

## Course Info

### Objectives:

- Know clearly the components of a computer system, problems and solutions in the history of computing system development
- Master the algorithms used in the OS, be able to evaluate algorithms, as well as be able to apply them to solve real-life problems
- Understand basic OS's services (related to processes, threads, network, memory, directories, files)
- Grasp knowledge of data structures used in OS, improve concurrency programming capabilities, and system-level programming



ĐẠI HỌC BÁCH KHOA HÀ NỘI  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

25 VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

# Operating Systems

(*Principles of Operating Systems*)

- Know clearly the components of a computer system, problems and solutions in the history of computing system development
- Master the algorithms used in the OS, be able to evaluate algorithms, as well as be able to apply them to solve real-life problems
- Understand basic OS's services (related to processes, threads, network, memory, directories, files)
- Grasp knowledge of data structures used in OS, improve concurrency programming capabilities, and system-level programming

Đỗ Quốc Huy  
huydq@soict.hust.edu.vn

Department of Computer Science  
School of Information and Communication Technology



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## Course Outline

### Chapters:

1. Introduction (2.5 weeks)
2. Process Management (5.5 weeks)
3. Memory Management (3 weeks)
4. File Systems (2 weeks)
5. IO Management (1 week)

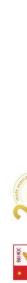
### Prerequisite:

- Introduction to information technology, Data structure and algorithm
  - C programming skill

## Course Info

### Prerequisite:

- Introduction to information technology, Data structure and algorithm
  - C programming skill



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

# Chapter 1 Operating System Overview

## Course Info

- ① Operating system concept
- ② History of operating Systems
- ③ Definition and Classifications
- ④ Concepts in operating systems
- ⑤ Operating System structures
- ⑥ Basic Properties of Operating Systems
- ⑦ Principles of Operating Systems

### Text book:

- Operating System Concepts – Abraham Siblerschatz
- Modern Operating System – Andrew Tanenbaum
- Bài giảng Hệ Điều Hành – Nguyễn Thành Tùng
  - Nguyễn lý Hệ Điều Hành – Đỗ Văn Uy
  - Others universities' textbook

# Chapter 1 Operating System Overview

### Lectures

- Come! Cannot guarantee content will be identical to previous years.
- Meet your fellow students, they are your future teammate or colleagues!
- Electronic devices used only for **note taking**.

### Exams

- Attendance mandatory
- Communication Policy
- Communicate with lecturer, classmate through **Team**s rather than email

## Class Expectations

### Lectures

- Come! Cannot guarantee content will be identical to previous years.
- Meet your fellow students, they are your future teammate or colleagues!
- Electronic devices used only for **note taking**.

### Exams

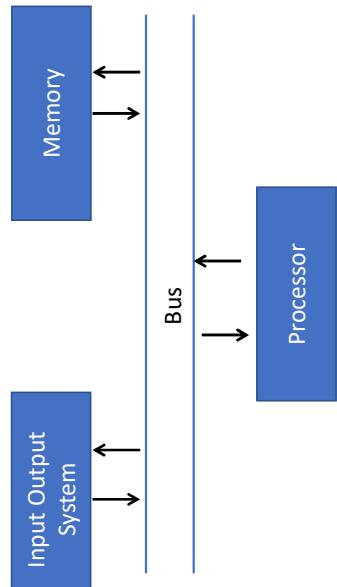
- Attendance mandatory
- Communication Policy
- Communicate with lecturer, classmate through **Team**s rather than email

## Chapter 1. Operating System Overview

### 1. Operating system concept

#### 1.1. Layered structure of a computing system

## A computer system's structure



- One/ many CPUs, controlling devices are linked by a common bus system to access a shared memory.
- Controlling devices and CPU operate simultaneously and compete with each other.

## Chapter 1. Operating System Overview

### 1. Operating system concept

#### 1.1. Layered structure of a computing system

- ① Operating system concept
- Layered structure of a computing system
  - Operating system's functions



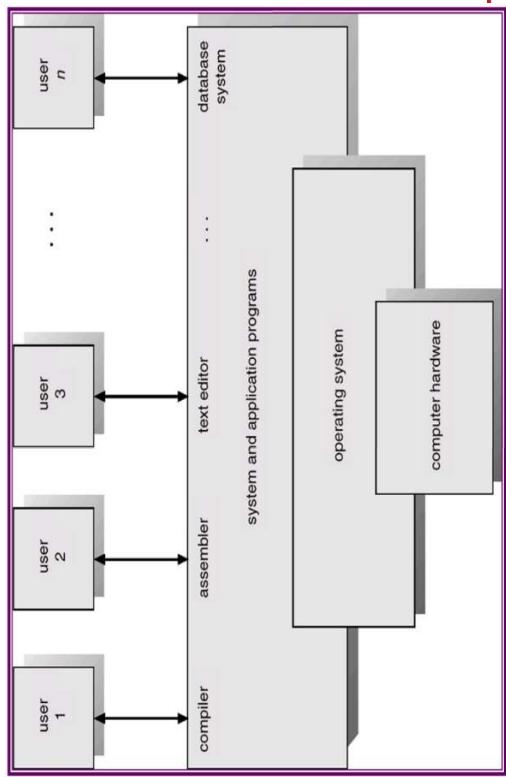
VĨNH CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## Chapter 1. Operating System Overview

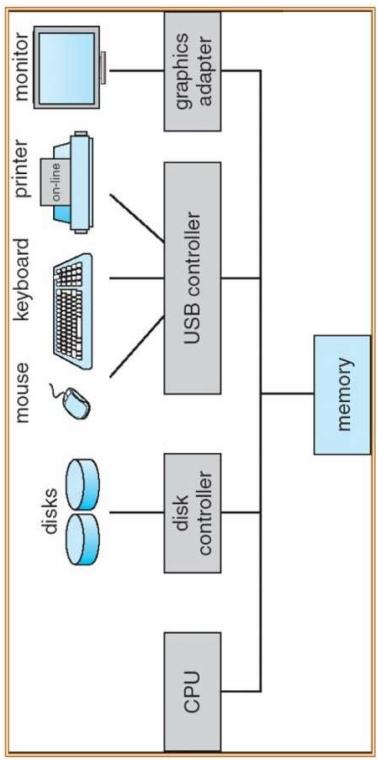
### 1. Operating system concept

#### 1.1. Layered structure of a computing system

## A computer system's components (Silberschatz 2002)



## A computer system's structure (Silberschatz 2002)



## Chapter 1. Operating System Overview

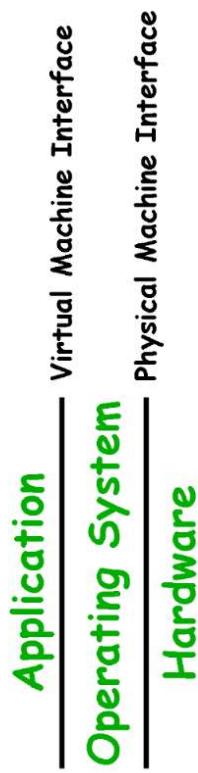
### 1. Operating system concept

#### 1.1. Layered structure of a computing system

VĨNH CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

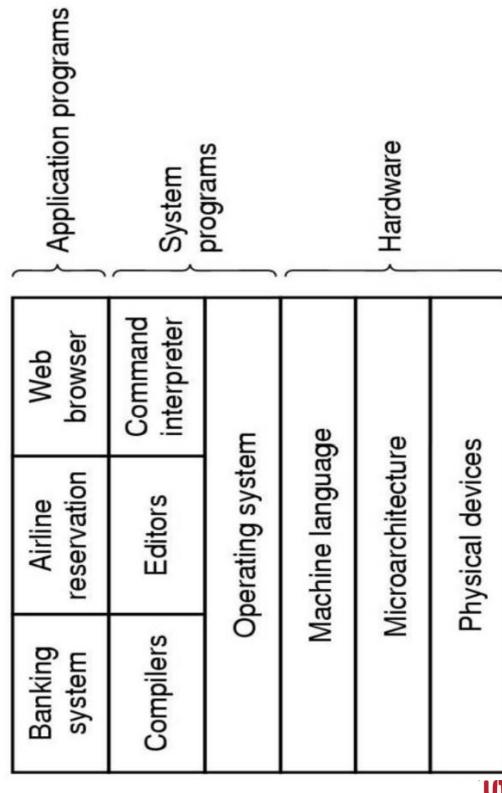


## Objectives



- **Objectives:** To provide an environment which helps user run application program and use the computer system easier, more conveniently and effectively.
- Standardize the user interface for different hardware systems
  - Utilize hardware's resource effectively and exploit the hardware's performance optimally.

## A computer system's components (Tanenbaum 2001)



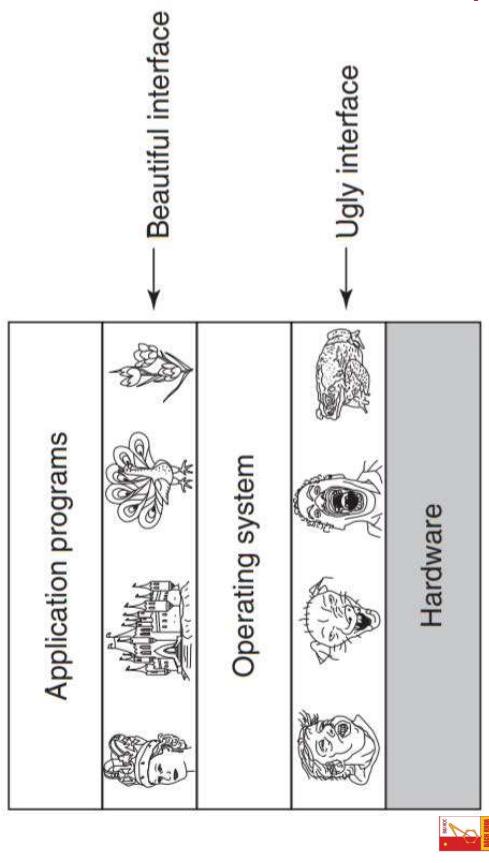
## A computing system's components

1. **Hardware**
2. **Operating system**
3. **Applications programs**
4. **Users**

## Chapter 1. Operating System Overview

1. Operating system concept
- 1.2. Operating system's functions

### Simulate a virtual computer machine



## Chapter 1. Operating System Overview

1. Operating system concept
- 1.2. Operating system's functions

### ① Simulate a virtual computer machine

### ② Manage system resources



## Chapter 1. Operating System Overview

1. Operating system concept
- 1.2. Operating system's functions

### Simulate a virtual computer

- Extend the system's abilities: The system seem to have **desired resources** (virtual memory, virtual printer...)

Help **hide detailed works** and exploit computer hardware's functions easier and more effectively.

- Simplify programming problem
  - No need to work with binary sequences
  - Each program thinks that it own the whole computer's memory, CPU time, devices...
- Help communicating with devices easier than with original device. Ex: Ethernet card: Reliable communication, ordered (TCP/IP)
  - If experimental OS get errors, only limited in the virtual machine
  - Help verifying other programs in the OS



## System's resources management (cont.)

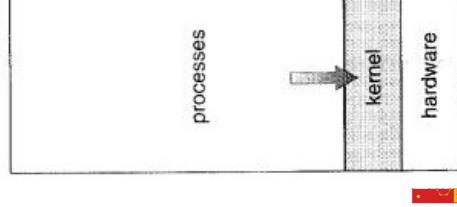
- The OS has to manage the resource so that the computer can work in the most effective way.
  - Provide resource for program when it's necessary.
  - Handle competition
  - Decide the order of resource providing for the programs' requirements
  - Example: memory resource management (limited)
    - Many program can operate at the same time.
    - Avoid illegal access
    - Data protection (memory sharing: file)

## How do operating system work

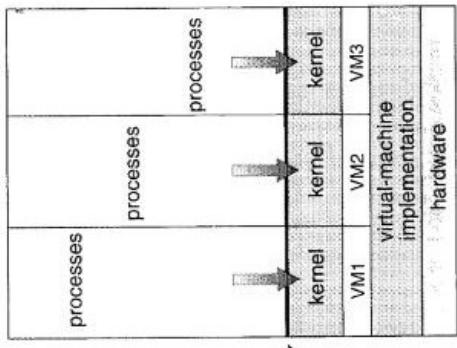
- System's resources (CPU, memory, IO devices, files...) are utilized by program to perform a determined task.
- Programs require resources: time (CPU-usage) and space (memory)
- The OS has to manage the resource so that the computer can work in the most effective way.

## Simulate a virtual computer machine

### Non virtual machine



### With virtual machine



## System's resources management

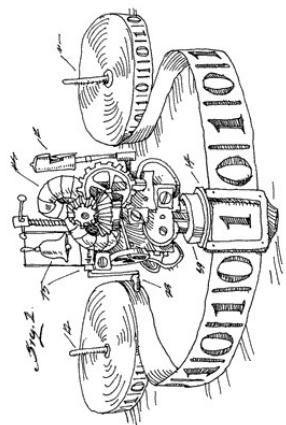
# Chapter 1 Operating System Overview

Chapter 1. Operating System Overview  
2.History of Operating Systems  
2.1. History of electronic computer

## Development history of electronic computers

- 1936 - A. Turing & Church present logic computing model and prove the existence of a computer: Turing machine

- Model of a character processing device; Simple but able to perform all the computer's algorithm
- A Turing machine that is able to simulate any other Turing machine -> a universal Turing machine
- *Turing is considered as the father of computer science and artificial intelligence*

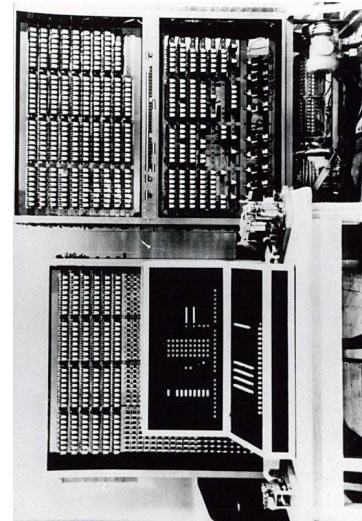


VIỆT CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 1. Operating System Overview  
2.History of Operating Systems  
2.1. History of electronic computer

## Development history of electronic computers

- 1941- Konrad Zuse (German) Constructed world's first programmable computer; the functional program-controlled Turing-complete Z3



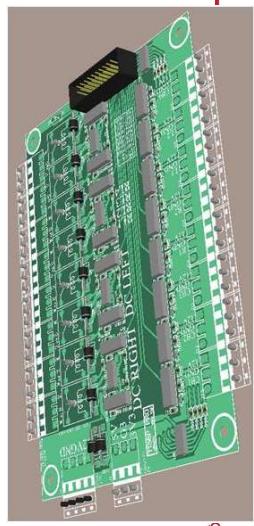
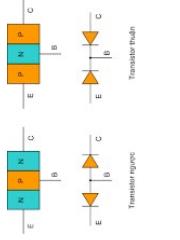
- Z3: use binary system
- Has separated memory and controller
- Mechanical technique

## Operating systems development's history

- History of electronic computer
- Operating systems development's history

## Development history of electronic computers

- 1950-1958 Transistor
- 1959-1963 Semiconductor
  - Transistor logic
  - Transistor switch
- 1964-1974 Integrated Circuit (IC)



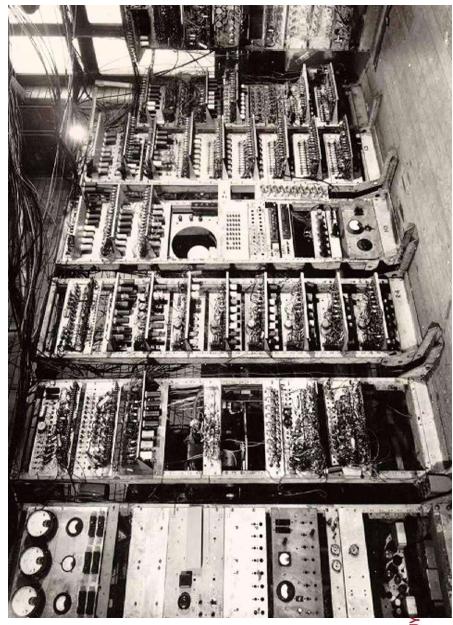
## Development history of electronic computers

- 1974-1990 Large scale IC:
  - Allow CPU, main memory or similar device to be produced in a single integrated circuit
  - new class of smaller, cheaper computer and parallel processor with multi CPUs



