Main Memory (I)

Operating Systems

Yu-Ting Wu

(with slides borrowed from Prof. Jerry Chou)

1

Outline

- Background
- Swapping
- Contiguous allocation
- Paging

2

Background

3

Background

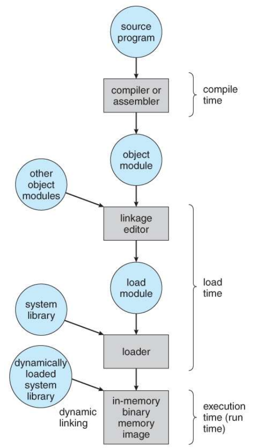
- **Main memory** and **registers** are the only storage CPU can access directly
- Collection of **processes** are waiting on disk to be brought into memory and be executed
- **Multiple programs** are brought into memory to improve resource utilization and response time to users
- A process may be **moved between disk and memory** during run time

4

Questions

- How to refer memory in a program?
  - Address binding
- How to load a program into memory?
  - Static / dynamic loading and linking
- How to move a program between memory and disk?
  - Swap
- How to allocate memory
  - Paging, segment

Steps of Processing a Program



Address Binding: Compile Time

- Program is written as symbolic code
- Compiler translates symbolic code into **absolute code**
- If starting location changes ➔ **recompile**

Compile → Load

Source Program:

```
int data;
main( ) {
    data = 3 * 7;
    print(data);
}
```

Disk Image:

```
.BASE  0x1000
.START
PUSH   AX
MOVE   AX, 3
MULT   AX, 7
MOVE   (0x1018), AX
CALL   print, (0x1018)
POP    AX
.END
.SPACE (4)
```

Memory Content:

```
0x1000  PUSH   AX
        MOVE   AX, 3
        MULT   AX, 7
        MOVE   (0x1018), AX
0x1010  CALL   print, (0x1018)
        POP    AX
0x1018
```


Address Binding: Load Time

- Compiler translates symbolic code into **relocatable code**
- **Relocatable code**
  - Machine language that can be run from any memory location
  - If starting location changes ➔ **reload the code**

Compile → Load

Source Program:

```
int data;
main( ) {
    data = 3 * 7;
    print(data);
}
```

Disk Image:

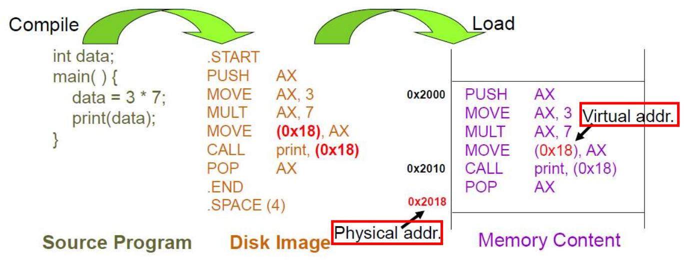
```
.START
PUSH   AX
MOVE   AX, 3
MULT   AX, 7
MOVE   (.BS+0x18), AX
CALL   print, (.BS+0x18)
POP    AX
.END
.SPACE (4)
```

Memory Content:

```
0x2000  PUSH   AX
        MOVE   AX, 3
        MULT   AX, 7
        MOVE   (0x2018), AX
0x2010  CALL   print, (0x2018)
        POP    AX
0x2018
```

Address Binding: Execution Time

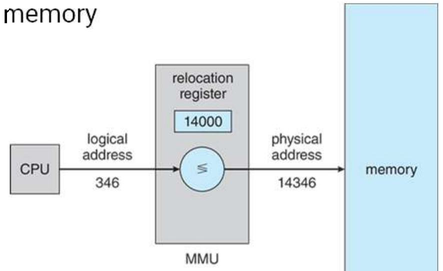
- Compiler translates symbolic code into **logical-address (i.e. virtual-address) code**
- **Special hardware (i.e. MMU)** is needed for this scheme
- Most general-purpose OS use this method



