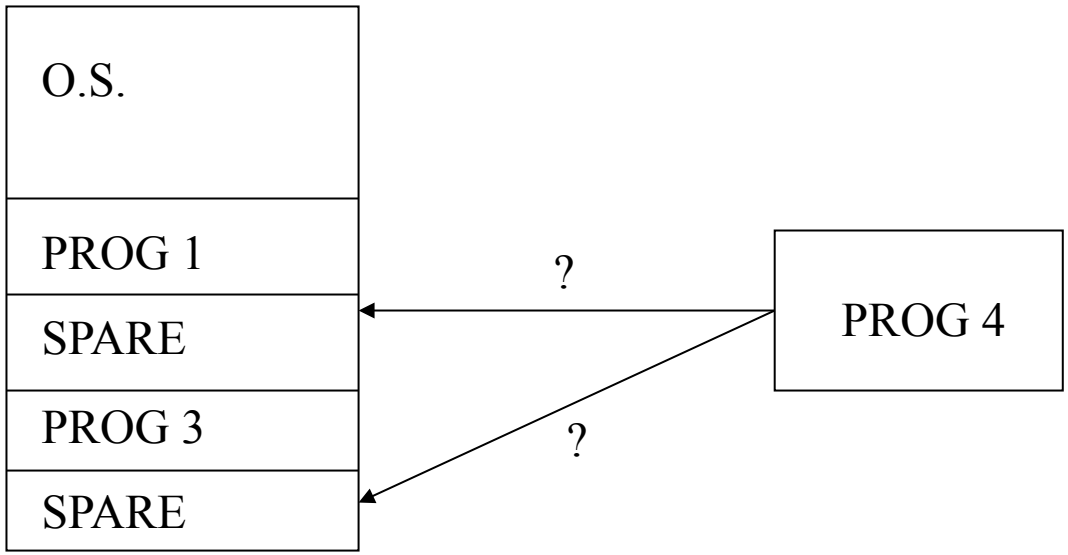
Simple System Store Management



- Store is allocated to programs in contiguous partitions from one end of the store to the other
- Each process has a base, or datum (where it starts)
- Each process also has a limit (length)

Problem

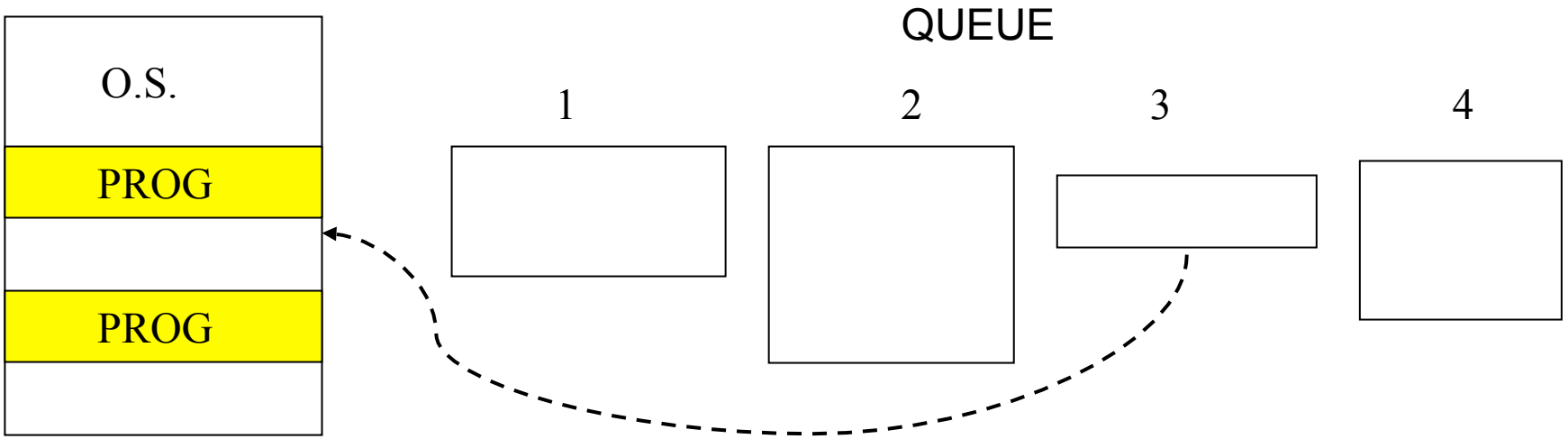
- What if one program terminates and we want to replace it by another?



