• Was Great Dijkstra a magician?
• Is this true - If the system is in a safe state, you can run process in any sequence?
• Why are available resources called “work” in the Detection algorithm?
• A computing example of livelock?
  – Two processes, both stepping back to avoid deadlock.
Example A: Banker’s Algorithm

- Is this a safe state?
- Let us see. Is there a safe sequence?

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>available</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

P1 run to completion. Available becomes $[3 \ 3 \ 2]+[2 \ 0 \ 0] = [5 \ 3 \ 2]$
P3 run to completion. Available becomes $[5 \ 3 \ 2]+[2 \ 1 \ 1] = [7 \ 4 \ 3]$
P4 run to completion. Available becomes $[7 \ 4 \ 3]+[0 \ 0 \ 2] = [7 \ 4 \ 5]$
P2 run to completion. Available becomes $[7 \ 4 \ 5]+[3 \ 0 \ 2] = [10 \ 4 \ 7]$
P0 run to completion. Available becomes $[10 \ 4 \ 7]+[0 \ 1 \ 0] = [10 \ 5 \ 7]$

Hence state above is safe.
Example of Detection Algorithm

• Is this deadlocked?

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>available</td>
<td>0 1 0</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>2 0 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 3</td>
<td>0 0 1</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Available</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ini</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P0</td>
<td>0 2 0</td>
</tr>
<tr>
<td>P2</td>
<td>- - -</td>
</tr>
</tbody>
</table>

• State of system?
  – Can reclaim resources held by process \( P_0 \), but insufficient resources to fulfill other processes; requests
  – Deadlock exists, consisting of processes \( P_1, P_2, P_3, \) and \( P_4 \)
Main Memory
Chapter 8: Main Memory

Objectives:

- Organizing memory for multiprogramming environment
  - Partitioned vs separate address spaces
- Memory-management techniques
  - Virtual vs physical addresses
  - Segmentation
  - Paging: page tables, caching (“TLBs”)
- Examples: the Intel (old/new) and ARM architectures
What we want

• Memory capacities have been increasing
  – But programs are getting bigger faster
  – Parkinson’s Law: Programs expand to fill the memory available to hold

• What we would like
  – Memory that is
    • infinitely large, infinitely fast
    • Non-volatile
    • Inexpensive too

• Unfortunately, no such memory exists as of now
Background

• Program must be brought (from disk) into memory and run as a process
• Main memory and registers are only storage CPU can access directly
• Memory unit only sees a stream of
  – addresses + read requests, or
  – address + data and write requests
• Access times:
  – Register access in one CPU clock (or less)
  – Main memory can take many cycles, causing a stall
  – Cache sits between main memory and CPU registers
    making main memory appear much faster
• Protection of memory required to ensure correct operation
Main memory and registers are only storage CPU can access directly.

Register access in one CPU clock (or less).
Main memory can take many cycles, causing a stall.

Cache sits between main memory and CPU registers making main memory appear much faster.

Ch 8

Ch 9

Ch 10, 11, 12: Disk, file system

Cache: CS470
Protection: Making sure each process has separate memory spaces

- OS must be protected from accesses by user processes
- User processes must be protected from one another
  - Determine range of legal addresses for each process
  - Ensure that process can access only those
- Approach:
  - Partitioning address space
  - Separate address spaces (modern practice, we will see later)
Partitioning: Base and Limit Registers

- Base and Limit for a process
  - **Base**: Smallest legal physical address
  - **Limit**: Size of the range of physical address

- A pair of **base** and **limit registers** define the logical address space for a process

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

- **Base**: Smallest legal physical address
- **Limit**: Size of the **range** of physical address
- Eg: Base = 300040 and limit = 120900
- Legal: 300040 to \((300040 + 120900 - 1)\) = 420939
Hardware Address Protection

Legal addresses: **Base address** to **Base address + limit -1**
Address Binding Questions

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program’s life
  - **Source code** addresses are symbolic
  - **Compiled code** addresses **bind** to relocatable addresses
    - i.e. “14 bytes from beginning of this module”
  - **Linker or loader** will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another
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)
THE HISTORY OF MEMORY

IN 1940 I INVENTED A BIT
OF MEMORY

1953: I INVENTED A
BYTE OF MEMORY

1966: 1 K

1978: 32 K
ONLY $999

2011: LOOK, 100
TERABYTES

2038: WHAT
MEMORY?
Multistep Processing of a User Program

1. Source program
2. Compiler or assembler
3. Object module
4. Linkage editor
5. Load module
6. Loader
7. In-memory binary memory image

- Compile time
- Load time
- Execution time (run time)
• 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 address space** is the set of all logical addresses generated by a program

• **Physical address space** is the set of all physical addresses
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
  - Many methods possible, we will see them soon
- Consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called relocation register
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses
Dynamic relocation using a relocation register
Loading vs Linking

• **Loading**
  – Load executable into memory prior to execution

• **Linking**
  – Takes some smaller executables and joins them together as a single larger executable.
Linking: Static vs Dynamic

- **Static linking** – system libraries and program code combined by the loader into the binary image
  - Every program includes library: wastes memory
- **Dynamic linking** – linking postponed until execution time
  - Operating system checks if routine is in processes’ memory address
Dynamic Linking

- **Dynamic linking** – linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes’ memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for
  - shared libraries
Dynamic loading of routines

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- OS can help by providing libraries to implement dynamic loading
Memory Allocation Approaches

- **Contiguous allocation**: entire memory for a program in a single contiguous memory block. Find where a program will “fit”. Earliest approach.