9

Memory-Management Unit (MMU)

- **Hardware** device that **maps virtual to physical** address
- The value in the **relocation register is added to every address** generated by a user process at the time it is sent to memory



Note: this is the simplest case

10

Logical v.s. Physical Address

- **Logical address** – generated by CPU
  - a.k.a virtual address
- **Physical address** – seen by the memory module
- **Compile-time** and **load-time** address binding
  - Logical address = physical address
- **Execution-time** address binding
  - Logical address ≠ physical address
- The user program deals with **logical** addresses; it never sees the real physical addresses

11

Questions

- How to refer memory in a program?
  - Address binding
- How to load a program into memory?
  - Static / dynamic loading and linking
- How to move a program between memory and disk?
  - Swap
- How to allocate memory
  - Paging, segment

12

Dynamic Loading

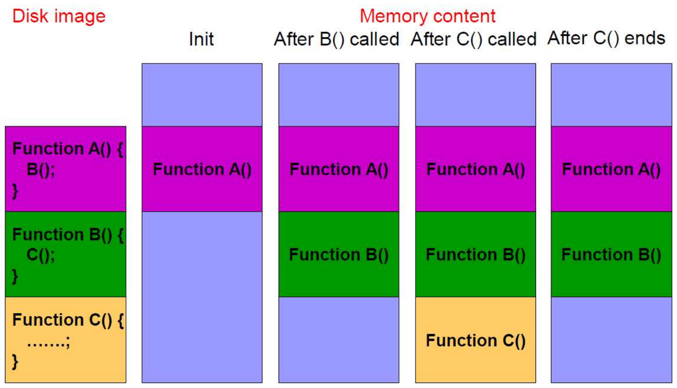
- The entire program must be in memory for it to execute?
- No, we can use **dynamic loading**
  - A routine is loaded into memory when it is called
- **Better memory-space utilization**
  - Unused routine is never loaded
  - Particularly useful when large amounts of code are infrequently used (e.g., error handling code)
- **No special support from OS** is required, implemented through programs (library, API calls)

Dynamic Loading Example in C

- ***dlopen()***: opens a library and prepares it for use
- ***dlsym()***: looks up the value of a symbol in a given (opened) library
- ***dlclose()***: closes a DL library

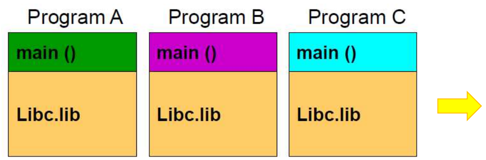
```
#include <dlfcn.h>
int main() {
  double (*cosine)(double);
  void* handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
  cosine = dlsym(handle, "cos");
  printf ("%f\n", (*cosine)(2.0));
  dlclose(handle);
}
```

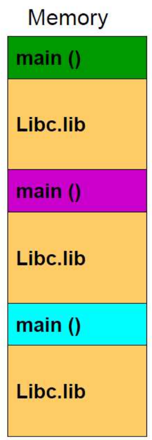

Dynamic Loading



Static Linking

- **Static linking:** libraries are combined by the loader into the program in-memory image
  - Waste memory: **duplicated code**
  - Faster during execution time
- **Static linking + dynamic loading ?**
  - Still can't prevent duplicate code

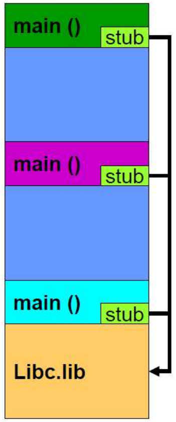




Dynamic Linking

- **Dynamic linking:** linking postponed **until execution time**
  - **Only one code copy** in memory and shared by everyone
  - **A stub** is included in the program in-memory image for each lib reference
  - **Stub call**
    - ➔ check if the referred lib is in memory
    - ➔ if not, load the lib
    - ➔ execute the lib
  - DLL (dynamic link library) on Windows