## Development history of electronic computers

- 1946 ENIAC based on electric bulbs



## Development history of electronic computers

- plug board
  - ENIAC
  - EDVAC
  - UNIVAC
  - IBM 701
  - IBM 704
  - IBM 7090
  - IBM 7094
  - IBM 7030
  - IBM 7040
  - IBM 7050
  - IBM 7060
  - IBM 7070
  - IBM 7072
  - IBM 7074
  - IBM 7076
  - IBM 7077
  - IBM 7078
  - IBM 7079
  - IBM 7080
  - IBM 7085
  - IBM 7090
  - IBM 7094
  - IBM 7096
  - IBM 7098
  - IBM 7099
  - IBM 7100
  - IBM 7101
  - IBM 7102
  - IBM 7103
  - IBM 7104
  - IBM 7105
  - IBM 7106
  - IBM 7107
  - IBM 7108
  - IBM 7109
  - IBM 7110
  - IBM 7111
  - IBM 7112
  - IBM 7113
  - IBM 7114
  - IBM 7115
  - IBM 7116
  - IBM 7117
  - IBM 7118
  - IBM 7119
  - IBM 7120
  - IBM 7121
  - IBM 7122
  - IBM 7123
  - IBM 7124
  - IBM 7125
  - IBM 7126
  - IBM 7127
  - IBM 7128
  - IBM 7129
  - IBM 7130
  - IBM 7131
  - IBM 7132
  - IBM 7133
  - IBM 7134
  - IBM 7135
  - IBM 7136
  - IBM 7137
  - IBM 7138
  - IBM 7139
  - IBM 7140
  - IBM 7141
  - IBM 7142
  - IBM 7143
  - IBM 7144
  - IBM 7145
  - IBM 7146
  - IBM 7147
  - IBM 7148
  - IBM 7149
  - IBM 7150
  - IBM 7151
  - IBM 7152
  - IBM 7153
  - IBM 7154
  - IBM 7155
  - IBM 7156
  - IBM 7157
  - IBM 7158
  - IBM 7159
  - IBM 7160
  - IBM 7161
  - IBM 7162
  - IBM 7163
  - IBM 7164
  - IBM 7165
  - IBM 7166
  - IBM 7167
  - IBM 7168
  - IBM 7169
  - IBM 7170
  - IBM 7171
  - IBM 7172
  - IBM 7173
  - IBM 7174
  - IBM 7175
  - IBM 7176
  - IBM 7177
  - IBM 7178
  - IBM 7179
  - IBM 7180
  - IBM 7181
  - IBM 7182
  - IBM 7183
  - IBM 7184
  - IBM 7185
  - IBM 7186
  - IBM 7187
  - IBM 7188
  - IBM 7189
  - IBM 7190
  - IBM 7191
  - IBM 7192
  - IBM 7193
  - IBM 7194
  - IBM 7195
  - IBM 7196
  - IBM 7197
  - IBM 7198
  - IBM 7199
  - IBM 7200
  - IBM 7201
  - IBM 7202
  - IBM 7203
  - IBM 7204
  - IBM 7205
  - IBM 7206
  - IBM 7207
  - IBM 7208
  - IBM 7209
  - IBM 7210
  - IBM 7211
  - IBM 7212
  - IBM 7213
  - IBM 7214
  - IBM 7215
  - IBM 7216
  - IBM 7217
  - IBM 7218
  - IBM 7219
  - IBM 7220
  - IBM 7221
  - IBM 7222
  - IBM 7223
  - IBM 7224
  - IBM 7225
  - IBM 7226
  - IBM 7227
  - IBM 7228
  - IBM 7229
  - IBM 7230
  - IBM 7231
  - IBM 7232
  - IBM 7233
  - IBM 7234
  - IBM 7235
  - IBM 7236
  - IBM 7237
  - IBM 7238
  - IBM 7239
  - IBM 7240
  - IBM 7241
  - IBM 7242
  - IBM 7243
  - IBM 7244
  - IBM 7245
  - IBM 7246
  - IBM 7247
  - IBM 7248
  - IBM 7249
  - IBM 7250
  - IBM 7251
  - IBM 7252
  - IBM 7253
  - IBM 7254
  - IBM 7255
  - IBM 7256
  - IBM 7257
  - IBM 7258
  - IBM 7259
  - IBM 7260
  - IBM 7261
  - IBM 7262
  - IBM 7263
  - IBM 7264
  - IBM 7265
  - IBM 7266
  - IBM 7267
  - IBM 7268
  - IBM 7269
  - IBM 7270
  - IBM 7271
  - IBM 7272
  - IBM 7273
  - IBM 7274
  - IBM 7275
  - IBM 7276
  - IBM 7277
  - IBM 7278
  - IBM 7279
  - IBM 7280
  - IBM 7281
  - IBM 7282
  - IBM 7283
  - IBM 7284
  - IBM 7285
  - IBM 7286
  - IBM 7287
  - IBM 7288
  - IBM 7289
  - IBM 7290
  - IBM 7291
  - IBM 7292
  - IBM 7293
  - IBM 7294
  - IBM 7295
  - IBM 7296
  - IBM 7297
  - IBM 7298
  - IBM 7299
  - IBM 7300
  - IBM 7301
  - IBM 7302
  - IBM 7303
  - IBM 7304
  - IBM 7305
  - IBM 7306
  - IBM 7307
  - IBM 7308
  - IBM 7309
  - IBM 7310
  - IBM 7311
  - IBM 7312
  - IBM 7313
  - IBM 7314
  - IBM 7315
  - IBM 7316
  - IBM 7317
  - IBM 7318
  - IBM 7319
  - IBM 7320
  - IBM 7321
  - IBM 7322
  - IBM 7323
  - IBM 7324
  - IBM 7325
  - IBM 7326
  - IBM 7327
  - IBM 7328
  - IBM 7329
  - IBM 7330
  - IBM 7331
  - IBM 7332
  - IBM 7333
  - IBM 7334
  - IBM 7335
  - IBM 7336
  - IBM 7337
  - IBM 7338
  - IBM 7339
  - IBM 7340
  - IBM 7341
  - IBM 7342
  - IBM 7343
  - IBM 7344
  - IBM 7345
  - IBM 7346
  - IBM 7347
  - IBM 7348
  - IBM 7349
  - IBM 7350
  - IBM 7351
  - IBM 7352
  - IBM 7353
  - IBM 7354
  - IBM 7355
  - IBM 7356
  - IBM 7357
  - IBM 7358
  - IBM 7359
  - IBM 7360
  - IBM 7361
  - IBM 7362
  - IBM 7363
  - IBM 7364
  - IBM 7365
  - IBM 7366
  - IBM 7367
  - IBM 7368
  - IBM 7369
  - IBM 7370
  - IBM 7371
  - IBM 7372
  - IBM 7373
  - IBM 7374
  - IBM 7375
  - IBM 7376
  - IBM 7377
  - IBM 7378
  - IBM 7379
  - IBM 7380
  - IBM 7381
  - IBM 7382
  - IBM 7383
  - IBM 7384
  - IBM 7385
  - IBM 7386
  - IBM 7387
  - IBM 7388
  - IBM 7389
  - IBM 7390
  - IBM 7391
  - IBM 7392
  - IBM 7393
  - IBM 7394
  - IBM 7395
  - IBM 7396
  - IBM 7397
  - IBM 7398
  - IBM 7399
  - IBM 7400
  - IBM 7401
  - IBM 7402
  - IBM 7403
  - IBM 7404
  - IBM 7405
  - IBM 7406
  - IBM 7407
  - IBM 7408
  - IBM 7409
  - IBM 7410
  - IBM 7411
  - IBM 7412
  - IBM 7413
  - IBM 7414
  - IBM 7415
  - IBM 7416
  - IBM 7417
  - IBM 7418
  - IBM 7419
  - IBM 7420
  - IBM 7421
  - IBM 7422
  - IBM 7423
  - IBM 7424
  - IBM 7425
  - IBM 7426
  - IBM 7427
  - IBM 7428
  - IBM 7429
  - IBM 7430
  - IBM 7431
  - IBM 7432
  - IBM 7433
  - IBM 7434
  - IBM 7435
  - IBM 7436
  - IBM 7437
  - IBM 7438
  - IBM 7439
  - IBM 7440
  - IBM 7441
  - IBM 7442
  - IBM 7443
  - IBM 7444
  - IBM 7445
  - IBM 7446
  - IBM 7447
  - IBM 7448
  - IBM 7449
  - IBM 7450
  - IBM 7451
  - IBM 7452
  - IBM 7453
  - IBM 7454
  - IBM 7455
  - IBM 7456
  - IBM 7457
  - IBM 7458
  - IBM 7459
  - IBM 7460
  - IBM 7461
  - IBM 7462
  - IBM 7463
  - IBM 7464
  - IBM 7465
  - IBM 7466
  - IBM 7467
  - IBM 7468
  - IBM 7469
  - IBM 7470
  - IBM 7471
  - IBM 7472
  - IBM 7473
  - IBM 7474
  - IBM 7475
  - IBM 7476
  - IBM 7477
  - IBM 7478
  - IBM 7479
  - IBM 7480
  - IBM 7481
  - IBM 7482
  - IBM 7483
  - IBM 7484
  - IBM 7485
  - IBM 7486
  - IBM 7487
  - IBM 7488
  - IBM 7489
  - IBM 7490
  - IBM 7491
  - IBM 7492
  - IBM 7493
  - IBM 7494
  - IBM 7495
  - IBM 7496
  - IBM 7497
  - IBM 7498
  - IBM 7499
  - IBM 7500
  - IBM 7501
  - IBM 7502
  - IBM 7503
  - IBM 7504
  - IBM 7505
  - IBM 7506
  - IBM 7507
  - IBM 7508
  - IBM 7509
  - IBM 7510
  - IBM 7511
  - IBM 7512
  - IBM 7513
  - IBM 7514
  - IBM 7515
  - IBM 7516
  - IBM 7517
  - IBM 7518
  - IBM 7519
  - IBM 7520
  - IBM 7521
  - IBM 7522
  - IBM 7523
  - IBM 7524
  - IBM 7525
  - IBM 7526
  - IBM 7527
  - IBM 7528
  - IBM 7529
  - IBM 7530
  - IBM 7531
  - IBM 7532
  - IBM 7533
  - IBM 7534
  - IBM 7535
  - IBM 7536
  - IBM 7537
  - IBM 7538
  - IBM 7539
  - IBM 7540
  - IBM 7541
  - IBM 7542
  - IBM 7543
  - IBM 7544
  - IBM 7545
  - IBM 7546
  - IBM 7547
  - IBM 7548
  - IBM 7549
  - IBM 7550
  - IBM 7551
  - IBM 7552
  - IBM 7553
  - IBM 7554
  - IBM 7555
  - IBM 7556
  - IBM 7557
  - IBM 7558
  - IBM 7559
  - IBM 7560
  - IBM 7561
  - IBM 7562
  - IBM 7563
  - IBM 7564
  - IBM 7565
  - IBM 7566
  - IBM 7567
  - IBM 7568
  - IBM 7569
  - IBM 7570
  - IBM 7571
  - IBM 7572
  - IBM 7573
  - IBM 7574
  - IBM 7575
  - IBM 7576
  - IBM 7577
  - IBM 7578
  - IBM 7579
  - IBM 7580
  - IBM 7581
  - IBM 7582
  - IBM 7583
  - IBM 7584
  - IBM 7585
  - IBM 7586
  - IBM 7587
  - IBM 7588
  - IBM 7589
  - IBM 7590
  - IBM 7591
  - IBM 7592
  - IBM 7593
  - IBM 7594
  - IBM 7595
  - IBM 7596
  - IBM 7597
  - IBM 7598
  - IBM 7599
  - IBM 7600
  - IBM 7601
  - IBM 7602
  - IBM 7603
  - IBM 7604
  - IBM 7605
  - IBM 7606
  - IBM 7607
  - IBM 7608
  - IBM 7609
  - IBM 7610
  - IBM 7611
  - IBM 7612
  - IBM 7613
  - IBM 7614
  - IBM 7615
  - IBM 7616
  - IBM 7617
  - IBM 7618
  - IBM 7619
  - IBM 7620
  - IBM 7621
  - IBM 7622
  - IBM 7623
  - IBM 7624
  - IBM 7625
  - IBM 7626
  - IBM 7627
  - IBM 7628
  - IBM 7629
  - IBM 7630
  - IBM 7631
  - IBM 7632
  - IBM 7633
  - IBM 7634
  - IBM 7635
  - IBM 7636
  - IBM 7637
  - IBM 7638
  - IBM 7639
  - IBM 7640
  - IBM 7641
  - IBM 7642
  - IBM 7643
  - IBM 7644
  - IBM 7645
  - IBM 7646
  - IBM 7647
  - IBM 7648
  - IBM 7649
  - IBM 7650
  - IBM 7651
  - IBM 7652
  - IBM 7653
  - IBM 7654
  - IBM 7655
  - IBM 7656
  - IBM 7657
  - IBM 7658
  - IBM 7659
  - IBM 7660
  - IBM 7661
  - IBM 7662
  - IBM 7663
  - IBM 7664
  - IBM 7665
  - IBM 7666
  - IBM 7667
  - IBM 7668
  - IBM 7669
  - IBM 7670
  - IBM 7671
  - IBM 7672
  - IBM 7673
  - IBM 7674
  - IBM 7675
  - IBM 7676
  - IBM 7677
  - IBM 7678
  - IBM 7679
  - IBM 7680
  - IBM 7681
  - IBM 7682
  - IBM 7683
  - IBM 7684
  - IBM 7685
  - IBM 7686
  - IBM 7687
  - IBM 7688
  - IBM 7689
  - IBM 7690
  - IBM 7691
  - IBM 7692
  - IBM 7693
  - IBM 7694
  - IBM 7695
  - IBM 7696
  - IBM 7697
  - IBM 7698
  - IBM 7699
  - IBM 7700
  - IBM 7701
  - IBM 7702
  - IBM 7703
  - IBM 7704
  - IBM 7705
  - IBM 7706
  - IBM 7707
  - IBM 7708
  - IBM 7709
  - IBM 7710
  - IBM 7711
  - IBM 7712
  - IBM 7713
  - IBM 7714
  - IBM 7715
  - IBM 7716
  - IBM 7717
  - IBM 7718
  - IBM 7719
  - IBM 7720
  - IBM 7721
  - IBM 7722
  - IBM 7723
  - IBM 7724
  - IBM 7725
  - IBM 7726
  - IBM 7727
  - IBM 7728
  - IBM 7729
  - IBM 7730
  - IBM 7731
  - IBM 7732
  - IBM 7733
  - IBM 7734
  - IBM 7735
  - IBM 7736
  - IBM 7737
  - IBM 7738
  - IBM 7739
  - IBM 7740
  - IBM 7741
  - IBM 7742
  - IBM 7743
  - IBM 7744
  - IBM 7745
  - IBM 7746
  - IBM 7747
  - IBM 7748
  - IBM 7749
  - IBM 7750
  - IBM 7751
  - IBM 7752
  - IBM 7753
  - IBM 7754
  - IBM 7755
  - IBM 7756
  - IBM 7757
  - IBM 7758
  - IBM 7759
  - IBM 7760
  - IBM 7761
  - IBM 7762
  - IBM 7763
  - IBM 7764
  - IBM 7765
  - IBM 7766
  - IBM 7767
  - IBM 7768
  - IBM 7769
  - IBM 7770
  - IBM 7771
  - IBM 7772
  - IBM 7773
  - IBM 7774
  - IBM 7775
  - IBM 7776
  - IBM 7777
  - IBM 7778
  - IBM 7779
  - IBM 7780
  - IBM 7781
  - IBM 7782
  - IBM 7783
  - IBM 7784
  - IBM 7785
  - IBM 7786
  - IBM 7787
  - IBM 7788
  - IBM 7789
  - IBM 7790
  - IBM 7791
  - IBM 7792
  - IBM 7793
  - IBM 7794
  - IBM 7795
  - IBM 7796
  - IBM 7797
  - IBM 7798
  - IBM 7799
  - IBM 7800
  - IBM 7801
  - IBM 7802
  - IBM 7803
  - IBM 7804
  - IBM 7805
  - IBM 7806
  - IBM 7807
  - IBM 7808
  - IBM 7809
  - IBM 7810
  - IBM 7811
  - IBM 7812
  - IBM 7813
  - IBM 7814
  - IBM 7815
  - IBM 7816
  - IBM 7817
  - IBM 7818
  - IBM 7819
  - IBM 7820
  - IBM 7821
  - IBM 7822
  - IBM 7823
  - IBM 7824
  - IBM 7825
  - IBM 7826
  - IBM 7827
  - IBM 7828
  - IBM 7829
  - IBM 7830
  - IBM 7831
  - IBM 7832
  - IBM 7833
  - IBM 7834
  - IBM 7835
  - IBM 7836
  - IBM 7837
  - IBM 7838
  - IBM 7839
  - IBM 7840
  - IBM 7841
  - IBM 7842
  - IBM 7843
  - IBM 7844
  - IBM 7845
  - IBM 7846
  - IBM 7847
  - IBM 7848
  - IBM 7849
  - IBM 7850
  - IBM 7851
  - IBM 7852
  - IBM 7853
  - IBM 7854
  - IBM 7855
  - IBM 7856
  - IBM 7857
  - IBM 7858
  - IBM 7859
  - IBM 7860
  - IBM 7861
  - IBM 7862
  - IBM 7863
  - IBM 7864
  - IBM 7865
  - IBM 7866
  - IBM 7867
  - IBM 7868
  - IBM 7869
  - IBM 7870
  - IBM 7871
  - IBM 7872
  - IBM 7873
  - IBM 7874
  - IBM 7875
  - IBM 7876
  - IBM 7877
  - IBM 7878
  - IBM 7879
  - IBM 7880
  - IBM 7881
  - IBM 7882
  - IBM 7883
  - IBM 7884
  - IBM 7885
  - IBM 7886
  - IBM 7887
  - IBM 7888
  - IBM 7889
  - IBM 7890
  - IBM 7891
  - IBM 7892
  - IBM 7893
  - IBM 7894
  - IBM 7895
  - IBM 7896
  - IBM 7897
  - IBM 7898
  - IBM 7899
  - IBM 7900
  - IBM 7901
  - IBM 7902
  - IBM 7903
  - IBM 7904
  - IBM 7905
  - IBM 7906
  - IBM 7907
  - IBM 7908
  - IBM 7909
  - IBM 7910
  - IBM 7911
  - IBM 7912
  - IBM 7913
  - IBM 7914
  - IBM 7915
  - IBM 7916
  - IBM 7917
  - IBM 7918
  - IBM 7919
  - IBM 7920
  - IBM 7921
  - IBM 7922
  - IBM 7923
  - IBM 7924
  - IBM 7925
  - IBM 7926
  - IBM 7927
  - IBM 7928
  - IBM 7929
  - IBM 7930
  - IBM 7931
  - IBM 7932
  - IBM 7933
  - IBM 7934
  - IBM 7935
  - IBM 7936
  - IBM 7937
  - IBM 7938
  - IBM 7939
  - IBM 7940
  - IBM 7941
  - IBM 7942
  - IBM 7943
  - IBM 7944
  - IBM 7945
  - IBM 7946
  - IBM 7947
  - IBM 7948
  - IBM 7949
  - IBM 7950
  - IBM 7951
  - IBM 7952
  - IBM 7953
  - IBM 7954
  - IBM 7955
  - IBM 7956
  - IBM 7957
  - IBM 7958
  - IBM 7959
  - IBM 7960
  - IBM 7961
  - IBM 7962
  - IBM 7963
  - IBM 7964
  - IBM 7965
  - IBM 7966
  - IBM 7967
  - IBM 7968
  - IBM 7969
  - IBM 7970
  - IBM 7971
  - IBM 7972
  - IBM 7973
  - IBM 7974
  - IBM 7975
  - IBM 7976
  - IBM 7977
  - IBM 7978
  - IBM 7979
  - IBM 7980
  - IBM 7981
  - IBM 7982
  - IBM 7983
  - IBM 7984
  - IBM 7985
  - IBM 7986
<

## 1948-1970 :Hardware expensive; human labor cheap

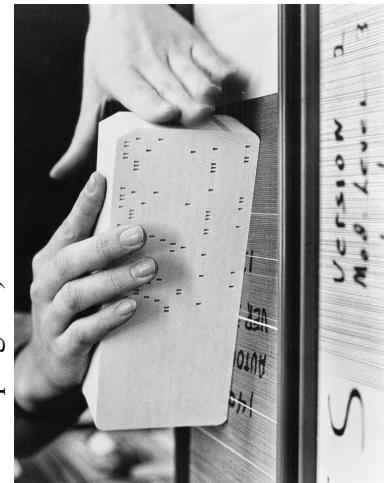
- Computer 1-5 M\$ : Nation 's property, mainly used for military's purposes ⇒ Require optimization for using hardware effectively
- Lack of human-machine interact
- User, programmer; operator are same group of people
- 1 user at a single time



VIỆT CỘNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## 1948-1970 :Hardware expensive; human labor cheap

- User wrote program on punched cards
- First card is bootstraps loader is loaded into memory and executed
- Instructions in bootstraps loader fetch into memory and execute instructions on other later cards (application program)
- Check light bulb for results, debug



## view of change

## Operating systems development's history

- History of electronic computer
- Operating systems development's history



VIỆT CỘNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

- Debugging is difficult
- Waste processor time
- Solution: batch processing

## The first Operating System

### GM-NAA I/O

General Motors and North American Aviation  
Input Output System.

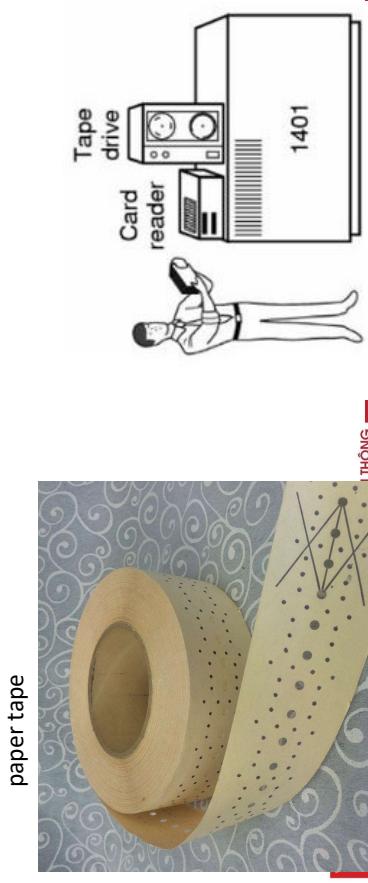
- Created in 1956 by Robert L. Patrick of Owen Mock of North American Aviation.
- Main function: automatically execute a new program once the one that was being executed had finished (batch processing).
- Formed of shared routines to the programs that provided common access to the input/output devices.
- Installed on 40 IBM 704 computers



VIỆT NAM INSTITUTE OF TECHNOLOGY  
VIỆT NAM  
VIỆT NAM INSTITUTE OF TECHNOLOGY  
VIỆT NAM INSTITUTE OF TECHNOLOGY

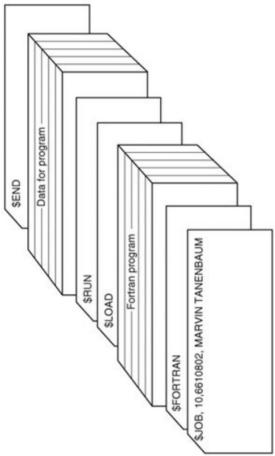
### Independent computer for reading/writing data

- Replace card reader by tape ⇒ Independent external computers for reading/writing data



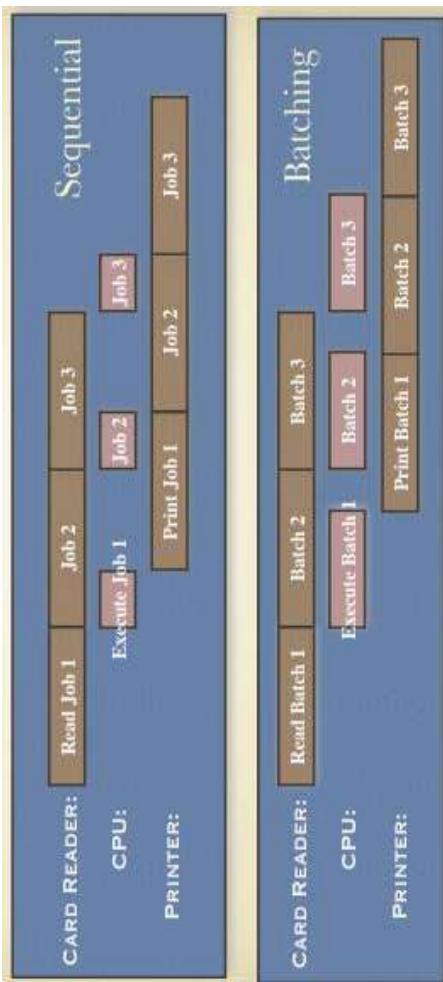
## Batch processing and professional operator

- Programmer give program to operator
- Operator groups program into a single pack (batch)
- Computer reads and run each program consequently
- Operator takes the result, prints out and gives to programmer



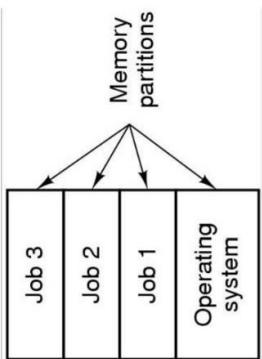
VIỆT NAM INSTITUTE OF TECHNOLOGY  
VIỆT NAM  
VIỆT NAM INSTITUTE OF TECHNOLOGY  
VIỆT NAM INSTITUTE OF TECHNOLOGY

## Batch and sequential processing



## Multi programming

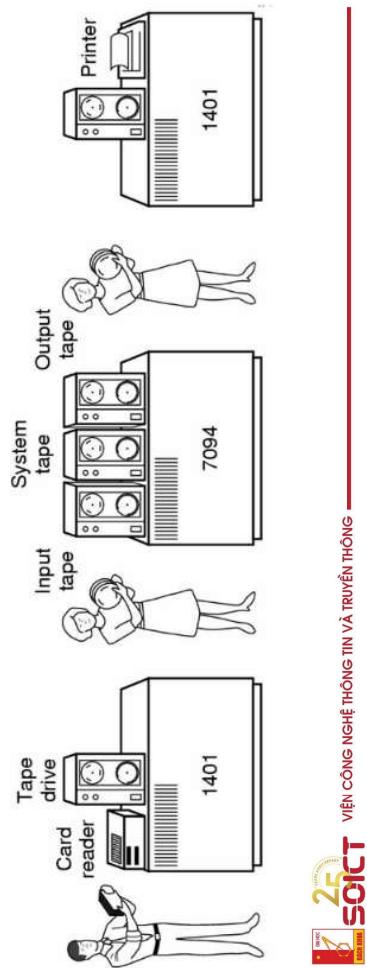
- Hardware: memory space larger and cheaper => Some programs can run simultaneously -> multi programming



- More overlap between computing and I/O
- Require memory protection between programs and keep one crashed program from damaging the system
- Problem: OS must manage all interactions => out of control (OS360: 1000 errors)

## Independent computer for reading/writing data

- Replace card reader by tape => Independent external computers for reading/writing data



VIỆT NAM CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## 1970-1981 :

- Computers prices about 10.000\$ => used widely for different jobs
- OS technology became stable.