- **Segmentation**: program divided into logically divided “segments” such as main program, function, stack etc.
  - Need table to track segments.

- **Paging**: program divided into fixed size “pages”, each placed in a fixed size “frame”.
  - Need table to track pages.
Swapping a process

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
Schematic View of Swapping

1. **Swap out**: Process $P_1$ is swapped out from main memory to the backing store.
2. **Swap in**: Process $P_2$ is swapped in from the backing store to main memory.

Diagram:
- **Operating System**
- **User Space**
- **Main Memory**
- **Back ing Store**
- **Process $P_1$**
- **Process $P_2$**
Context Switch Time including Swapping

• If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
• Context switch time can then be very high
• 100MB process swapping to hard disk with transfer rate of 50MB/sec
  – Swap out time of 100MB/50MB/s = 2 seconds
  – Plus swap in of same sized process
  – Total context switch swapping component time of 4 seconds + some latency
• Can reduce if reduce size of memory swapped – by knowing how much memory really being used by a process
• Standard swapping not used in modern operating systems
  – But modified version common
    • Swap only when free memory extremely low
Contiguous Allocation
Contiguous Allocation

• Main memory must support both OS and user processes
• Limited resource, must allocate efficiently
• Contiguous allocation is one early method
• Main memory usually into two partitions:
  – Resident operating system, usually held in low memory with interrupt vector
  – User processes then held in high memory
  – Each process contained in single contiguous section of memory
• **Registers** used to protect user processes from each other, and from changing operating-system code and data
  - **Relocation (Base) register** contains value of smallest physical address
  - **Limit register** contains range of logical addresses – each logical address must be less than the limit register

• **MMU** maps logical address *dynamically*
Hardware Support for Relocation and Limit Registers

MMU maps logical address *dynamically*

*Physical address = relocation reg + valid logical address*
Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process’ needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions  
  b) free partitions (hole)
Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes?

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

**Simulation studies:**

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- Best fit is *slower* than first fit. Surprisingly, it also results in more *wasted memory* than first fit
  - Tends to fill up memory with tiny, useless holes
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

- Simulation analysis reveals that given $N$ blocks allocated, 0.5 $N$ blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**
• Reduce external fragmentation by **compaction**
  – Shuffle memory contents to place all free memory together in one large block
  – Compaction is possible *only* if relocation is dynamic, and is done at execution time
  – I/O problem
    • Latch job in memory while it is involved in I/O
    • Do I/O only into OS buffers
FAQ

Why we need virtual address space?
• Support for multiprogramming
• Protection

Paging:
• No external fragmentation!
• Internal fragmentation in Paging: part of a page not used
• Optimal page size:
  – Total Overhead = Page table overhead + Internal fragmentation loss
• TLB: on the processor chip or outside?

How does MMU handle dynamic memory allocation?
Paging vs Segmentations
Pages

- Pages and frames
- Page tables
- TLB: page table caching
- Memory protection and sharing
- Multilevel page tables
Paging

• Divide physical memory into fixed-sized blocks called **frames**
  – Size is power of 2, between 512 bytes and 16 Mbytes
• Divide logical memory into blocks of same size called **pages**
• To run a program of size $N$ pages, need to find $N$ free frames and load program
• Still have Internal fragmentation
• Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  – Avoids external fragmentation
  – Avoids problem of varying sized memory chunks
Paging

- physical memory - **frames**
- logical memory - **pages**
- Keep track of all free frames
- Set up a **page table** to translate logical to physical addresses
Address Translation Scheme

• Address generated by CPU is divided into:
  – **Page number** \((p)\) – used as an index into a page table which contains base address of each page in physical memory
  – **Page offset** \((d)\) – combined with base address to define the physical memory address that is sent to the memory unit

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p)</td>
<td>(d)</td>
</tr>
</tbody>
</table>

  – For given logical address space \(2^m\) and page size \(2^n\)
Paging Hardware

Page number \( p \) to frame number \( f \) translation
Paging Example

Page 0 maps to frame 5

$n = 2$ and $m = 4$  Logical add. space = $2^4$ bytes, $2^2$=4-byte pages
32-byte memory with 8 frames
• **Internal fragmentation**
  – Ex: Page size = 2,048 bytes, Process size = 72,766 bytes
    • 35 pages + 1,086 bytes
    • Internal fragmentation of 2,048 - 1,086 = 962 bytes
  – Worst case fragmentation = 1 frame – 1 byte
  – On average fragmentation = 1 / 2 frame size
  – So small frame sizes desirable?
    • But each page table entry takes memory to track
  – Page size growing over time
    • X86-64: 4 KB (common), 2 MB (“huge” for servers), 1GB (“large”)

• **Process view and physical memory now very different**

• **By implementation, a process can only access its own memory** unless ..
Free Frame allocation

Before allocation

After allocation
Implementation of Page Table

Page table is kept in main memory

- **Page-table base register (PTBR)** points to the page table

- **Page-table length register (PTLR)** indicates size of the page table

- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction

  The *two memory access problem* can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**.