New program may not fit any available partition

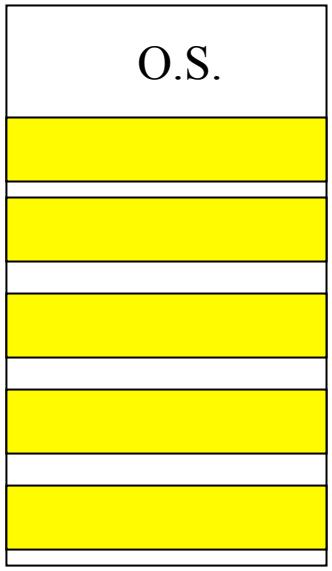
First Possibility

- In a multiprogramming system where we have a queue of programs waiting, select a program of right size to fit partition

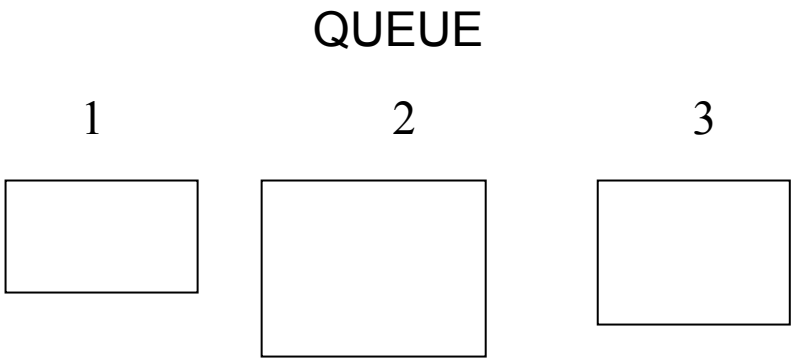


Problems

- Large programs may never get loaded (permanent blocking or starvation)
- Small gaps are created (fragmentation)



FRAGMENTED STORE



BLOCKED PROGRAMS

Question

- In a computer memory, a 100K partition becomes available. In the ready list is a program image of size 300K, plus three others of sizes 100K, 85K and 15K.
- Assuming that our current priority is to avoid starvation of the 300K program, which of those in the list should be swapped into the available partition?

a) The 100K program
b) The 85K program
c) The 15K program
d) Both the 15K and the 85K programs
e) None of the above

Answer: e
If we are avoiding the starvation of the 300K program, then we need to wait for an adjacent partition to become available to allow for the 300K program to run

Loading Programs

- To avoid starvation, may need to let a large program hold up queue until enough space becomes available
- In general, may have to make a choice as to which partition to use
- Selection policies:
  - First fit: Choose first partition of suitable size
  - Best fit: Choose smallest partition which is big enough
  - Worst fit: Choose biggest partition

Question

- A new program requires 100K of memory to run. The memory management approach adopted is a simple partitioning one, and the operating system has the following list of empty partitions:

  60K, 240K, 150K, 600K, 108K, 310K

- Assuming that the 150K partition is chosen, say which of the following selection strategies is being used:

a) First fit
b) Best fit
c) Worst fit
d) All of the above
e) None of the above

Answer: e
First Fit would select 240K
Best Fit would select 108K
Worst Fit would select 600K
… as none of these select the 150K partition, then some other strategy has been used!

Problems with Approach

- Fragmentation may be severe
  - 50% rule
    - For first-fit, if amount of memory allocated is N, then the amount unusable owing to fragmentation is 0.5N
  - Overhead to keep track of gap may be bigger than gap itself
  - May have to periodically compact memory
    - Requires programs to be dynamically relocatable
- Difficult dealing with large programs

Problems (cont'd)

- Shortage of memory
  - arising from fragmentation and/or anti-starvation policy
  - may not be able to support enough users
  - may have difficulty loading enough programs to obtain good job mix
- Imposes limitations on program structure
  - not suitable for sharing routines and data
  - does not reflect high level language structures
  - does not handle store expansion well
- Swapping is inefficient

Swapping

- Would like to start more programs than can fit into physical memory
- To enable this, keep some program images on disk
- During scheduling, a process image may be swapped in, and another swapped out to make room
  - also helps to prevent starvation
- For efficiency, may have dedicated swap space on disk
- However, swapping whole processes adds considerably to time of context switch