Memory



Swapping

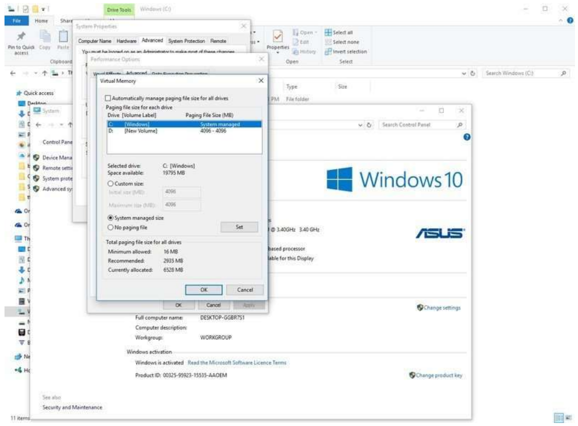
Questions

- How to refer memory in a program?
  - Address binding
- How to load a program into memory?
  - Static / dynamic loading and linking
- How to move a program between memory and disk?
  - Swap
- How to allocate memory
  - Paging, segment

Swapping

- A process can be swapped out of memory to a **backing store**, and later brought back into memory for continuous execution
  - Also used by **midterm scheduling**, different from context switch
- **Backing store** – a chunk of the disk, **separated from the file system**, to provide direct access to these memory images
- Why swap a process?
  - Free up memory
  - **Roll out, roll in:** swap lower-priority process with a higher one

Swapping (cont.)



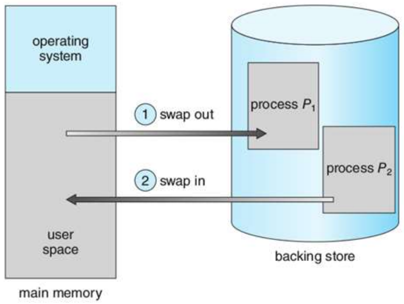
Swapping (cont.)

- Swap back memory location
  - If binding is done at compile / load time
    - Swap back memory address must be the **same**
  - If binding is done at execution time
    - Swap back memory address can be **different**

- A process to be swapped ➔ **must be idle**
  - Imagine a process that is waiting for I/O is swapped
  - Solutions:
    - Never swap a process with pending I/O
    - I/O operations are done through **OS buffers** (i.e. a memory space not belongs to any user processes)

Process Swapping to Backing Store

- **Major part of swap time is transfer time**; total transfer time is directly proportional to the amount of memory swapped



Contiguous Allocation

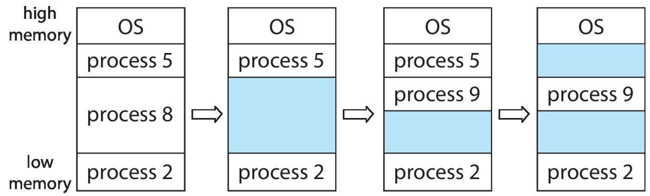
Memory Allocation

- **Fixed-partition allocation**
  - Each process loads into one partition of fixed-size
  - **Degree of multi-programming** is bounded by the number of partitions
- **Variable-size partition**
  - Hole: block of contiguous free memory
  - Holes of various sizes are scattered in memory

25

Multiple Partition (Variable-Size) Method

- When a process arrives, it is allocated a hole **large enough** to accommodate it
- The OS maintains info. of each in-use and free hole
- A freed hole can be merged with another hole to form a larger hole



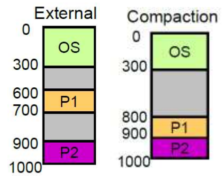
26

Dynamic Storage Allocation Problem

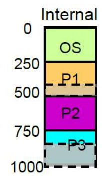
- How to satisfy a request of size n from a list of free holes
- **First-fit** – allocate the 1st hole that fits
- **Best-fit** – allocate the smallest hole that fits
  - Must search through the whole list
- **Worst-fit** – allocate the largest hole
  - Must also search through the whole list
- **First-fit** and **best-fit** are better than worst-fit in terms of **speed** and **storage utilization**