- Using cheap terminal device (1000\$) allow many user to interact with the system at the same time

- CPU is reprogrammed to be switch easily between programs
- => Allow overlap between computing and I/O

## 1970-1981 :

- User perform different works (text editor, chat, program debugging...)  $\Rightarrow$  require system to be exploited effectively
- Example: a PC: 10M calculation/s; typing speed 0.2s/1 character  $\Rightarrow$  lost 2M calculation per one typing  
 $\Rightarrow$  Time sharing operating system
  - Problem: system's response time

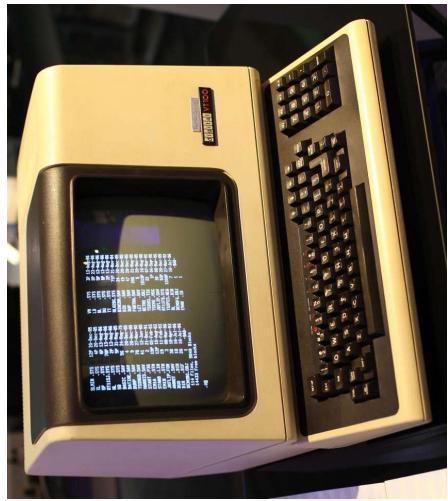
- Computer network was born (ARPANet : 1968) Communication between computer; Protection against network attack



VIỆT CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## 1970-1981 :

- video display terminal
- Ex: DEC VT100 (1978)

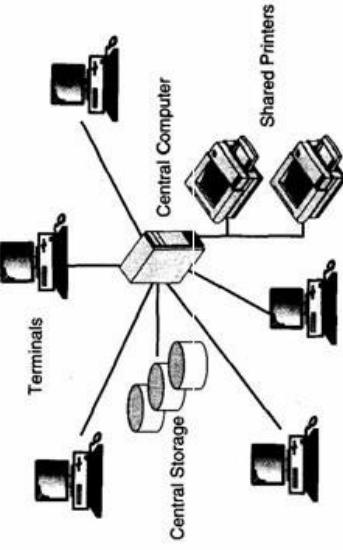


## 1981-1995 :

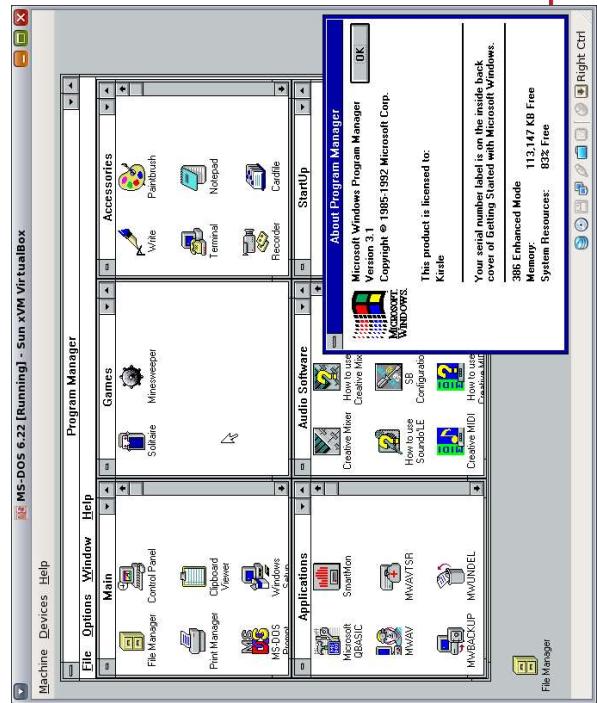
- Computers prices about 1000\$;
- Human labor 100K \$/year

If you can make someone 1% more efficient by giving them a computer, it's worth it!

$\Rightarrow$  Computers are used more widely for working more effectively



Time-sharing Environment

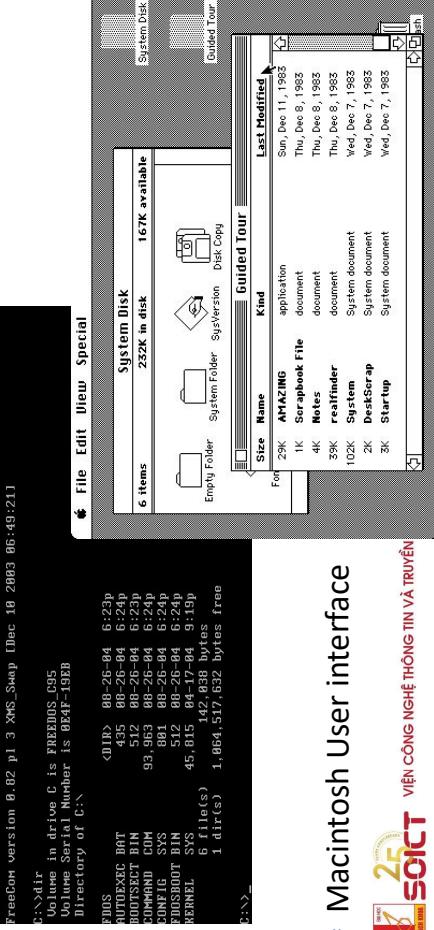
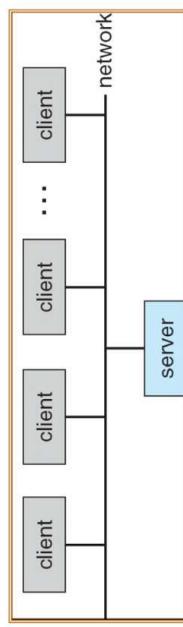


## 1981-1995 :

- Personal computing
  - Cheap computer, single person can afford (PC).
  - OS on PC
- Hardware resources are limited (Early : 1980s)
  - OS become library of available procedures
  - Run 1 program at a time (DOS)
  - PC become more powerful
  - OS meet complex problems: multi tasking, memory protection... (WINXP)
  - Graphical user interface (MAC, WIN,..)

## Distributed systems

- Development time of networking and distributed operating systems



\* Macintosh User Interface



- Local area network
  - Computers share resources: printer, File servers,..
  - Client / Server model
  - Services
    - Computing, storage
    - Services provided through Internet.

## Mobile devices

- Wide area network, wireless network
- Traditional computer divided into many components (wireless keyboards, mouse, remoting storage)
- peer-to-peer system
  - Devices with the same role working together
  - “Operating system’s” components are spread globally
- Cloud computing
  - Cloud operating system
  - Ex: Microsoft Windows Azure, Google Chrome OS

## Conclusion

- The development of the operating systems are strongly connected with the computers' development
- Operating systems have to change with changing technology

## Distributed systems

- Problems
  - Transmission delay; bandwidth, reliable...
  - Virus (love letter virus 05/2000),...
  - >45 millions computers were infected
  - Stole information
  - Auto send emails from contact list
  - Download Trojan
-  VIỆT NAM CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## Mobile devices

- Mobile devices become more popular
- Phone, Laptop, PDA . . .
  - Small, changeable and cheap → More computers/human
  - Limited ability: speed, memory,...



# Chapter 1 Operating System Overview

Chapter 1. Operating System Overview  
3. Operating System definition and classification  
3.1. Definition of operating system

## Observer's perspective

- Different objects have different requirements for OS
- Different observing perspectives ⇒ different definitions
  - ❖ User
  - ❖ Manager
  - ❖ Technical perspective
  - ❖ System engineer perspective

Chapter 1. Operating System Overview  
3. Operating System definition and classification  
3.1. Definition of operating system

## User's perspective:



“A system of programs that help exploit the computing system conveniently”

Operating System definition and classification

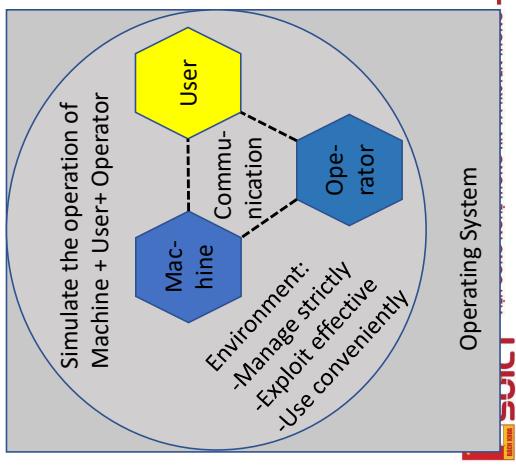
- Definitions
- Classifications

## System engineer's perspective

“A system of programs that model and simulate the operation of the computer, user, and operators. It works in a communicating mode to make a convenient environment for exploiting the computer system and maximum resource management.”

## System engineer's perspective

- Simulate 3 roles ⇒ require 3 types of languages
  - **Machine language**
    - The only working language of the system
    - All other languages have to be translated into machine language
  - **System operation's language**
    - OS commands (DOS: Dir, Del...; Unix: ls, rm,...)
    - Translated by the Shell
  - **Algorithm language**
    - Programming language
    - Compiler



## Manager's perspective

“A system of programs that help manage the computing system's resources effectively”



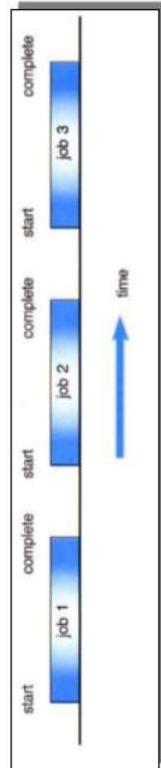
## Manager's perspective

- “A system of programs that help manage the computing system's resources effectively”



## Batch processing single program system

- Programs are performed **consequently** follow **predetermined instructions**
- When a program finished, the system **automatically** run the **next program** without any external intervention



- Require a **special process** to supervise the sequence of jobs and the supervisor must stay permanent in the **memory**
- Need to **organize** a job queue



## Operating System definition and classification

- Definitions
- Classifications

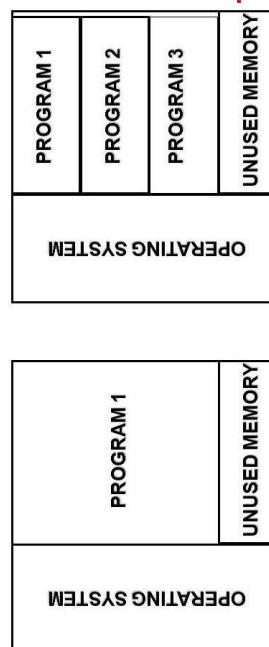
## Batch processing multi-program system

- Problem: when a program access an **I/O device**, **processor** has to wait
- Solution: allow **many programs** to run at the same time

## MULTIPROGRAMMING

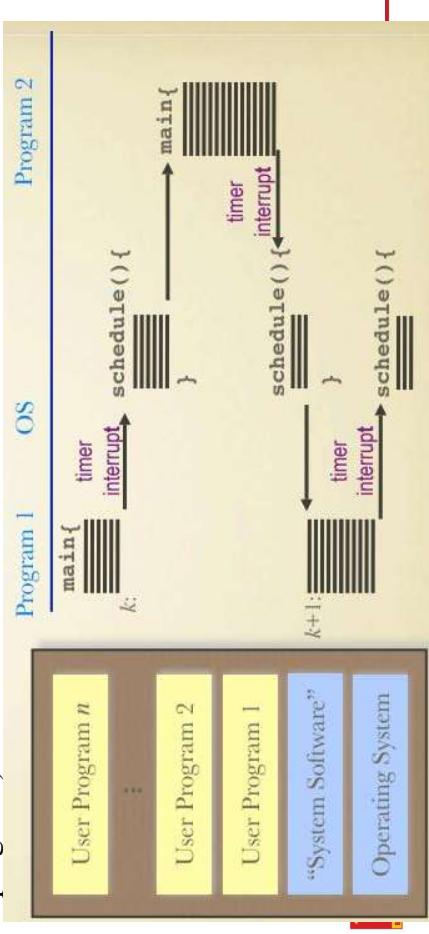
TRADITIONAL SINGLE-  
PROGRAM SYSTEM

MULTIPROGRAMMING  
ENVIRONMENT



## Time sharing system

- Processor's usage allowance time is **shared** among **ready-to-run programs**
- Similar to batch processing multi program system (only load part of the programs)



## Time sharing system

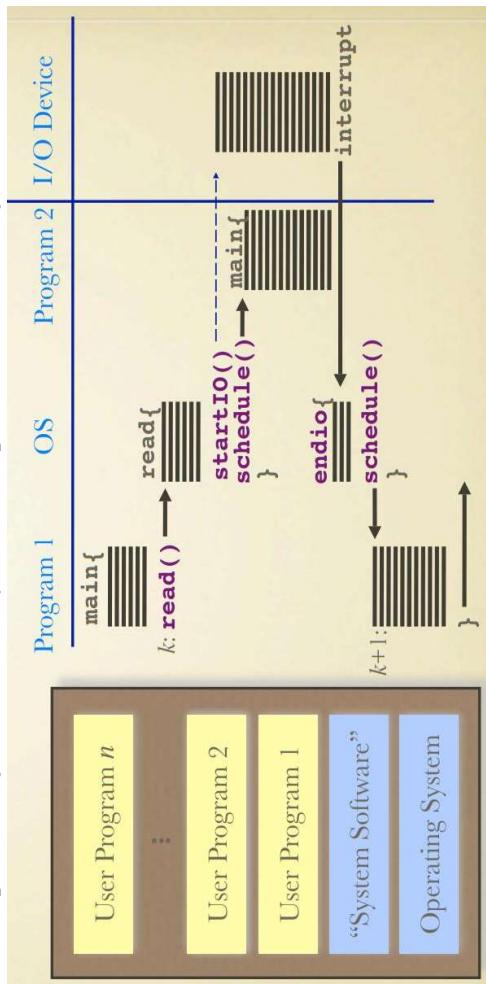
- Processor is issued mainly** by the **OS**  $\Rightarrow$  how ?  $\Rightarrow$  Chapter 2

- Swapping time** between programs are **small**  $\rightarrow$  programs seem to run **parallel**

- Usually called: **Multi tasking** operating system (Windows, Unix)
- How to solve the **memory sharing problem** between programs ?

## Batch processing multi-program system

- Keep several jobs in memory and multiplex CPU between jobs

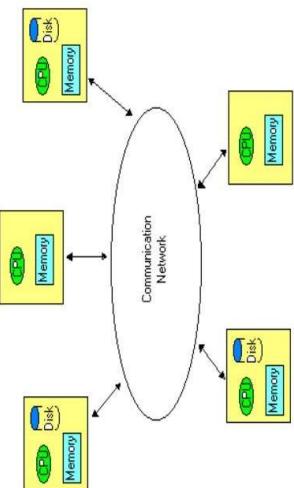


## Batch processing multi-program system

- Save **memory** (no need to load all the program into the memory)
- Reduce processor spare-time
- High cost for **processor scheduling**. Which program can use processor next?

## Distributed system

- Each processor has a **local memory** and **communicate via transmission lines**
- Processors are **different** from sizes to functions (personal machine, workstation, mini computer,..)

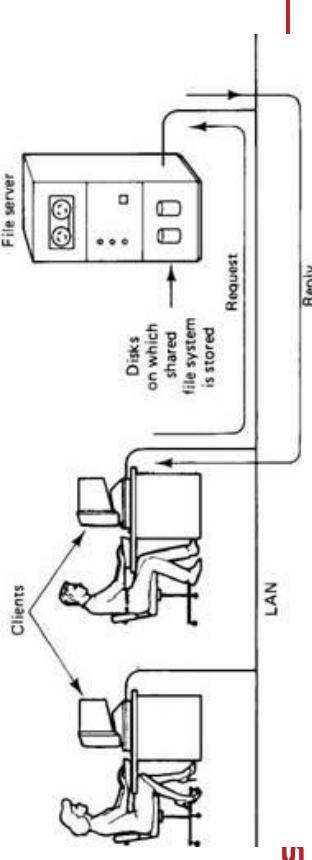


### Architecture of Distributed OS

- Ex: Solaris, AIX

## Distributed system

- Used for**
  - Resource sharing**
  - Increase computing speed**
  - Safety**



## Parallel system

- Constructed for system that has **many processors**
  - Many processors, **works** are **done faster**
  - More **reliable**: one processor breaks down will not affect the system
  - Advantage over single processor computer due to **memory, peripheral devices sharing...**

### Symmetric multi processing (SMP: symmetric)

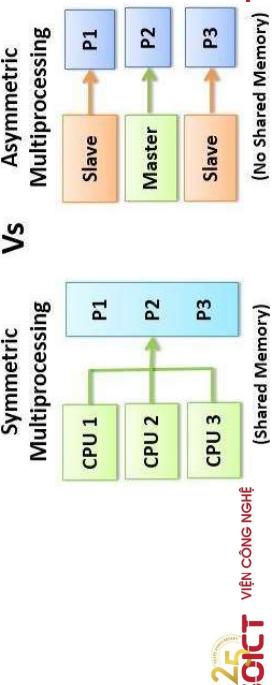
- Each processor run a single program
- Processors **communicate** via a **shared memory**
  - Fault tolerance** mechanism and optimal load balance
- Problem:** processor synchronization
- Example: WinNT operating system

## Parallel system

### (cont.)

## Parallel system

- Asymmetric multi processing (ASMP: asymmetric)
  - One** processor **controls** the **whole system**
  - Other** processors **follow** the main processor's **commands** or predetermined instructions
  - This model has the **master-slave relation** form: The main process will make schedule for other processors
  - Example: IBM System/360



## Concepts in operating systems

- Process
- System's resources
  - Shell
  - Kernel
- System calls

## Process

- A running program
  - Codes: Program's executable instruction
  - Program's data
  - Stack, stack pointer, registers
  - Information that is necessary for running program
- Process  $\gg$  program
  - Program: a passive object, contains computer's instructions to perform a specific task
  - Process: program's active state.



## Real-time processing system

- Used mainly in **controlling** field.
- Solve a problem **no late** than a specific time.
  - Each problem has a **deadline**
    - The system must **generate** **correct result** in a determined time period
- This OS requires **high cooperation** between **software** and **hardware**.
  - Example: VxWorks, RT Linux

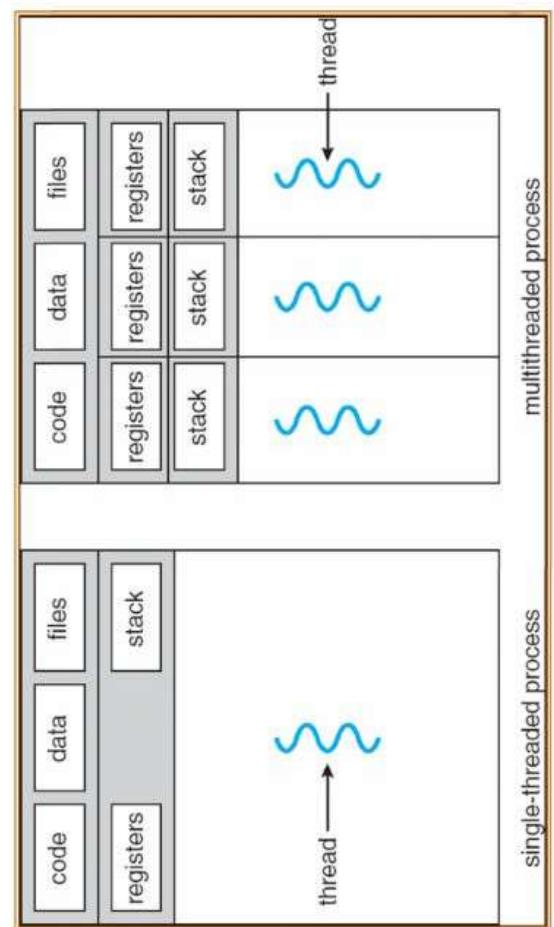
# Chapter 1 Operating System Overview

- ① Operating system Definition
- ② History of Operating Systems
- ③ Definition and Classifications
- ④ Concepts in operating systems
- ⑤ Operating System structures
- ⑥ Basic Properties of Operating Systems
- ⑦ Principles of Operating Systems

## Thread

- A sequence/thread of instructions executed in the program
  - Executable code, data
  - Instruction pointer, stack, registers
- Heavyweight process: contains 1 thread
- Lightweight process: contains more than 1 thread
- Multi\_ Threading model:
  - Threads running parallel, sharing process's global variables

BASIS FOR COMPARISON	PROGRAM	PROCESS
Basic	Program is a set of instruction.	When a program is executed, it is known as process.
Nature	Passive	Active
Lifespan	Longer	Limited
Required resources	Program is stored on disk in some file and does not require any other resources.	Process holds resources such as CPU, memory address, disk, I/O etc.



## Multi-process timesharing system

- Periodically: OS pauses 1 process and starts another process
  - Need to store processes' information  $\Rightarrow$  process table
  - 1 process can start other process
    - Ex: OS's Shell start a process to perform the command; when the command is done, terminate the started process
    - Process can exchange information
    - 1 process can include many threads

## System's resources

(cont.)

