• Last class:
  – Deadlocks

• Today:
  – Memory Management
Binding of Programs to Addresses

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)
Loading User Programs
Logical and Physical Addresses

• The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  – **Logical address** – generated by the CPU; also referred to as **virtual address**
  – **Physical address** – address seen by the memory unit

• Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
Memory Management Unit

CPU

logical address 346

+ 

relocation register 14000

physical address 14346

MMU

memory
Need for Memory Management

• Physical memory (DRAM) is limited
  – A single process may not all fit in memory
  – Several processes (their address spaces) also need to fit at the same time
Swapping

1. Swap out
2. Swap in

operating system

user space

main memory

process $P_1$

backing store

process $P_2$
Swapping

• If something (either part of a process, or multiple processes) does not fit in memory, then it has to be kept on disk.
• Whenever that needs to be used by the CPU (to fetch the next instruction, or to read/write a data word), it has to be brought into memory from disk.
• Consequently, something else needs to be evicted from memory.
• Such transfer between memory and disk is called swapping.
• Disallow 1 process from accessing another process’s memory.
• Usually a separate portion of the disk is reserved for swapping, that is referred to as swap space.
• Note that swapping is different from any explicit file I/O (read/write) that your program may contain.
• Typically swapping is transparent to your program.
Early Days: Overlays

In this case, even a single application process does not fit entirely in memory.
Even if a process fits entirely in memory, we do not want to do the following …

Context switching will be highly inefficient, and it defeats the purpose of multiprogramming.
Need for Multiprogramming

- Say your program does explicit file I/O (read/write) for a fraction $f$ of its execution time, then with $p$ processes, CPU efficiency =
  \[(1 - f^p)\]

- To maintain high CPU efficiency, we need to increase $p$.
- But as we just saw, these processes cannot all be on disk. We need to keep as many of these processes in memory as possible.
- So even if we are not keeping all of the process, keep the essential parts of as many processes as possible in memory.

- *We will get back to this issue at a later point!*
Memory Allocation

Queue of waiting requests/jobs

P1

P3

P2

OS

Allocated Regions

Free Regions (Holes)

Question: How do we perform this allocation?
Goals

• `Allocation()` and `Free()` should be fairly efficient

• Should be able to satisfy more requests at any time (i.e. the sum total of holes should be close to 0 with waiting requests).
Solution Strategies

• Contiguous allocation
  – The requested size is granted as 1 big contiguous chunk.
  – E.g. first-fit, best-fit, worst-fit, buddy-system.

• Non-contiguous allocation
  – The requested size is granted as several pieces (and typically each of these pieces is of the same – fixed size).
  – E.g., paging
Contiguous Allocation

- Data structures:
  - Queue of requests of different sizes
  - Queues of allocated regions and holes.
- Find a hole and make the allocation (and it may result in a smaller hole).
- Eventually, you may get a lot of holes that become small enough that they cannot be allocated individually.
- This is called external fragmentation.
Easing External Fragmentation

Note that this can be done only with relocatable code and data (use indirect/indexed/relative addressing)

But compaction is expensive and we want to do this as infrequently as possible.
Contiguous Allocation

• Which hole to allocate for a given request?
• First-fit
  – Search through the list of holes. Pick the first one that is large enough to accommodate this request.
  – Though allocation may be easy, it may not be very efficient in terms of fragmentation.
• Best Fit
  – Search through the entire list to find the smallest hole that can accommodate the given request.
  – Requires searching through the entire list (or keeping it in sorted order).
  – This can actually result in very small sized holes making it undesirable.
• **Worst fit**
  – Pick the largest hole and break that.
  – The goal is to keep the size of holes as large as possible.
  – Allocation is again quite expensive (searching through entire list or keeping it sorted).
What do you do for a free()?

• You need to check whether nearby regions (on either side) are free, and if so you need to make a larger hole.

• This requires searching through the entire list (or at least keeping the holes sorted in address order).
Costs

• Allocation: Keeping list in sorted size order or searching entire list each time \(O(N)\).

• Free: Keeping list in sorted address order or searching entire list each time \(O(N)\).
Buddy System

- \( \log(N) \) cost for allocation/free.
An Example

1 MB block

Request 100K

1 M

128K | 256K | 512K

Request 240K

A=128K | 512K

Request 64K

64K | 512K
Request 256K

Release B

Release A

Request 75K
List of Available Holes

- Start address, end address, size=256K
- Start address, end address, size=128K
- Start address, end address, size=64K
- Start address, end address, size=256K
- Start address, end address, size=512 KB
- Start address, end address, size=1 MB
Slab Allocator

- **Slab** is one or more physically contiguous “pages”
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction
Slab Allocation

- Kernel objects
- Caches
- Slabs

- 3 KB objects
- 7 KB objects

Physical contiguous pages
Summary

• Memory Management
  – Limited physical memory resource
• Keep key process pages in memory
  – Swapping (and paging, later)
• Memory allocation
  – High performance
  – Minimize fragmentation
• Next time: Paging