27

Fragmentation



- **External fragmentation**
  - Total free memory space is big enough to satisfy a request but is not contiguous
  - Occur in **variable-size allocation**
  - Solution: **compaction**
    - Shuffle the memory contents to place all free memory together in one large block at execution time
    - Only if the binding is done at execution time
- **Internal fragmentation**
  - Memory that is internal to a partition but is not being used
  - Occur in **fixed-partition allocation**



28

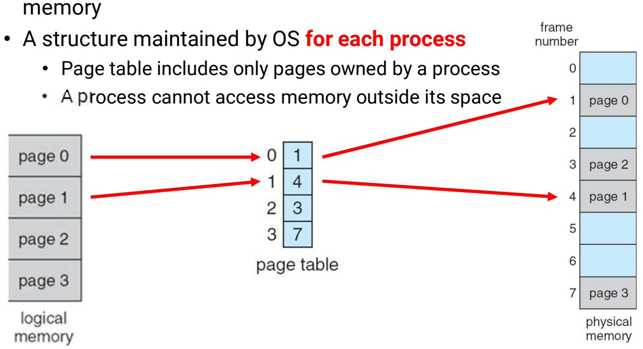
Paging
(Non-Contiguous Memory Allocation)

Paging Concept

- Method
  - Divide **physical memory** into fixed-size blocks called **frames**
  - Divide **logical address** space into blocks of the **same size** called **pages**
  - To run a program of *n* pages, need to find *n* free frames and load the program
  - **Must keep track of free frames**
  - Set up a **page table** to translate logical to physical addresses
- Benefit
  - Allow the physical-address space of a process to be **noncontiguous**
  - Avoid external fragmentation
  - Limited internal fragmentation
  - Provide **shared memory / pages**

Paging Example

- **Page table**
  - Each entry maps to the **base address of a page** in physical memory
  - A structure maintained by OS **for each process**
    - Page table includes only pages owned by a process
    - A process cannot access memory outside its space

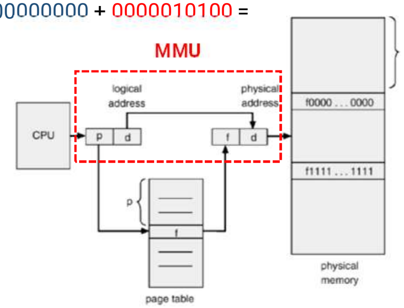


Address Translation Scheme

- Logical address is divided into two parts
  - **Page number (p)**
    - Used as an **index into a page table** which contains **base address** of each page in physical memory
    - *N* bits means a process can allocate at most 2^N pages
      ➔ **2^N x page size memory size**
  - **Page offset (d)**
    - Combined with base address to define the physical memory address that is sent to the memory unit
    - ***N* bits means the page size is 2^N**
- **Physical address** = **page base address** + **page offset**

Address Translation Architecture

- If page size is 1KB (2^10) and page 3 maps to frame 5
- Given 13 bits logical address (p = 3, d = 20), what is the physical address?
  - 5 * (1KB) + 20 = 1010000000000 + 0000010100 = 1010000010100