### Processor

- System's most important component
- Access level: instruction
- Processing time
- Multi-processor system: each processor's time is managed and scheduled independently



VIỆT CỘNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

### Concept

### Memory

- Distinguished by: Storage size, directly access, sequential access
- Levelled by main memory/internal; extend, external



VIỆT CỘNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## Concepts in operating systems

- Everything that is necessary for a program to be executed
  - Space: System's storage space
  - Time: Instruction executing time, data accessing time
- System's resources
  - Processor
  - Memory
  - Peripheral devices

### Concept of system resources

VIỆT CỘNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## Virtual resource

- provided to a user program in a modified form
- Appears only when the system needs it or when the system creates it
- Automatically disappears when the system terminates or, more precisely, when the process associated with it has terminated.
- Example: Virtual printer



## System's resources

- Peripheral devices
  - Retrieve, output information (I/O device)
  - Attached to the system via controller
  - Commonly considered peripheral devices-controller devices



## (cont.)

- Peripheral devices
  - Retrieve, output information (I/O device)
  - Attached to the system via controller
  - Commonly considered peripheral devices-controller devices



## Concepts in operating systems

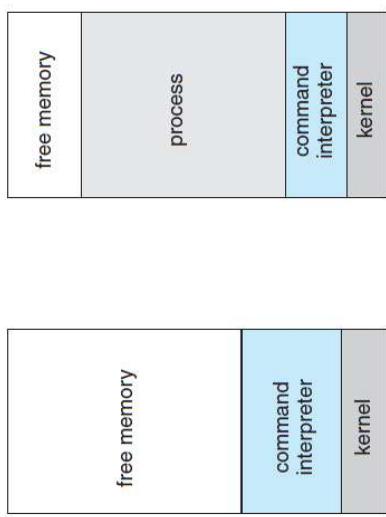
- Process and Thread
- System's resources
- Shell
- Kernel
- System calls

## Resource's classification

- Resource's types
  - Physical : physical devices
  - Logical: variable; virtual devices
- Sharing ability
  - Sharable: at a specific time, it can be allocated for different processes. Example: Memory
  - Non-sharable but dividable: Processes use the resource follow an order; Example: processor
  - Non-sharable and non-dividable : at a specific time, only one process can use the resource. Example: Printer

### Single task environment (MS-DOS)

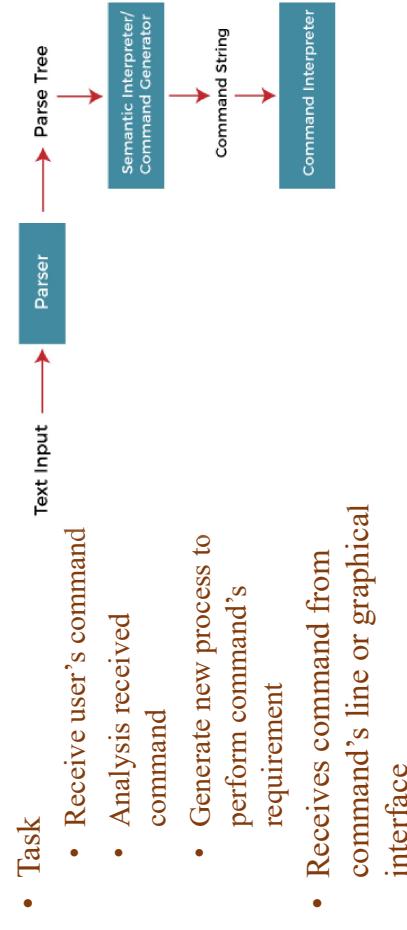
- Shell waits until the process finishes and then receive new command



(a) At system startup.

(b) Running a program

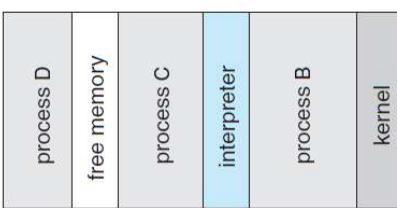
- A special process: user and OS communication environment



Chapter 1 Operating System Overview  
4. Concepts in operating systems  
4.3 Shell

### multi-tasking system (UNIX, WINXP, . . . )

- After creating and running new process, Shell can receive new command



- Shell on Linux

Chapter 1 Operating System Overview  
4. Concepts in operating systems  
4.3 Shell

### multi-tasking system (UNIX, WINXP, . . . )



## Kernel

- A computer program
  - At the lowest level, communicate with the hardware.
  - Control other parts, the core – heart of the OS
  - Loaded first right after the boot loader
  - Translate IO requests from software to a set of CPU instruction
- Hardware protection:
  - User's program directly access -> error notify



VIỆT NAM CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

## Concepts in operating systems

- Process and Thread
- System's resources
- Shell
- Kernel
  - System calls

## Kernel mode

- A privileged mode of operation where the OS has full access to all hardware resources and system components.
  - allows the execution of critical tasks
- Key Characteristics
  - Unlimited Access
  - Execution of Core OS Functions
  - Privileged Instructions
  - Risk of Crashes

## Kernel



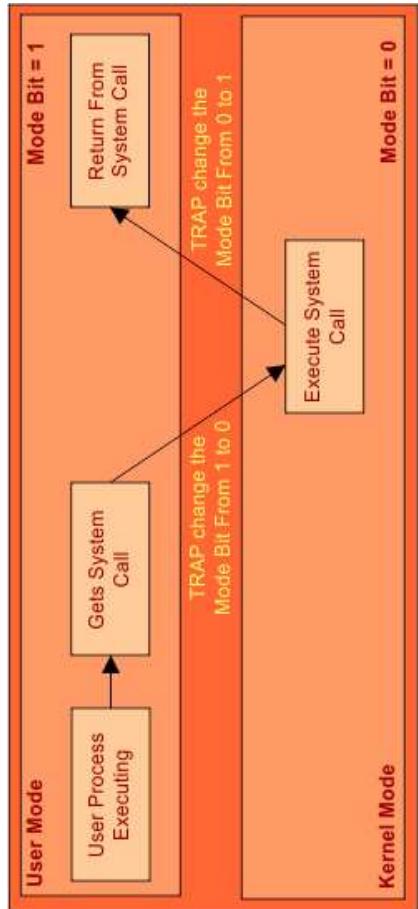
## Concepts in operating systems

- Process
- System's resources
- Shell
- Kernel
- System calls



- Provides environment for interacting between user's program and the OS
  - Programs utilize system calls to request services from OS
    - Create, delete, use other software objects operated by the OS
    - Every single system call is corresponding to a library of sub-programs (functions)
- System calls are done in the form of
  - Instructions in low-levels programming languages
  - Interrupt requests ([Int](#)) in assembly language
  - API functions calls in Windows
- Input parameters for the services and returned results are in special memory areas
  - For example: when making request for an interrupt, the function name is stored in the register AH
  - Int 13/AH=03h : DISK – WRITE/ DISK SECTOR
  - Int 05 : print to monitor ; Int 13/AH=03h : DISK – WRITE/ DISK SECTOR

## Kernel mode and User mode



## User mode

- A **restricted mode of operation** where applications run without direct access to hardware resources.
  - ensure that user programs do not accidentally or maliciously damage the system.

- **Key Characteristics**
  - Restricted Access
  - System Call Interface
  - Protection

## Example

- System's components
- Operating system's services
- System calls
- Operating system's structures

Func BOOL WINAPI ExitWindowsEx(int uFlags, int dwReason);	uFlags	Shutdown types
	EWX_LOGOFF	End process and exit Windows
	EWX_POWEROFF	Shutdown system and turn off computer
	EWX_REBOOT	Shutdown and restart computer
	dwReason	Reason for shutdown

### File log\_off.c

```
#include<windows.h>
int main(int argc, char* argv[]){
    _ExitWindowsEx(EWX_LOGOFF, 0);
    return 0;
}
```



# Chapter 1 Operating System Overview

- ① Operating system Definition
- ② History of Operating Systems
- ③ Definition and Classifications
- ④ Definitions in operating systems
- ⑤ Operating System structures
- ⑥ Basic Properties of Operating Systems
- ⑦ Principles of Operating Systems

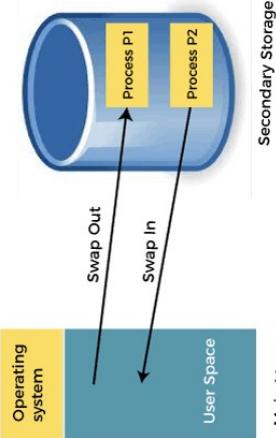


## Main memory management

- Main memory: an array of byte(word);
  - Each element has an **address**; where data are accessed by CPU
  - **To be executed**, a program must be given an **absolute address** and loaded into **main memory**.
  - When the program is running, the **system** accesses instructions and data in main memory.
  - To optimize CPU time and computer's speed, some processes are kept in memory at the same time

## Main memory management (cont.)

- Role of OS in main memory management
  - Store information about used areas in memory and who used them
  - Decide which process will be fetched into main memory when the memory is available.
  - Allocate and retrieve memory when it's necessary



## Process management

- Process: A running program
  - utilize system's resources to complete its task
  - Resources are allocated when process created or while it's running
  - Process terminates, resources are returned
- It is possible for many processes to exist in the system at the same time
  - System process
  - User process

## (cont.)

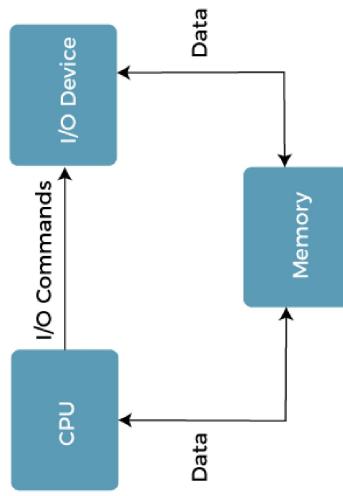
- tasks of OS in process management
  - Create and terminate user's process and system's process
  - Block or re-execute a process
  - Provide mechanism for
    - process synchronization
    - processes' communication
    - controlling deadlock among processes

## File management

- Computer can store information on many types of storage devices
  - File: storage unit
  - File management task
    - Creates/ deletes a file/directory
    - Provides operations over files and directory
    - Reflects file on secondary storage system
    - Backs up file system on storage devices

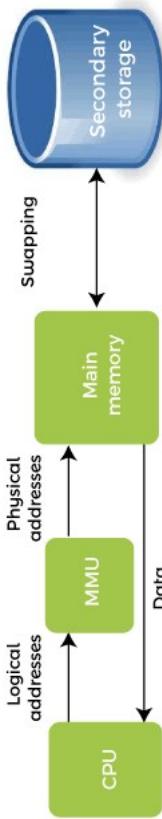
## Input-Output system management

- Objective: hide physical devices' details from users to help them operate easier.



## Storage memory management

- Program is stored in secondary memory (magnetic disk) until it's fetched into main memory and executed.
- Disk is utilized for storing data and processed result.



## Input-Output system management

- Input-Output system management includes
  - Memory management of buffering, caching, spooling
  - Communicate with device drivers.
  - Controller for special hardware devices. Only device driver understand its associated-device's specific structure

## System's protection

- Multi users operate with the system at the same time ⇒ Processes must be protected from other processes' activities
  - Protection is a controlling mechanism of program or user's access to system or resource
  - Protection mechanism will require
    - Distinguish between legal or illegal usage
    - Set imposed controls
    - Provide tools for imposing

## User interface

- Carry out user's command.
- Commands are provided for OS 's command controller to
  - Create and manage process
  - Manage main memory and storage memory
  - Access file system
  - Protect
  - Network system
  - ...
- User interface can be command line (DOS, UNIX) or more friendly with graphical interface (Windows, MacOS)

## Storage memory management

- Data and result can be stored temporarily on disk: virtual memory
  - Role of OS in disk management
    - Unused area management
    - Provide storage area as requested
    - Schedule disk accessing effectively

## Data transmission system (Distributed system)

- Distributed system combined of set of processor (sym/asym) without common clock and memory. Each processor has a local memory.
- Processor connected via transmission network
  - Transmission is performed via protocols (FTP, HTTP...)
  - Distributed system allow user to access different resource
  - Access to sharing resources will allow
    - Increase computing speed
    - Increase data availability

### Some forms of human-computer interact

- Symbol window, icon, desktop



### System's components

#### ● Operating system's services

#### ● System calls

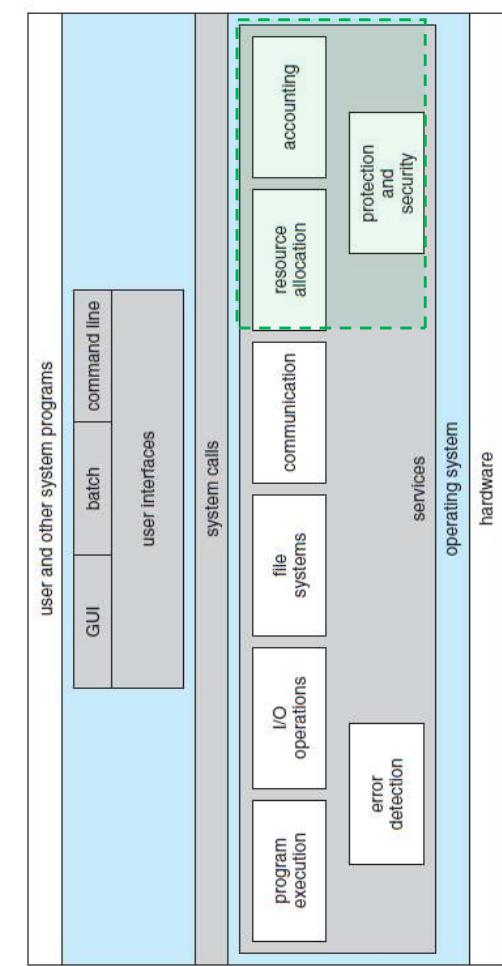
#### ● Operating system's structures

### Some forms of human-computer interact

- Selection table

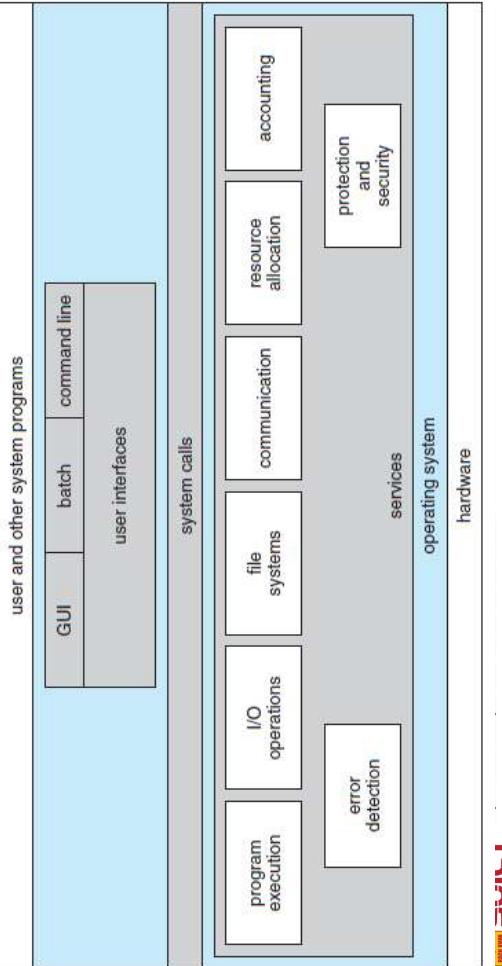


## Support services

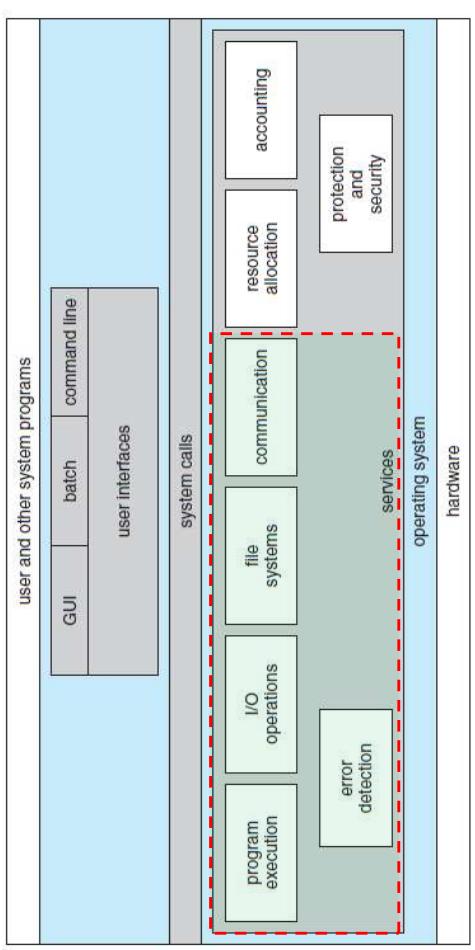


- System's components
- Operating system's services
- System calls
- Operating system's structures

## A view of operating system services

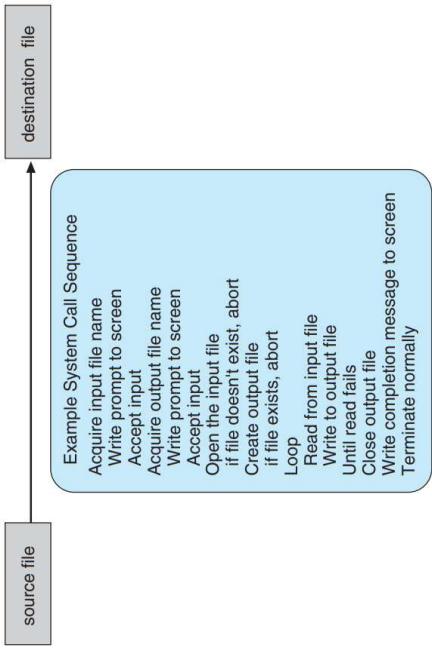


## Main and basic services



## System call

Example of how system calls are used in file copying

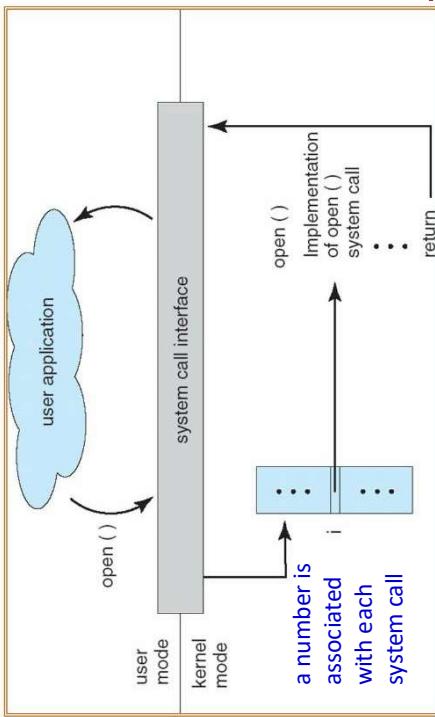
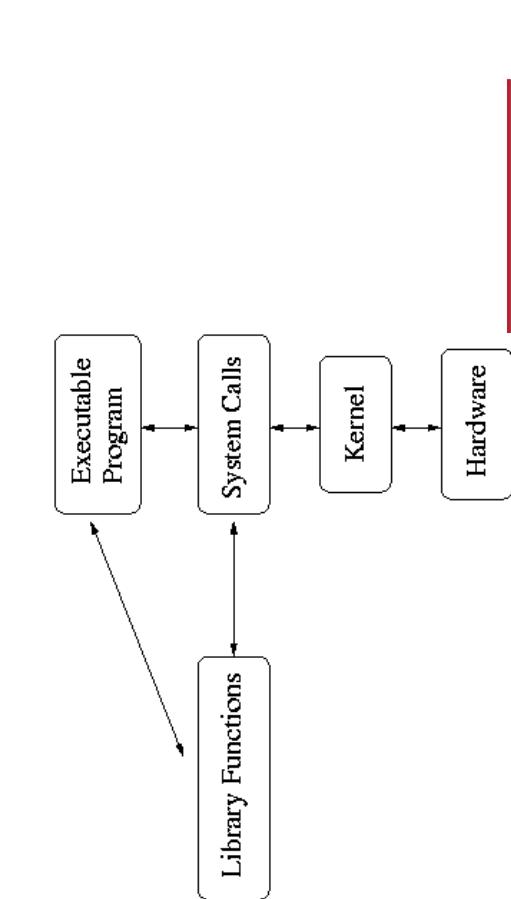


Frequently, systems execute thousands of system calls per second

## System call

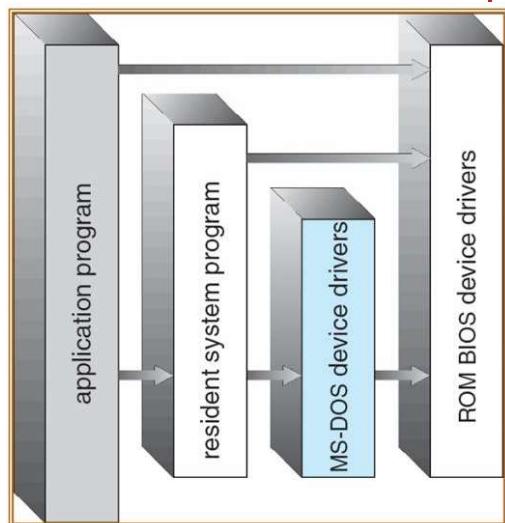
- Process management: initialize, terminate process...
- Memory management: allocate and free memory...
- File management: create, delete, read and write file...
- Input Output device management: perform input/output...
- Exchange information with the system. Example `get/set time/date...`
- Inter process communication

## System call

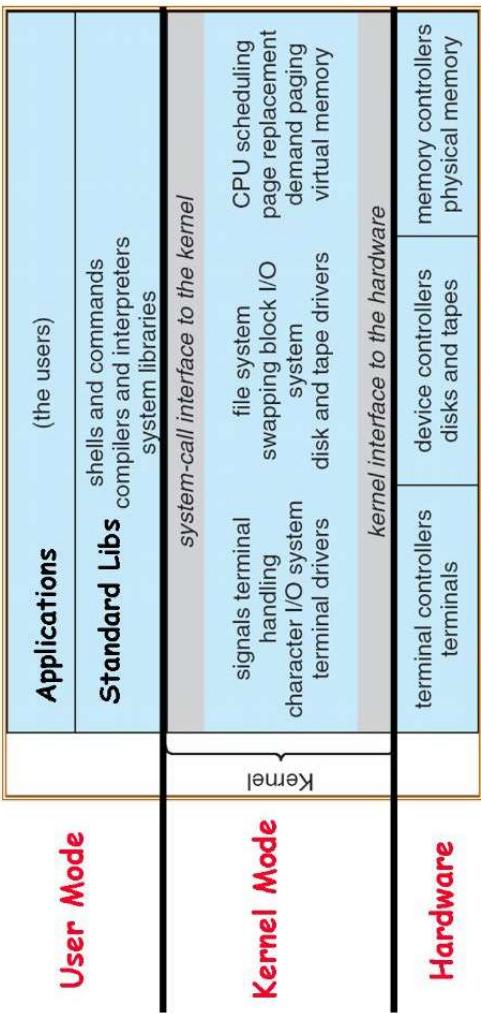


## System call

## MS-DOS structure (*Silberschatz 2002*)

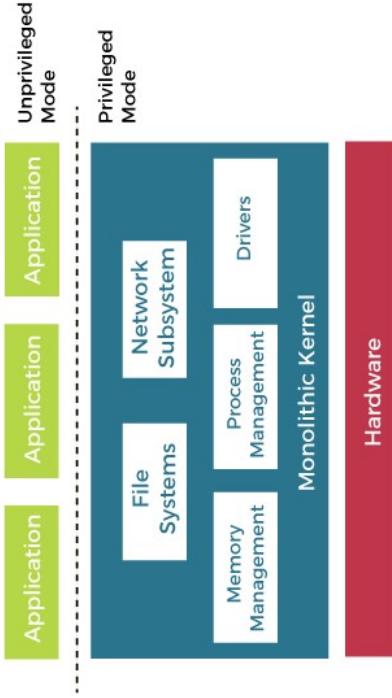


## UNIX structure (*Silberschatz 2002*)

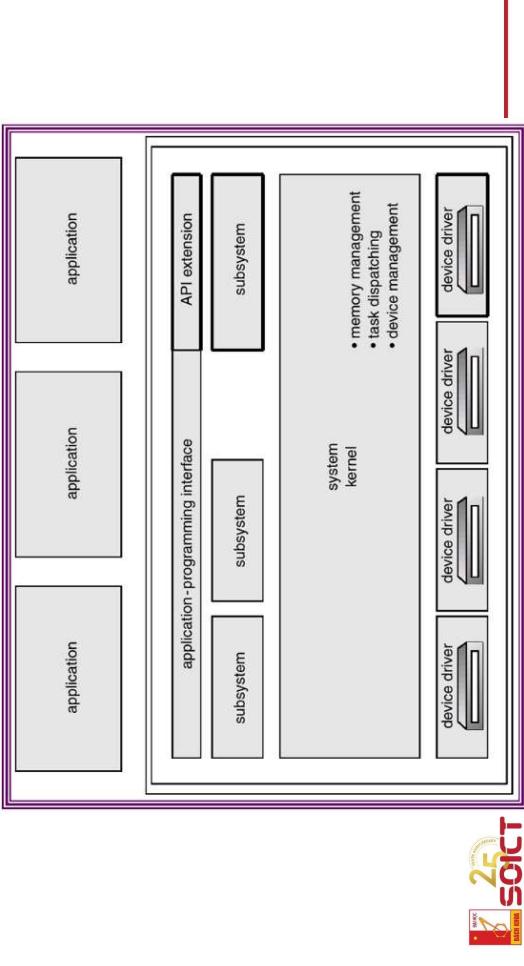


## Monolithic System Architecture

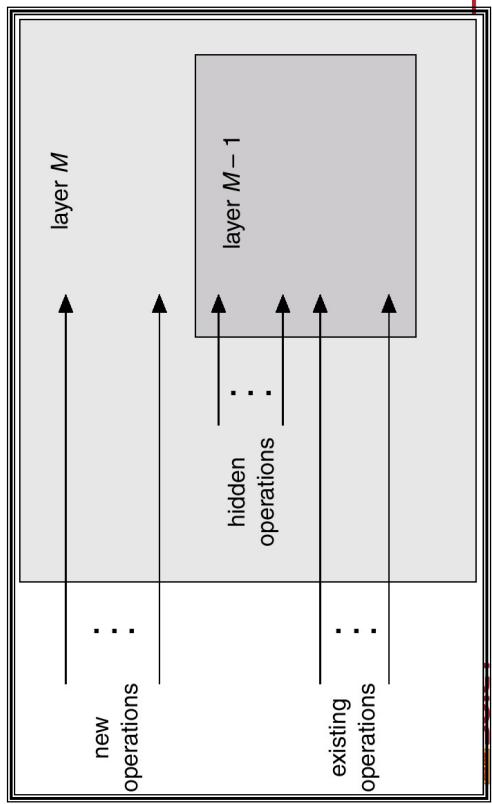
### Monolithic Kernel System



## OS/2 structure (*Silberschatz 2002*)

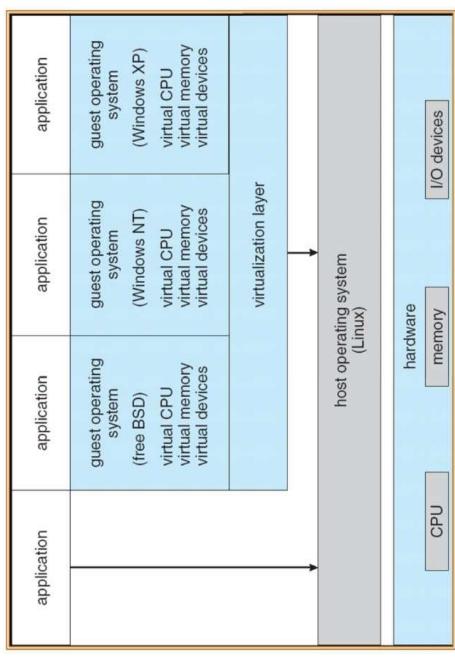


## Layered Approach



## Layered Approach

## Virtual machine(*Silberschatz 2002*)

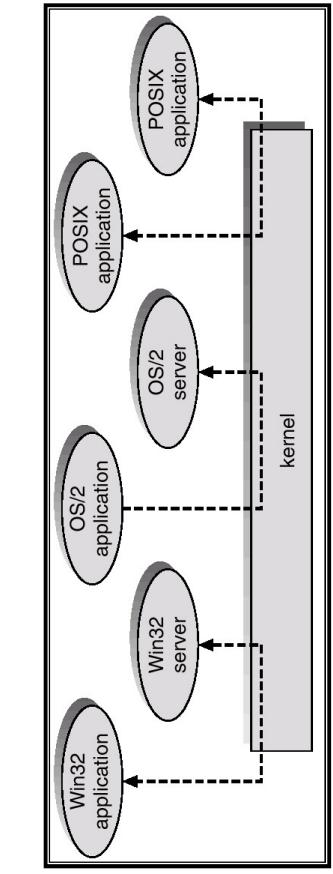


## Layered Approach

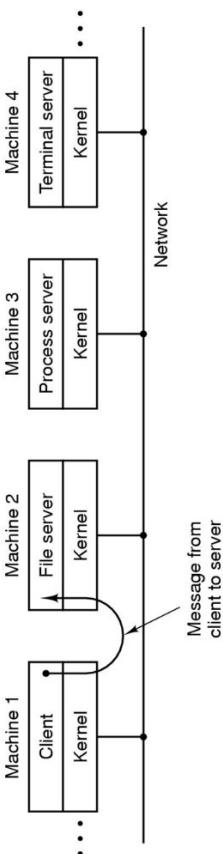
Ex: Structure of THE (Technische Hogeschool Eindhoven), 1968

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

## Windows NT Client-Server Structure)



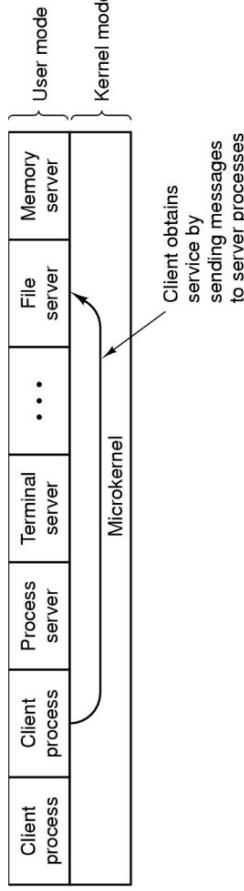
## Client-Server model in distributed OS (Tanenbaum 2001)



## Microkernel System Structure

- Moves as much from the kernel into “user” space.
- Communication takes place between user modules using message passing.
- Benefits:
  - easier to extend a microkernel
  - easier to port the operating system to new architectures
  - more reliable (less code is running in kernel mode)
  - more secure

## Client-Server Model (Tanenbaum 2001)



# Chapter 1 Operating System Overview

Chapter 1 Operating System Overview  
6. Basic Properties of Operating Systems

## High reliability

- Every actions, notifications must be **accurate**
- Only provide information when it's surely correct
  - **When error occurs** notify and stop the process or let the user decide
  - Require support from device
- Example: C:/>COPY C:/F.TXT A:



Chapter 1 Operating System Overview  
6. Basic Properties of Operating Systems

## High reliability

- Example: C:/>COPY C:/F.TXT A:
  - Check the syntax of command copy
  - Check I/O card (motor, drive accessibility)
  - Check for file F.TXT existence in C drive
  - Check A drive
  - Check if file F.TXT already existed in A drive
  - Check if there is enough space in A
  - Check if the disk is write protection
  - Check written information (if required)



Chapter 1 Operating System Overview  
6. Basic Properties of Operating Systems

## Basic Properties of Operating Systems

- High reliability
  - Secure
  - Effectiveness
  - General overtime/ Inherit and adaption
  - Convenience



## Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- General overtime/ Inherit and adaption
- Convenience

## Effectiveness

- Resources are exploited thoroughly;
- Resource that is limited still able to handle complex requirement.
- The system need to maintain the synchronization;
  - Slow devices do not affect the whole system operation

## Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- General overtime/ Inherit and adaption
- Convenience

## Security

- Data and programs must be protected
  - No unwanted modification happen in every working mode
    - protected from illegal access
  - Different resources have different protection requirements
  - Many levels protections with various of tools
  - Important for multi tasking system

## Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- General overtime/ Inherit and adaption
- Convenience

### Convenience

- Easy to use
- Various effective levels
- Have many assisting system

## Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- Generalizable overtime/ Inherit and adapt
- Convenience

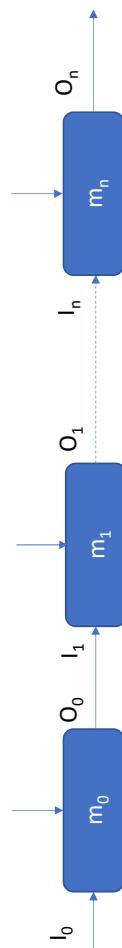
### Generalizable overtime

- System must be Inheritable
  - Operations, notification can not change
  - If changed: notify and with detailed guide (chkdisk/scandisk)
  - Help keeping and increasing users
  - System must have ability to adapt to changes that may happen
- Example: Y2K problem; FAT 12/16/32

# Chapter 1 Operating System Overview

Chapter 1 Operating System Overview  
7. Principles of Operating Systems

## ● Modular principle



## The principle of relativity in positioning

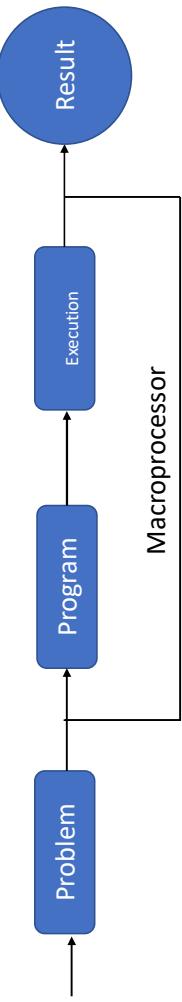
- General form of machine instruction:  
**Instruction\_code operand\_1 operand\_2**  
Ex: Mov AX, B  
Operand\_1, operand\_2 can not be absolute address

- Modular
- Relativity in positioning
- Macroprocessor
- Initialization in the installation
- Functional repetition
- Standard values
- Multi-levels protection

## Principles of Operating Systems

## Principle of functional repetition

- (1 job can be performed in combinations of different modules
- Make system safer
- Convenient for users' request
- Ex: multiple translation programs for the same language.



## The macroprocessor principle

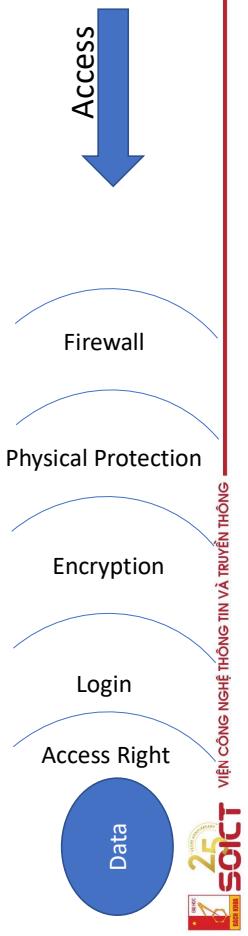
- (Also called generate principle)
- OS is made up of modules in the available module repository to satisfy
  - Machine's configuration
  - User's requirement
- Distinguish **set up** and **install**

## Standard values principle

- Increase module flexibility -> module parameterization
- For convenience and safety -> build standard values for modules

## Multi-levels protection

- Resources have different characteristics -> different methods of protection
  - Multiple layers of protection -> safety
  - Help significantly reduce unintentional errors



# Chapter 1 Operating System Overview

## Summary

- ① Definition of Operating system
  - Layering structure of OS
  - OS's functions
- ② History of Operating system
  - History of computers
  - History of Operating system
- ③ Definition and classification of OS
  - Definitions
  - Classification
- ④ Basic properties of OS
  - High reliability
  - Security
  - Effective
  - Generalize overtime
  - Convenience
- ⑤ Concepts of Operating system
  - Process and Thread
  - System's resources
  - Shell
  - System calls
- ⑥ Operating system's structure
  - OS's components
  - OS's services
  - System calls
  - System's structures
- ⑦ Principles of Operating System

**Operating System**  
(*Principles of Operating Systems*)

Đỗ Quốc Huy

huydq@soict.hust.edu.vn

Department of Computer Science  
School of Information and Communication Technology  
HUST

### ① OS's roles:

- Guarantee process and thread activities
- Create/delete process (user, system)
- Process scheduling
- Provide synchronization mechanism, communication and prevent deadlock among processes



- ① Process
- ② Thread
- ③ CPU scheduling
- ④ Critical resource and process synchronization
- ⑤ Deadlock and solutions



### Process (review)

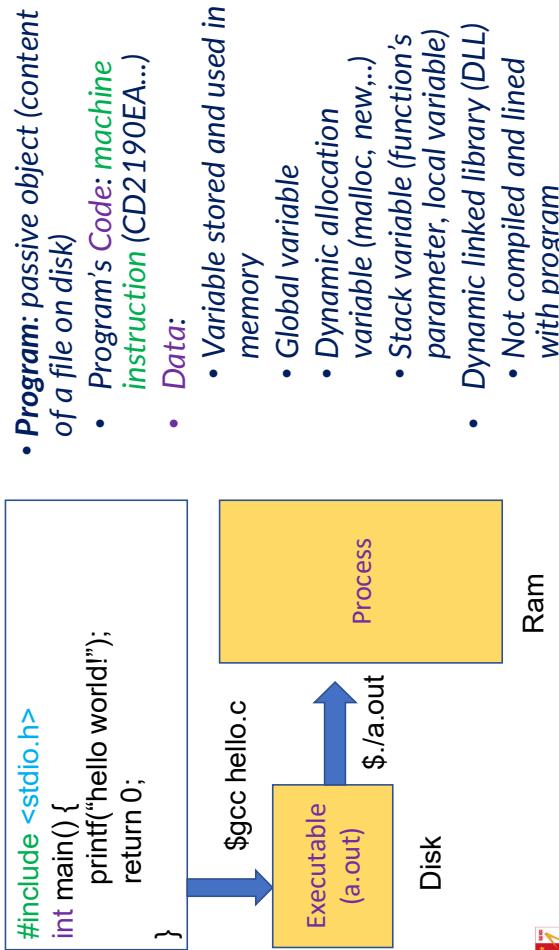
- A **running program**
- Provided resources (CPU, memory, I/O devices...) to complete its task
- Resources are provided at:
  - **process creating time**
    - While running
  - system includes many processes running at the same time
    - OS's : Perform system instruction code
    - User's: Perform user's code
  - May contain  $\geq 1$  **threads**



### ① Process's definition



## Process >< program



## Process >< program



Ram

- Concept of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

- Variable stored and used in memory
- Global variable
- Dynamic allocation variable (malloc, new,...)
- Stack variable (function's parameter, local variable)
- Dynamic linked library (DLL)
- Not compiled and linked with program



- System's state
  - Processor: Registers' values
  - Memory: Content of memory block
  - Peripheral devices: Devices' status
- Program's executing ⇒ system's state changing
  - Change discretely, follow each executed instruction



- Process: a sequence of system's state changing
  - Begin with start state
  - Change from state to state is done according to requirement in user's program

Process is the execution of a program

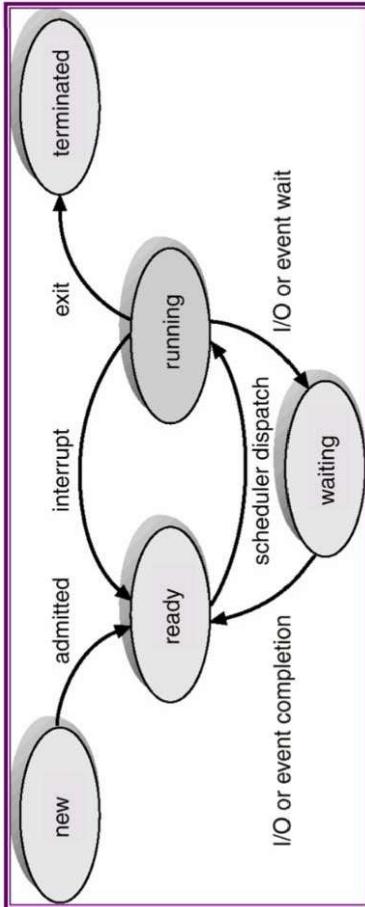
## Compile and run a program

- The OS creates a process and allocate a memory area for it
- System program loader/exec
  - Read and interprets the executive file (file's header)
  - Set up address space for process to store code and data from executive file
  - Put command's parameters, environment variable (argc, argv, envp) into stack
  - Set proper values for processor's register and call "start()" (OS's function)
- Program begins to run at "start()". This function call to main() (program's functions)
  - ⇒ "Process" is running, not mention "program" anymore
  - When main() function end, OS call to "exit()" to cancel the process and take back resources



## Process's state

Process's state is part of process's current activity



## Process's states changing flowchart [Silberschatz]

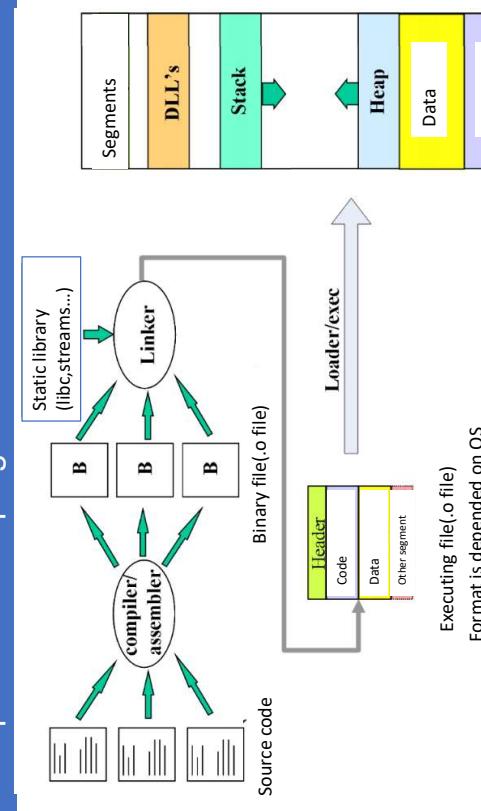
- System that has only 1 processor
  - Only 1 process in **running** state
  - Many processes in ready or **waiting** state

## Process >< program

- Process:** active object (instruction pointer, set of resources)
- A program can be
  - Part of process's **state**
    - One program, many process (different data set)
      - Example: `gcc hello.c || gcc baitap.c`
    - Call to many process