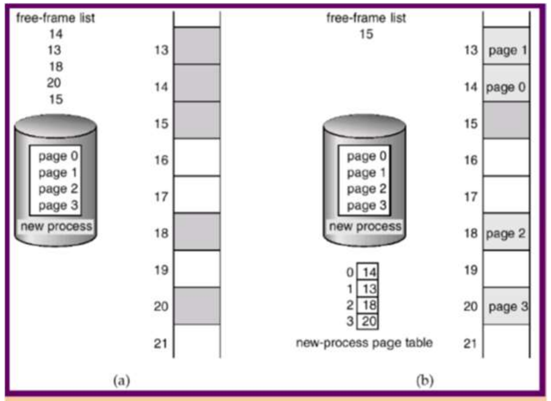
33

Address Translation

- Total number of pages does not need to be the same as the total number of frames
  - **Total # pages** determines the logical memory size of a process
  - **Total # frames** depending on the size of physical memory
- E.g.: Given 32 bits logical address, 36 bits physical address, and 4KB page size, what does it mean?
  - Number of bits for page offset: 4KB page size = 2^{12} bytes ➔12
  - Number of bits for page number: 2^{20} pages ➔ 20 bits
  - Page table size: 2^{32} / 2^{12} = 2^{20} entries
  - Max program memory: 2^{32} = 4GB
  - Number of bits for frame number: 2^{24} frames ➔ 24 bits
  - Total physical memory size: 2^{36} = 64GB

34

Free Frames



Before allocation After allocation

35

Page / Frame Size

- The page (frame) size is defined by hardware
  - **Typically, a power of 2**
  - Ranging from 512 bytes to 16 MB / page
  - 4KB / 8KB page is commonly used
- Internal fragmentation?
  - Larger page size ➔ More space waste
- But **page sizes cannot be too small**
  - Memory, process, and data sets have become larger
  - Need to keep page table small
  - Fewer access means better I/O performance

36

Paging Summary

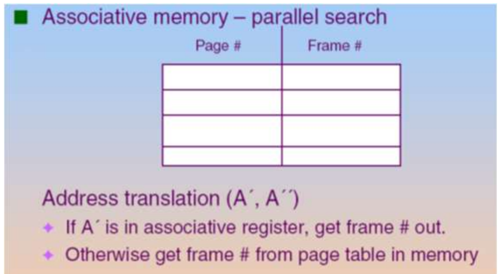
- Paging helps separate **user's view** of memory and the actual **physical memory**
- User view's memory: one single contiguous space
  - Actually, user's memory is scattered out in physical memory
- OS maintains a copy of the **page table** for each process
- OS maintains a **frame table** for managing physical memory
  - One entry for each physical frame
  - Indicate whether a frame is free or allocated
  - If allocated, to which page of which process or processes

Implementation of Page Table

- Page table is kept **in memory**
- **Page-table base register (PTBR)**
  - The **physical memory address** of the page table
  - The PTBR value is stored in **PCB** (Process Control Block)
  - Changing the value of PTBR during the context switch
- With PTBR, each memory reference results in **2 memory reads**
  - One for the page table and one for the real address
- The 2-access problem can be solved by
  - **Translate Look-aside Buffers (TLB)** (HW) which is implemented by **Associative memory** (HW)

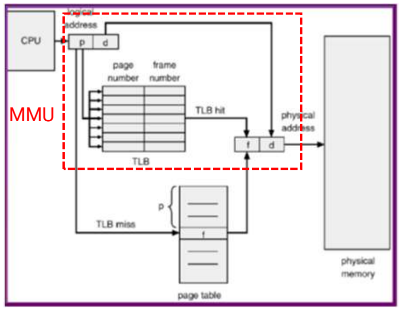
Associative Memory

- All memory entries can be accessed at the same time
  - Each entry corresponds to an associative register
- But **the number of entries are limited**
  - Typical number of entries: 64 ~ 1024



Translation Look-aside Buffer (TLB)

- **A cache for page table shared by all processes**
- TLB must be **flushed** after a context switch
  - Otherwise, TLB entry must has a PID field (address-space identifiers (ASIDs))



Effective Memory-Access Time

- 20 ns for TLB search
- 100 ns for memory access
- Effective Memory-Access Time (**EMAT**)
  - 70% TLB hit-ratio:

  ➔ EMAT = 0.70 x (**20 + 100**) + (1 – 0.70) * (**20 + 100 + 100**) = 150 ns
  - 98% TLB hit-ratio:

  ➔ EMAT = 0.98 x 120 + 0.02 x 220 = 122 ns