## Compile and run a program

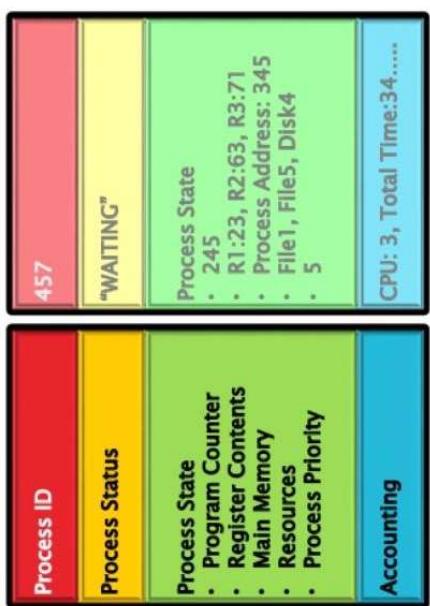


## (Cont.)

## Chapter 2 Process Management 1. Process

## Chapter 2 Process Management 1. Process

### Process Control Block (PCB)



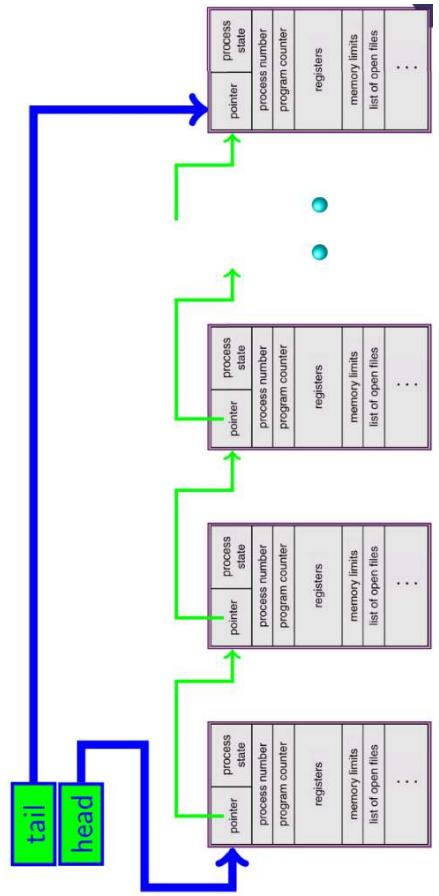
### Process's state



## Chapter 2 Process Management 1. Process

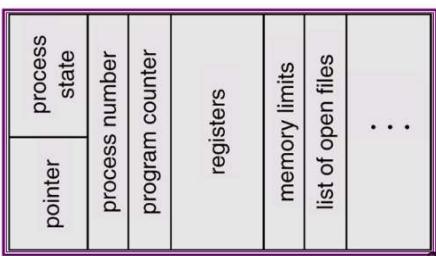
## Chapter 2 Process Management 1. Process

### Process List



### Process Control Block (PCB)

- Each process is presented in the system by a **process control block**
- PCB: an **information structure** allows **identify only 1 process**



- Process state
  - Process counter
  - CPU's registers
  - Information for process scheduling
  - Information for memory management
- Process number
  - Information of usable resources
  - Statistic information
  - Pointer to another PCB
  - ...**

## Chapter 2 Process Management 1. Process

- Concept of process
- Process Scheduling
  - Operations on process
  - Process cooperation
  - Inter-process communication

## Introduction

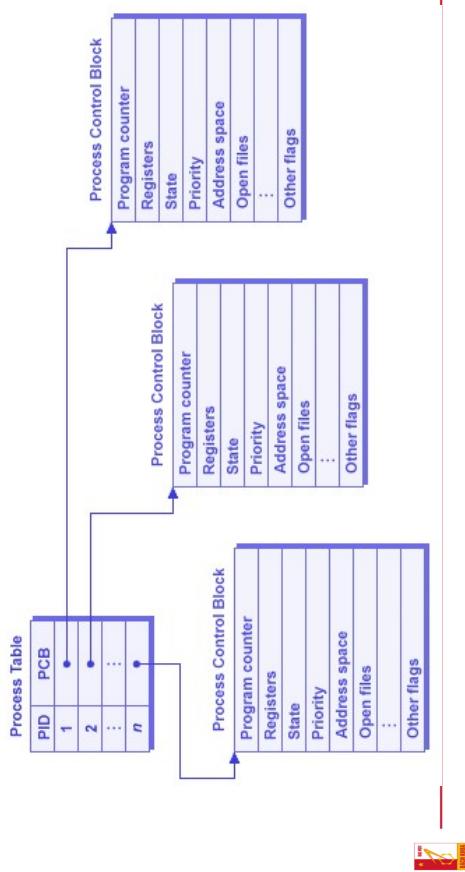
- Objective** Maximize CPU's usage time  
⇒ Need many processes in the system
- Problem** CPU switching between processes  
⇒ Need a queue for processes

**Single processor system**  
⇒ 1 process running  
⇒ Other processes must wait until processor is free

## Single-thread process and Multi-thread process

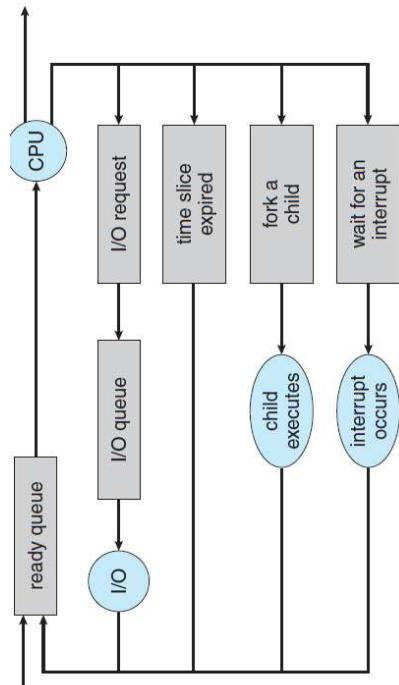
- Single-thread process** : A running process with **only 1** executed thread  
Have **1 thread of executive instruction**  
⇒ Allow executing **only 1 task** at a moment
- Multi-thread process** : Process with **more than 1** executed thread  
⇒ Allow more than 1 task to be done at a moment

## Process Table (Linux)



## Process queue II

- \* Process moves between different queues



\* Newly created process is putted in **ready queue** and wait until it's selected to execute

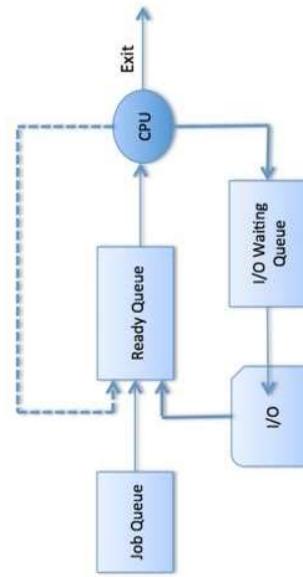
## Process queue III

- Process **selected** and **running**
- ① process issue an **I / O request**, and then be placed in an **I / O queue**
- ② process **create a new sub-process** and **wait** for its termination
- ③ process **removed forcibly** from the CPU, as a result of an **interrupt**, and be put back in the ready queue
- In case (1&2) after the waiting event is finished,
  - Process eventually **switches** from **waiting** → **ready state**
  - Process is then **put back to ready queue**
  - Process **continues** this **cycle** (**ready, running, waiting**) **until it terminates**.
- It's **removed** from all queues
  - its PCB and resources deallocated

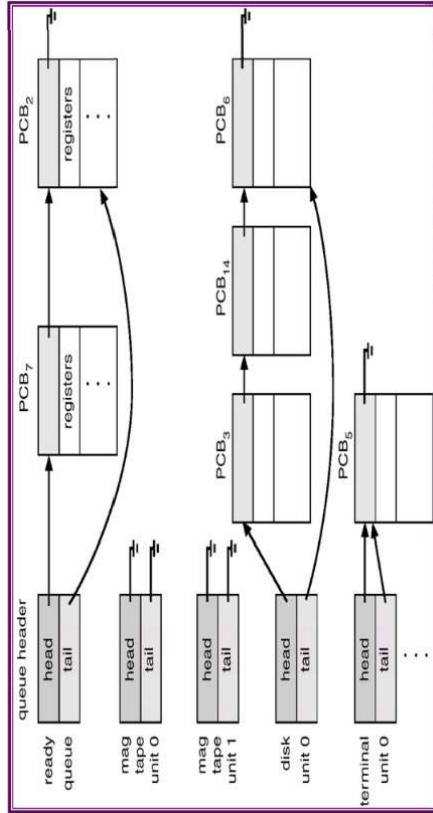
## Process queue I

The system have **many queues** for processes

- Job-queue Set of processes in the system
  - Ready-Queue Set of processes exist in the memory, ready to run
  - Device queues Set of processes waiting for an I/O device.
- Queues for each device are different



## Process queue I



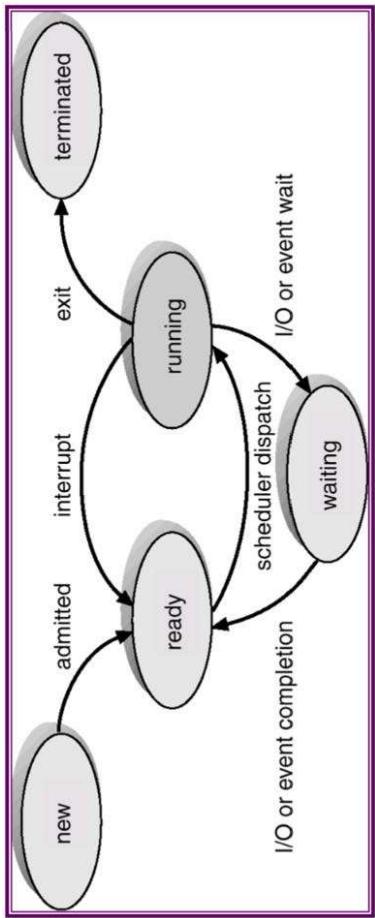
## Job Scheduler

- Select processes from program queue stored in disk and put into memory to execute
- Not frequently (seconds/minutes)
- Control the degree of the multi-programming (number of process in memory)

• When the degree of multi-programming is stable, the scheduler may need to be invoked when a process leaves the system



## Scheduler



- \* Process selection is carried out by appropriate scheduler
  - \* Job scheduler; Long-term scheduler
  - \* CPU scheduler; Short-term scheduler



Chapter 2 Process Management  
1. Process

## Job Scheduler

- Job selection's problem
- Consider the following program

```
PROGRAM PrintValue:
BEGIN
```

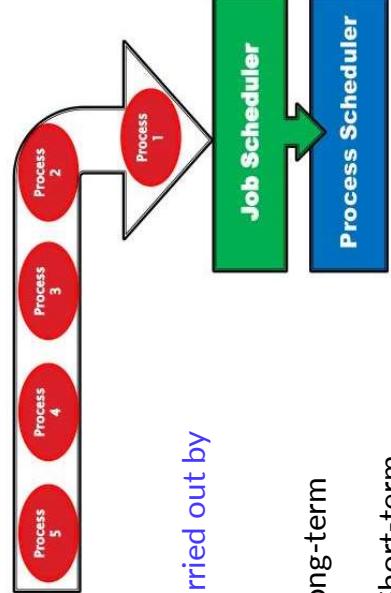
Input A;	IO
Input B;	CPU
C = A + B;	
D = A - B;	
Print "The sum of inputs is: ", C;	IO
Print "The Difference of inputs is: ", D;	
END.	

1



Chapter 2 Process Management  
1. Process

## Scheduler

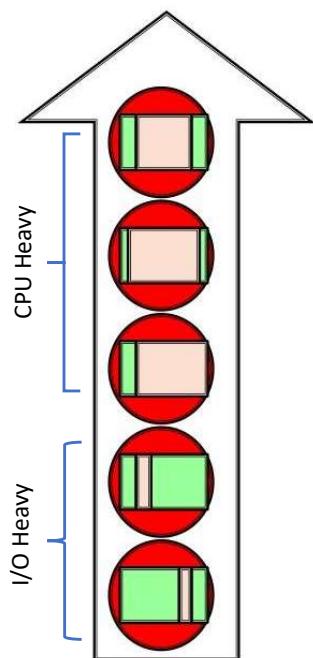


- \* process selection is carried out by appropriate scheduler
- \* Job scheduler; Long-term scheduler
- \* CPU scheduler; Short-term scheduler



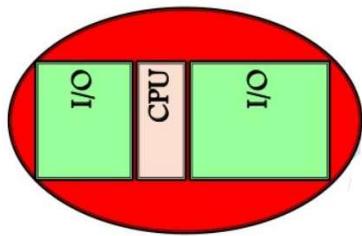
Job Scheduler

- Job selection's problem
    - Should select good process mix of both
      - ⇒ I/O bound : ready queue is empty, waste CPU
      - ⇒ CPU bound: device queue is empty, device is wasted



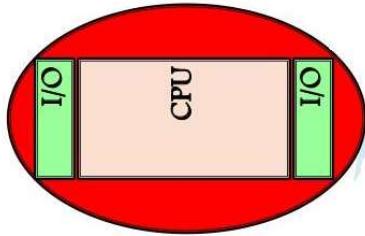
## Job Scheduler

- **I/O bound process:** utilizes less CPU time
    - Ex: Graphic programs



Job Scheduler

- **I/O bound process:** utilizes less CPU time
    - Ex: Graphic programs

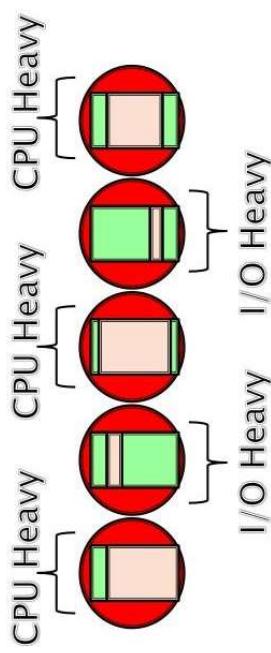


-Joh Schedler

- CPU bound process: uses more CPU time
    - Ex: math solving programs...

Job Scheduler

- Job scheduler will arrange the Jobs then send them to Process Scheduler



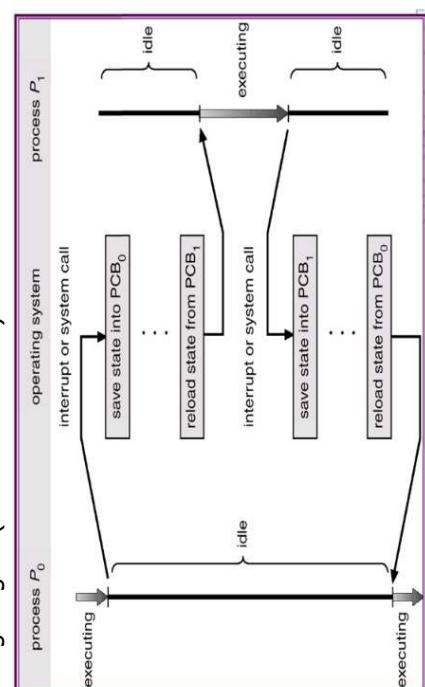
## Context switch

- **Switch CPU from 1 process to another process**
  - **Save the state** of the **old** process and **load the saved state** for the **new** process
  - includes the value of the CPU's **registers**, the process state and **memory-management information**
  - **Take time** and **CPU cannot do anything while switching**
  - Depend on the **support of the hardware**
  - More **complex** the OS, the **more works** must be done during a **context switch**



## Context switch

- Occurs when an **interrupt signal appear** (timely interrupt) or when process issue a system call (to perform I/O)
- CPU switching diagram(Silberschatz 2002)

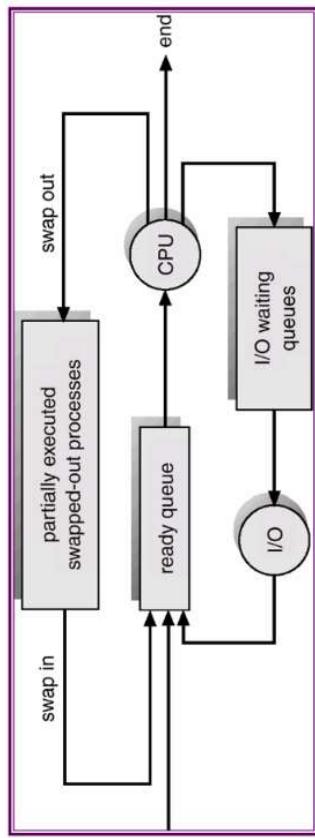


## CPU Scheduler

- **Selects** from processes that are **ready** to execute, and
  - allocates the CPU to 1 of them
  - **Frequently** work (example: at least 100ms/once)
    - A process may execute for only a few milliseconds before waiting for an I/O request
    - Select new process that is ready
    - the short-term **scheduler** must be **fast**
      - 10ms to decide  $\Rightarrow 10/(110)=9\%$  CPU time is wasted
    - Process selection problem (CPU scheduler)



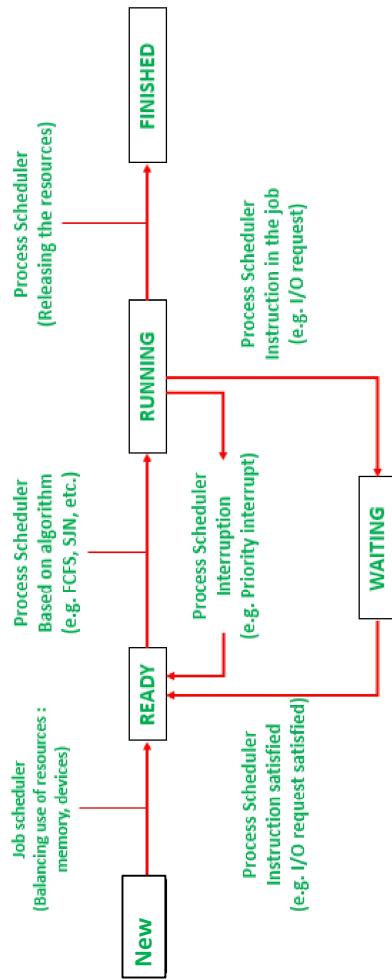
## Process swapping (Medium-term scheduler)



- Task
  - **Removes** processes from memory, reduces the degree of multiprogramming
  - Processes can be **brought back** into memory and its execution can be continued **where it left off**

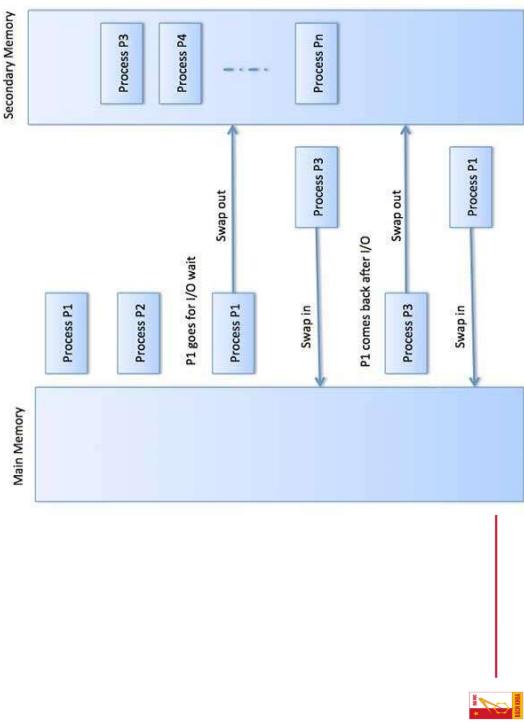
- Objective: Free memory, make a wider unused memory area

## Schedulers and process's states

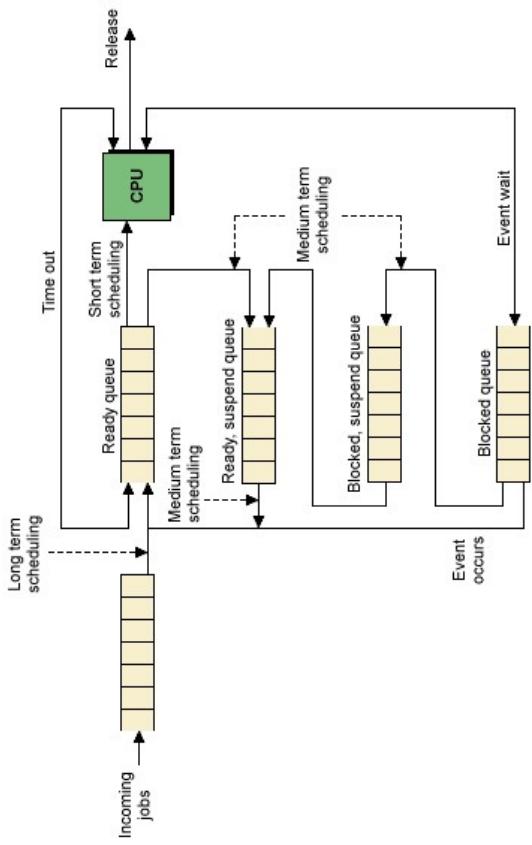


- Concept of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

## Process swapping (Medium-term scheduler)

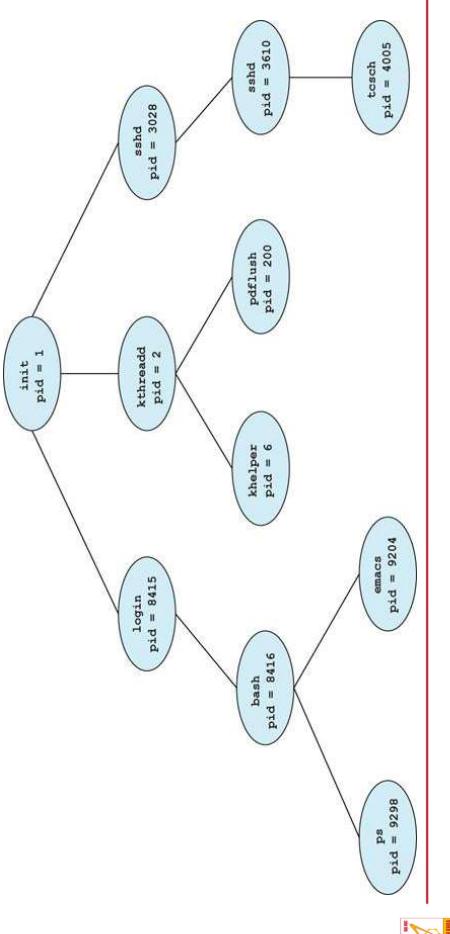


## Medium-term scheduler



## Process Creation

- Example: A process tree in Solaris OS



## Operations on process

- Process Creation

## Process Termination



## Example Process Creation in Linux

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
  
```

## Process Creation

- Process may create several new processes (CreateProcess(), fork())
  - The creating process: **parent-process**
  - The new process: **child-process**
  - Child-process may create new process ⇒ Tree of process



## Some functions with process in WIN32 API

- CreateProcess( ... )
  - LPCTSTR Name of the execution program
  - LPTSTR Command line parameter
  - LPSECURITY\_ATTRIBUTES A pointer to process **security attribute** structure
  - LPSECURITY\_ATTRIBUTES A pointer to thread **security attribute** structure
  - BOOL allow process inherit devices (Example: CREATE\_NEW\_CONSOLE)
  - DWORD process creation flag (Example: CREATE\_NEW\_CONSOLE)
  - LPVOID Pointer to environment block
  - LPCTSTR full path to program
  - LPSTARTUPINFO info structure for new process
  - LPPROCESS\_INFORMATION Information of new process
  - TerminateProcess(HANDLE hProcess, UINT uExitCode)
    - **hProcess** A handle to the process to be terminated
    - **uExitCode** process termination code
  - WaitForSingleObject(HANDLE hHandle, DWORD dwMs)
    - **hHandle** Object handle
    - **dwMs** Waiting time (INFINITE)



## Process Creation

- Resource allocation
  - Child receives resource from
    - OS
    - parent-process
    - All of the resource
  - a **subset** of the resources of the parent process (to prevent a process from creating too many children)
- When a process creates a new process, two possibilities exist in terms of **execution**:
  - The parent **continues to execute concurrently** with its children
  - The parent **waits until** some or all of its children have **terminated**



## Chapter 2 Process Management 1. Process

## Some functions with process in WIN32 API(Example)

```
#include <stdio.h>
#include <windows.h>
int main(VOID){
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    if (!CreateProcess(NULL, "/C:\\WINDOWS\\system32\\mspaint.exe", NULL,
        NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

## Process Termination

- Finishes executing its final statement and asks the OS to delete (exit)
  - **Return data** to parent process
  - Allocated **resources** are returned to the **system**
- Parent process may terminate the execution of child process
  - Parent must know **child process's id** ⇒ child process has to send its id to parent process when **created**
  - Use the **system call** (abort)
  - Parent process terminate child process when
    - The child has **exceeded its usage** of some of the **resources**
    - The task assigned to the child is **no longer required**
    - The **parent is exiting**, and the OS does not allow a child to continue
- ⇒ Cascading termination. Example: *turn off computer*

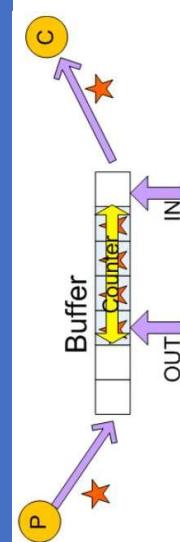


## Processes' relation classification

- Cooperation in order to
  - Share Information
  - Speedup Computation
  - Provide Modularity
  - Increase the convenience
- Process collaborate requires mechanism that allow process to
  - Communicate with each other
  - Synchronize their actions



## Producer - Consumer Problem



- The system includes 2 processes

- Producer creates product
- Consumer consumes created product

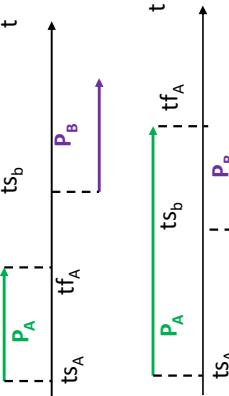
### Application

- Printing program (producer) creates characters that are consumed by printer driver (consumer)
- Compiler (producer) creates assembly code, Assembler (consumer/producer) consumes assembly code and generate object module which is consumed by the exec/loader (consumer)

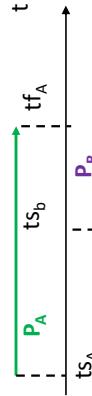


## Processes' relation classification

### Sequential processes



- Parallel processes



- Independence: Not affect or affected by other running process in the system
- Cooperation: affect or affected by other running process in the system



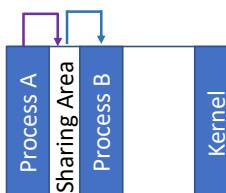
- Concept of process
- Process Scheduling
- Operations on process
- Process cooperation

### ● Inter-process communication



### Communicate between process

- Memory sharing model
  - Process share a **common memory area**
  - Implementation codes are written explicitly by application **programmer**
  - Example: Producer-Consumer problem



### Producer - Consumer Problem

```
while(1){
    /*produce an item in nextProduced*/
    while (Counter == BUFFER_SIZE); /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}
```

```
while(1){
    while(Counter == 0); /*do nothing*/
    nextConsumed = Buffer[OUT];
    OUT = (OUT + 1) % BUFFER_SIZE;
    Counter--; /*consume the item in nextConsumed*/
}
```

```
Producer
Consumer
```

```
while(Counter == 0); /*do nothing*/
nextConsumed = Buffer[OUT];
OUT = (OUT + 1) % BUFFER_SIZE;
Counter--; /*consume the item in nextConsumed*/
}
```

### Producer - Consumer Problem

- Producer and Consumer work simultaneously
- Use a sharing **Buffer** which store product filled in by Producer and taken out by Consumer
  - IN Next empty place in buffer
  - OUT First filled place in the buffer.
  - Counter Number of products in the buffer
- **Producer** and **Consumer** must be synchronized
- Consumer does not try to consume a product that was not created
- Unlimited size buffer
  - Buffer is empty -> Consumer need to wait
  - Producer can put product into buffer without waiting
- Limited size buffer
  - Buffer is empty -> Consumer need to wait
  - Producer need to wait if the Buffer is full

### Producer - Consumer Problem

### Producer - Consumer Problem

## Chapter 2 Process Management 1. Process

## Chapter 2 Process Management 1. Process

### Ex: File mapping- Process 2

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#pragma comment(lib, "user32.lib")
#define BUF_SIZE 256
TCHAR szName[] = TEXT("Global\\MyFileMappingObject");

int _tmain()
{
    HANDLE hMapFile;
    LPCTSTR pBuf;

    hMapFile = OpenFileMapping(
        FILE_MAP_ALL_ACCESS, // do not inherit the name
        FALSE, // name of mapping object
        szName);
    if (hMapFile == NULL)
    {
        fprintf(TEXT("Could not open file mapping object\n"),
                GetLastError());
        return 1;
    }

    pBuf = (LPVOID) MapViewOfFile(hMapFile,
        FILE_MAP_ALL_ACCESS,
        0,
        0,
        BUF_SIZE);
    if (pBuf == NULL)
    {
        fprintf(TEXT("Could not map view of file (%d)\n",
                    GetLastError()));
        CloseHandle(hMapFile);
        return 1;
    }

    MessageBox(NULL, pBuf,
        TEXT("Process 2"), MB_OK);
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
    return 0;
}
```

D:\HO\BICH Khoa\HANH  
HOI\Windows\Process\

### Ex: File mapping- Process 1

```
Chapter 2 Process Management  
1. Process
```

**Example file mapping in Windows**

Memory sharing model

```
pBuf = (LPVOID) MapViewOfFile(hMapFile
    // handle to map object
    FILE_MAP_ALL_ACCESS,
    // read/write permission
    0,
    0,
    BUF_SIZE);

if (pBuf == NULL)
{
    fprintf(TEXT("Could not map view of file (%d)\n",
                GetLastError()));
    CloseHandle(hMapFile);
    return 1;
}

MessageBox(NULL, pBuf,
    TEXT("Process 1"), MB_OK);
UnmapViewOfFile(pBuf);
CloseHandle(hMapFile);
return 0;
```

**Chapter 2 Process Management  
1. Process**

**Ex: File mapping- Process 1**

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#define BUF_SIZE 256
TCHAR szName[] = TEXT("Global\\MyFileMappingObject");
int _tmain()
{
    HANDLE hMapFile;
    LPCTSTR pBuf;
    hMapFile = CreateFileMapping(
        INVALID_HANDLE_VALUE,
        NULL, // default security
        PAGE_READWRITE,
        0, // maximum object size (high-order
        // maximum object size (low-order
        // name of mapping object
        szName);
    if (hMapFile == NULL)
    {
        fprintf(TEXT("Could not create file mapping object (%d).\n",
                    GetLastError()));
        CloseHandle(hMapFile);
        return 1;
    }

    pBuf = (LPVOID) MapViewOfFile(hMapFile,
        FILE_MAP_ALL_ACCESS,
        0,
        0,
        BUF_SIZE);
    if (pBuf == NULL)
    {
        fprintf(TEXT("Could not map view of file (%d).\n",
                    GetLastError()));
        CloseHandle(hMapFile);
        return 1;
    }

    MessageBox(NULL, pBuf,
        TEXT("Process 1"), MB_OK);
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
    return 0;
```

**Chapter 2 Process Management  
1. Process**

**Communicate between process**

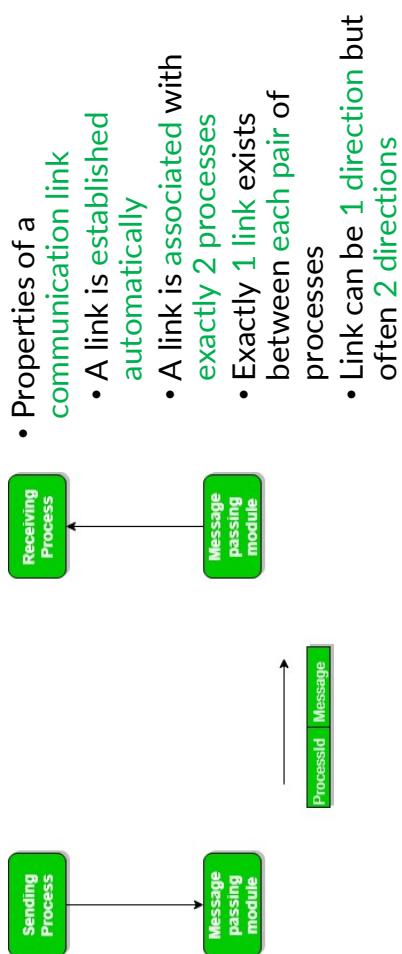
**Inter-process communication model**

- A mechanism allow processes to communicate and synchronize
- Often used in distributed system where processes lie on different computers (chat)
- Guaranteed by message passing system

**Diagram:** process A and process B share a message queue in the kernel. The queue contains messages m<sub>0</sub>, m<sub>1</sub>, m<sub>2</sub>, ..., m<sub>n</sub>.



## Direct Communication



## Message passing system

- Allow processes to communicate without sharing variable
- Require 2 basic operations
  - Send (msg) msg has **fixed or variable size**
  - Receive (msg)
- If 2 processes **P** and **Q** want to **communicate**, they need to
  - Establish a **communication link** (physical/logical) between them
  - Exchange messages via operations: send/receive

## Chapter 2 Process Management 1. Process

## Indirect Communication

- Messages are sent to and received from mailboxes, or **ports**
- Each mailbox has a **unique identification**

- 2 processes can communicate only if they share a mailbox
- Communication link's properties
  - Established **only if both members** of the pair have a **shared mailbox**
  - A link may be associated with **more than 2** processes
    - Each pair** of communicating processes may have **many links**
    - Each link** corresponding to **1 mailbox**
  - A link can be either 1 or 2 directions



## Chapter 2 Process Management 1. Process

## Direct Communication

- Each process that wants to communicate must **explicitly name the recipient** or **sender** of the communication
  - send (**P**, message) – Send a message to process **P**
  - receive(**Q**, message) – Receive a message from process **Q**

```

void consumer(void)
{
    while (TRUE) {
        recv( producer, item );
        ...
        consume item;
        ...
    }
}

```

```

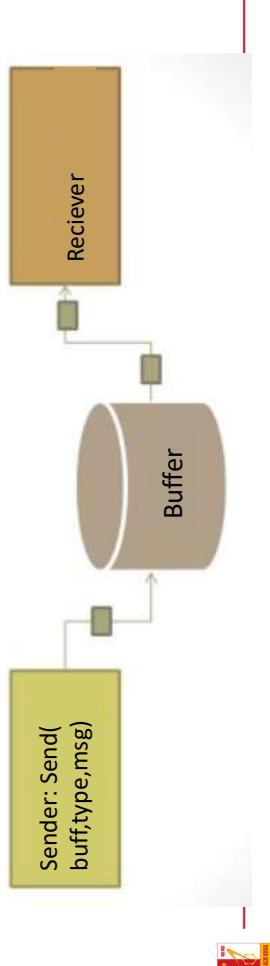
void producer(void)
{
    while (TRUE) {
        ...
        produce item;
        ...
        send( consumer, item );
        ...
    }
}

```



## Message Buffering

- Messages exchanged by communicating processes reside in a temporary **queue**



## Buffering

- A queue can be implemented in 3 ways
  - Zero capacity:
    - maximum length **0** => link cannot have any messages waiting in it
  - **sender must block** until the recipient receives the message
  - Bounded capacity
    - Queue has finite length **n** => store at most **n** messages
    - If the **queue is not full**, message is placed in the queue and the sender can continue execution without waiting
    - If **link is full**, **sender must block** until **space is available** in the queue
  - Unbound capacity
    - The sender never blocks

- Message passing may be either blocking or nonblocking
  - Blocking synchronous communication
  - Non-blocking asynchronous communication
  - send() and receive() can be blocking or non-blocking
  - Blocking send: sending process is blocked until the message is received by the receiving process or by the mailbox
  - Non-blocking send: The sending process sends the message and resumes operation
  - Blocking receive: receiver blocks until a message is available
  - Non-blocking receive: receiver retrieves either a valid message or a null

## Indirect Communication

- Operations:
  - Create a new mailbox
  - Send and receive messages through the mailbox
    - Send(A, msg): send a msg to mailbox A
    - Receive(A, msg): receive a msg from mailbox A
  - Delete a mailbox



## Synchronization

- Message passing may be either blocking or nonblocking
  - Blocking synchronous communication
  - Non-blocking asynchronous communication
  - send() and receive() can be blocking or non-blocking
  - Blocking send: sending process is blocked until the message is received by the receiving process or by the mailbox
  - Non-blocking send: The sending process sends the message and resumes operation
  - Blocking receive: receiver blocks until a message is available
  - Non-blocking receive: receiver retrieves either a valid message or a null

## Winsock API 32 functions

**socket()** Create socket for data communication  
**bind()** Assign an identifier for created socket (assign with a port)  
**listen()** Listen to one connection  
**accept()** Accept connection  
**connect()** Connect to the server.  
**send()** Send data with stream socket.  
**sendto()** Send data with datagram socket.  
**receive()** Receive data with stream socket.  
**recvfrom()** Receive data with datagram socket.  
**closesocket()** End an existed socket.



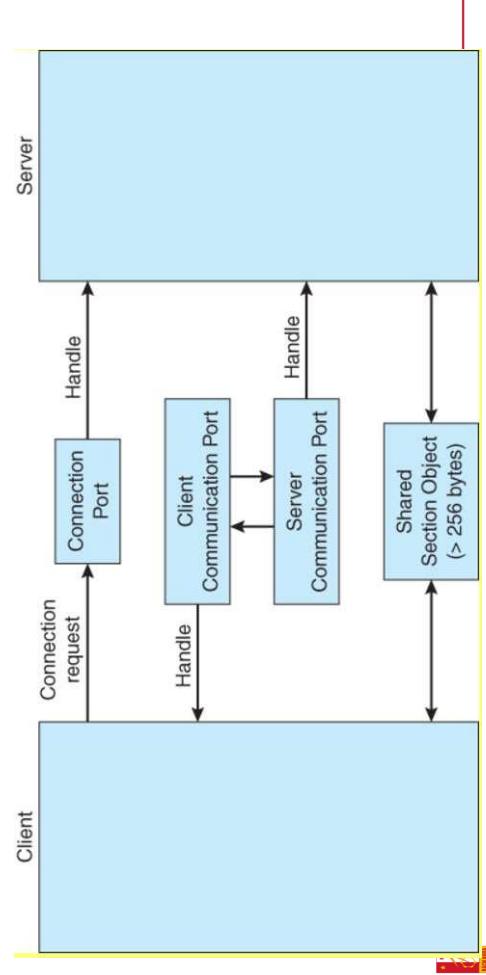
## Homework

Use Winsock to make a Client-Server program

Chat program.  
.....



## Windows XP Message Passing



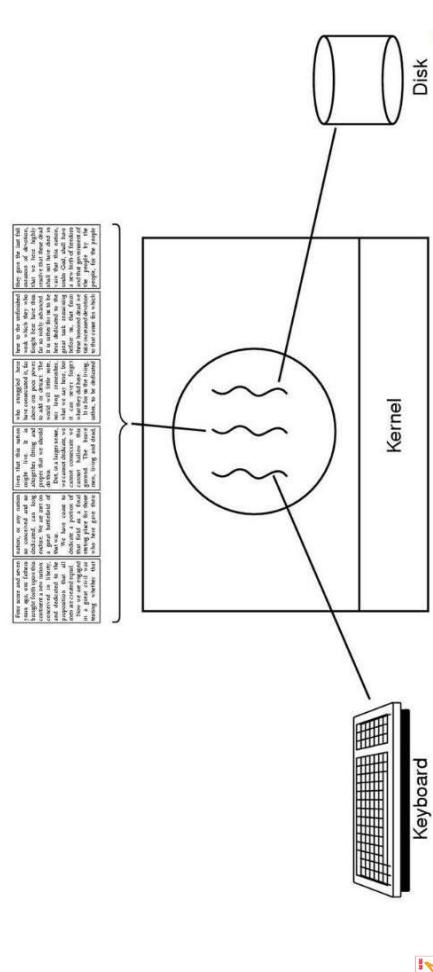
## Communication in Client - Server Systems with Sockets

- Socket is defined as an endpoint for communication,
- Each process has 1 socket
- Made up of an IP address and a port number.
  - E.g.: 161.25.19.8:1625
  - IP Address: Computer address in the network
  - Port: identifies a specific process
- Types of sockets
  - Stream Socket: Based on TCP/IP protocol → Reliable data transfer
  - Datagram Socket: based on UDP/IP protocol → Unreliable data transfer
- Win32 API: Winsock
  - Windows Sockets Application Programming Interface



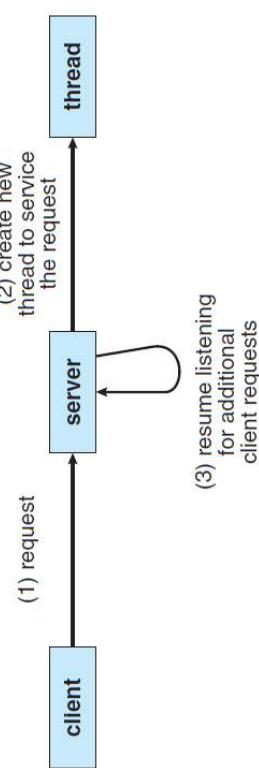


## Example: Word processor (Tanenbaum 2001)

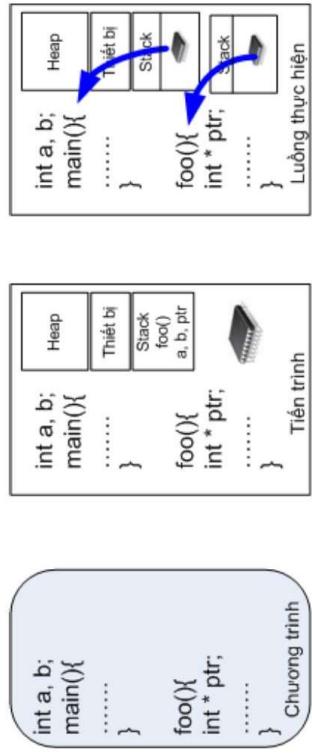


Chapter 2 Process Management  
2. Thread

## Example: Multithreaded server (Silberschatz)



## Program - Process - Thread

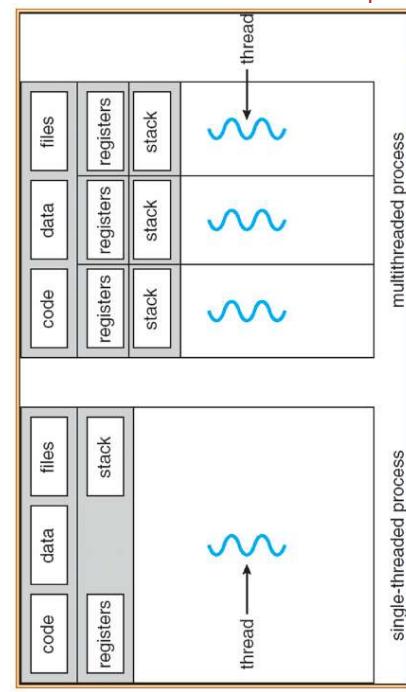


- Program: Sequence of instructions, variables,..
- Process: Running program: Stack, devices, processor,..
- Thread: A running program in process context
  - Multi-processor → Multi threads, each thread runs on 1 processor
  - Different in term of registers' values, stack's content

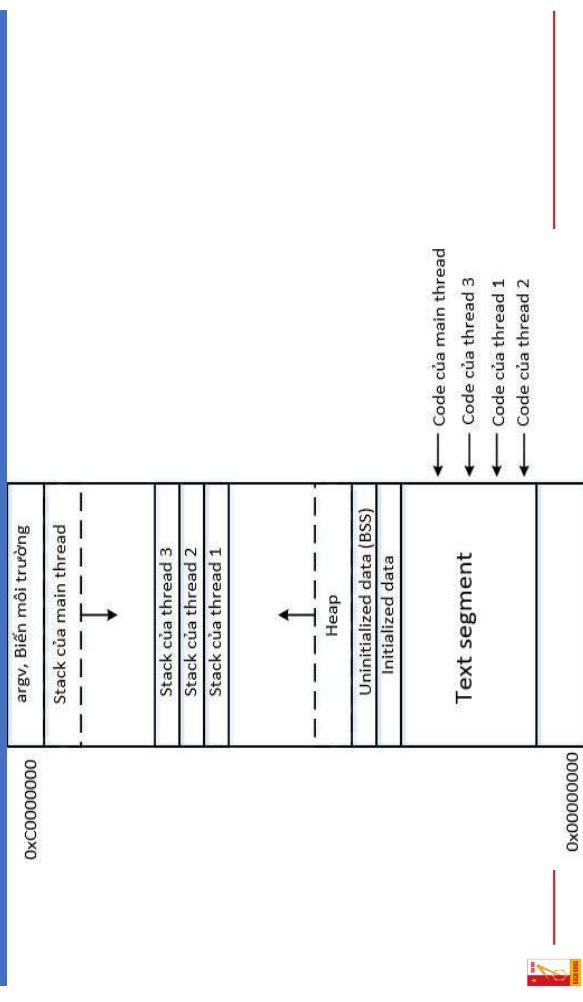
Chapter 2 Process Management  
2. Thread

## Single-threaded and multi-threaded process

- Traditional OS (MS-DOS, UNIX)
  - Process has 1 controlling thread (heavyweight process)
- Modern OS (Windows, Linux)
  - Process may have many threads
  - Perform many tasks at a single time

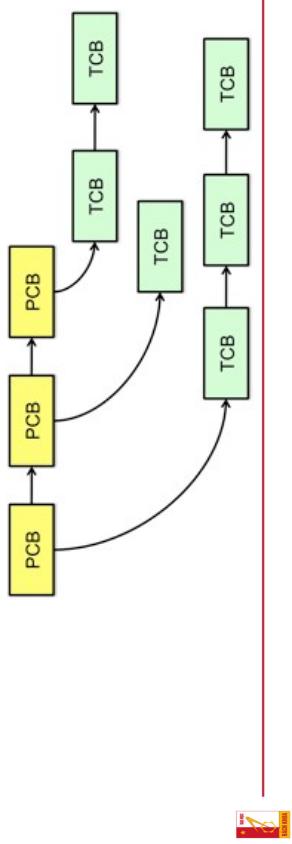


## Process's memory organization with 4 threads (Linux / x86-32)



## Concept of thread

- Basic CPU using unit, consists of
  - Thread ID
  - Program Counter
  - Registers
  - Stack space



## Distinguishing between process and thread

Process	Thread
Has code/data/heap and other segments	No separating code or heap segment
Has at least 1 thread	Not stand alone but must be inside a process
Threads in the same process share code/data/heap, devices but have separated stack and registers	Many threads can exist at the same time in each process. First thread is the main thread and own process's stack
Create and switch process operations are expansive	Create and switch thread operations are inexpensive
Good protection due to own address space	Common address space, need to protect
When process terminated, resources are returned and threads have to terminated	Thread terminated, its stack is returned

- Threads in the same process share
  - Code segment
  - Data segment (global objects)
  - Other OS's resources (opening file...)
- Thread can run same code segment with different context (Register set, program counter, stack)
- LWP: Lightweight Process
- A process has at least one thread



## Benefits of multithreaded programming

- Economy
  - Create, switch, terminate threads is less expensive than process
- Utilization of multiprocessor architectures
  - each thread may be running in parallel on a different processor.



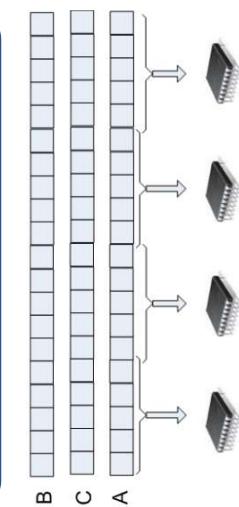
## Benefit of multithreading-> example

### Vector computing

```
for (k = 0;k < n;k++) {  
    a[k] = b[k]* c[k];  
}
```

### Multi-threading model

```
void fn(a,b)  
for(k = a; k < b; k ++){  
    a[k] = b[k] * c[k];  
}  
void main(){  
    Create Thread(fn(0, n/4));  
    Create Thread(fn(n/4, n/2));  
    Create Thread(fn(n/2, 3n/4));  
    Create Thread(fn(3n/4, n));  
}
```



## Benefits of multithreaded programming

- Responsiveness
- Resource sharing
- Economy
- Utilization of multiprocessor architectures



## Benefits of multithreaded programming

- Responsiveness
  - allow a program to continue running even if part of it is blocked or is performing a long operation
- Example: A multi-threaded Web browser
  - 1 thread interactives with user
  - 1 thread downloads data
- Resource sharing
  - threads share the memory and the resources of the process
  - Good for parallel algorithms (share data structures)
  - Threads communicate via sharing memory
  - Allow application to have many threads act in the same address space



## Chapter 2 Process Management 2. Thread

### User -Level Threads

- Advantage
  - Fast to create and manage
- Disadvantage
  - if a thread perform blocking system call , the entire process will be blocked ⇒ Cannot make use of multi-thread.



## Chapter 2 Process Management 2. Thread

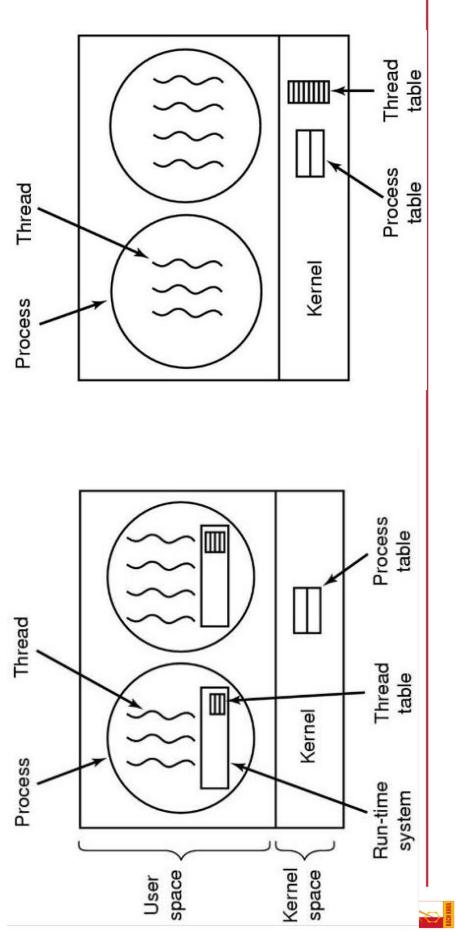
### Kernel - Level threads

- Kernel keeps information of process and threads
- Threads management is performed by kernel
  - No thread management code inside application
  - Thread scheduling is done by kernel
- OSs support kernel thread: Windows NT/2000/XP, Linux, OS/2,..



## Chapter 2 Process Management 2. Thread

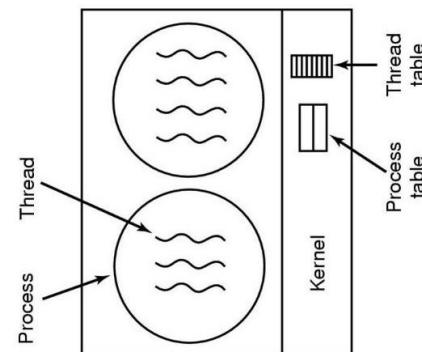
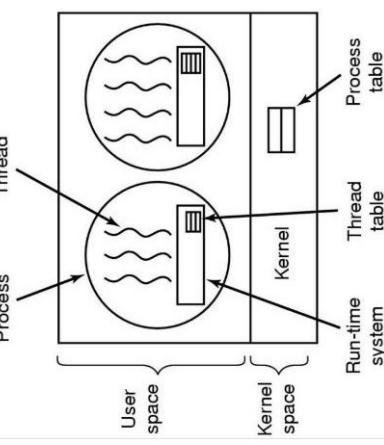
### Thread implementation



## Chapter 2 Process Management 2. Thread

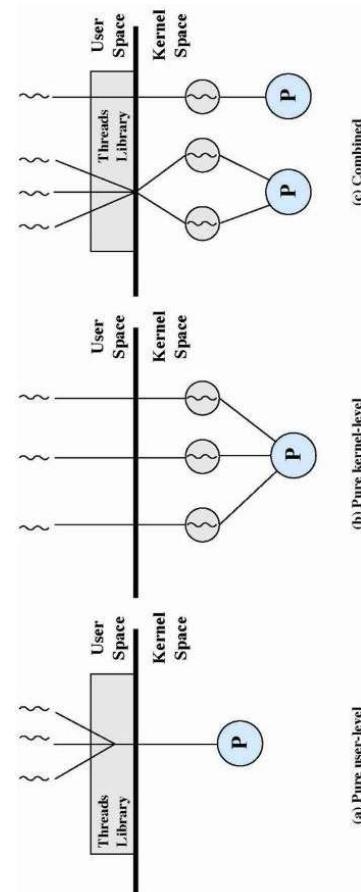
### User -Level Threads

- Thread management is done by application
- Kernel **does not know** about thread existence
- Process scheduled like a single unit
  - Each process is assigned with a single state
    - Ready, waiting, running,..
- User threads are supported above the kernel and are implemented by a **thread library**
- Library support creation, scheduling and management..



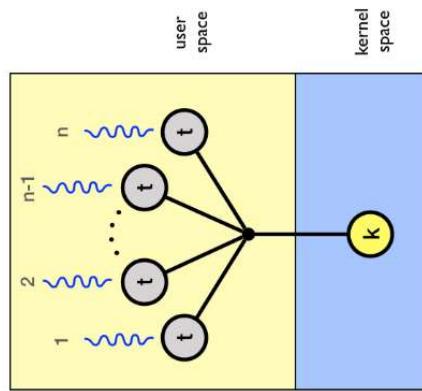
## Introduction

- Many systems provide support for both user and kernel threads -> different multithreading models



## Many-to-One Model

- Maps many user-level threads to 1 kernel thread.
- Thread management is done in user space
  - Efficient
  - the entire process will block if a thread makes a blocking system call
  - multiple threads are unable to run in parallel on multi processors
  - implemented on OSs that do not support kernel threads use the many-to-one model



## Kernel - Level threads

- Advantage:
  - 1 thread perform system call (e.g. I/O request), other threads are not affected
  - In a multiprocessor environment, the kernel can schedule threads on different processors

- Disadvantage:
  - Slow in thread creation and management



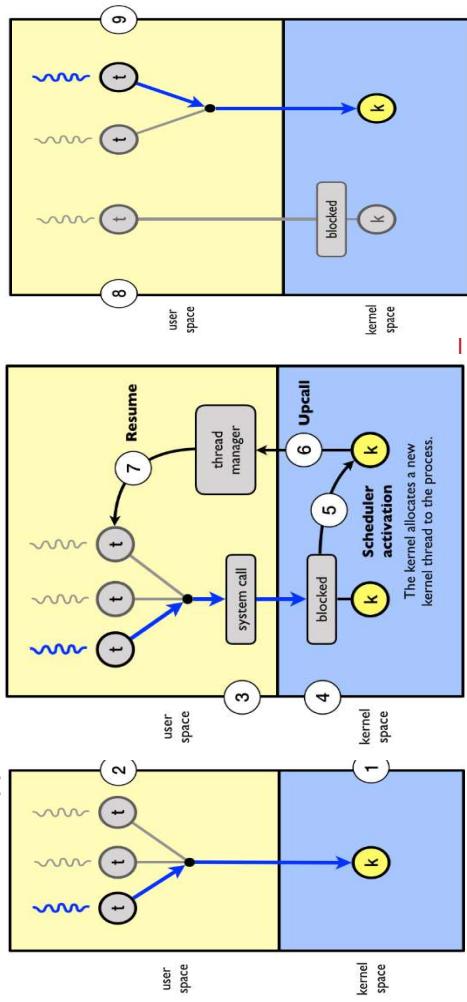
## Introduction

- Multithreading model
  - Thread implementation with Windows
  - Multithreading problem



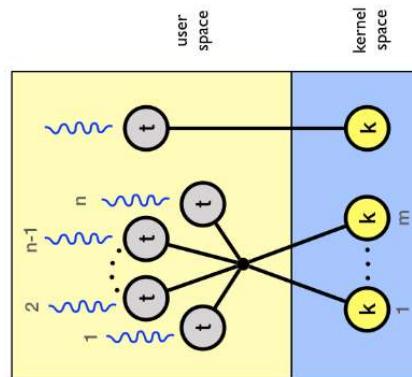
## scheduler activation

- A way for kernel to communicate with user level thread manager to maintain an appropriate number of kernel level thread for process

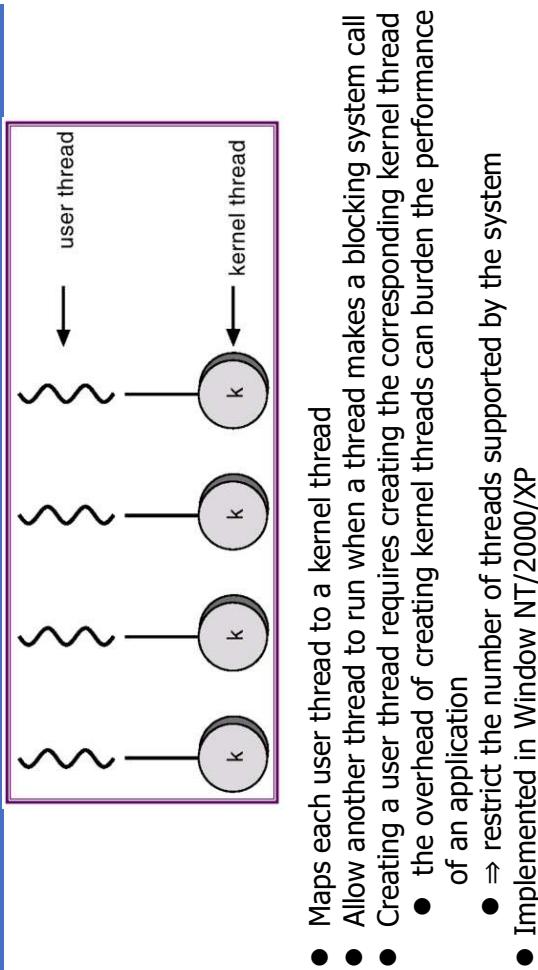


## Two-level model

- A variation of many to many
- Allow favor process with higher priority

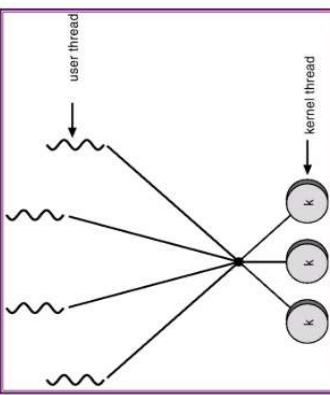


## One-to-one Model



## Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads
- Number of kernel threads: specific to either a particular application or a particular machine
  - E.g.: on multiprocessor → more kernel threads than on uniprocessor
  - Developers can create as many user threads as necessary
  - Kernel threads can run in parallel on a multiprocessor
  - When a thread performs a blocking system call, the kernel can schedule another thread for execution
  - Supported in: UNIX



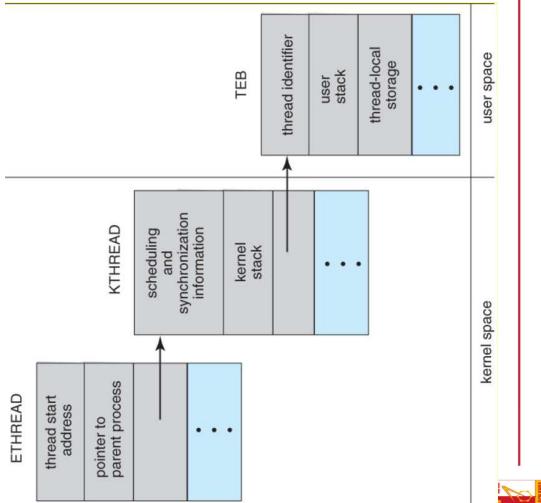
## Example

```

#include <windows.h>
#include <stdio.h>
void Routine(int *n){
    printf("My argument is %d\n", *n);
}
int main(){
    int i, P[5]; DWORD Id;
    HANDLE hHandles[5];
    for (i=0;i < 5;i++) {
        P[i] = i;
        hHandles[i] = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)Routine,&P[i],0,&Id);
        printf("Thread %d was created\n",Id);
    }
    for (i=0;i < 5;i++) WaitForSingleObject(hHandles[i],INFINITE);
    return 0;
}

```

Chapter 2 Process Management  
2. Thread



Chapter 2 Process Management  
2. Thread

## Functions with thread in WIN32 API

- HANDLE CreateThread( . . . );
  - LPSECURITY\_ATTRIBUTES lpThreadAttributes,  
    ⇒ A pointer to a SECURITY\_ATTRIBUTES structure that determines whether  
        the returned handle can be inherited by child processes
  - DWORD dwStackSize,  
    ⇒ The initial size of the stack, in bytes
  - LPTHREAD\_START\_ROUTINE lpStartAddress,  
    ⇒ pointer to the application-defined function to be executed by the thread
  - LPVOID lpParameter,  
    ⇒ A pointer to a variable to be passed to the thread
  - DWORD dwCreationFlags,  
    ⇒ The flags that control the creation of the thread
  - CREATE\_SUSPENDED : thread is created in a suspended state
    - 0: thread runs immediately after creation
  - LPDWORD lpThreadId  
    ⇒ pointer to a variable that receives the thread identifier
  - Return value: handle to the new thread or NULL if the function fails

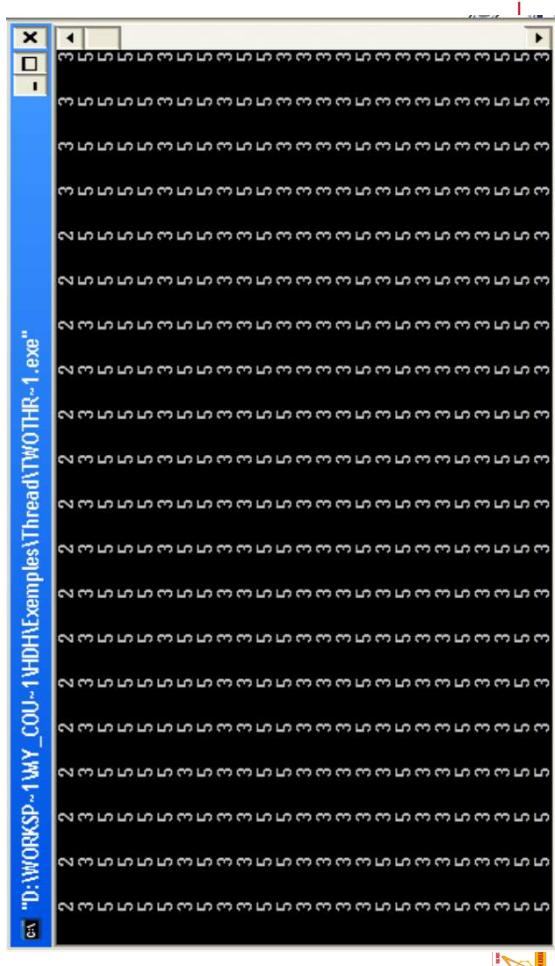
1



## Chapter 2 Process Management 2. Thread

## Chapter 2 Process Management 2. Thread

### Running result



## Chapter 2 Process Management 2. Thread

### Explanation

Shared int y = 1	Thread T1	Thread T2
	$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y * 2$	$y = y * 2$
	$x = ?$	$x = y + 1$

The result of parallel running threads depends on the order of accessing the sharing variable

- Introduction
- Multithreading model
- Thread implementation with Windows

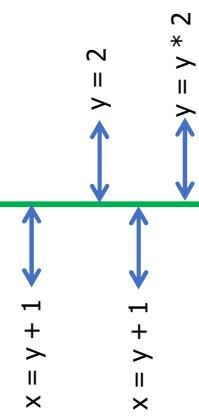
### Multithreading problem

## Chapter 2 Process Management 2. Thread

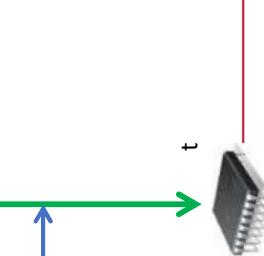
### Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
void T1(){
    while(1){ x = y + 1; printf("%d", x); }
}
void T2(){
    while(1){ y = 2; y = y * 2; }
}
int main(){
    HANDLE h1, h2; DWORD Id;
    h1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T1,NULL,0,&Id);
    h2=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T2,NULL,0,&Id);
    WaitForSingleObject(h1,INFINITE);
    WaitForSingleObject(h2,INFINITE);
    return 0;
}
```

### Thread T1



### Thread T2



}

- System has one *processor* → Only 1 process is running at a time
- Process is executed until it must **wait**, typically for the completion of some **I/O request**
  - Simple system: CPU is not used ⇒ Waste CPU time
  - Multiprogramming system: try to use this time productively, give CPU for another process
- Several **ready processes** are kept **inside memory** at a single time
  - When a process has to **wait**, OS takes the **CPU** away and gives the **CPU** to **another process**
- Scheduling is a **fundamental OS's function**
  - Switch CPU between processes → exploit the system more efficiently



## Introduction

- ① Process
- ② Thread
- ③ CPU scheduling
- ④ Critical resource and process synchronization
- ⑤ Deadlock and solutions



- Basic Concepts
- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling



## Introduction



## CPU Scheduler

- CPU idle → select 1 process from ready queue to run it
  - Process in ready queue
    - FIFO queue, Priority queue, Simple linked list . . .
- CPU scheduling decisions may take place when process switches from
  - 1) running → waiting state (I/O request)
  - 2) running → ready state (out of CPU usage time → time interrupt)
  - 3) waiting → ready state (e.g. completion of I/O)
  - 4) Or process terminates
- Note
  - Case 1&4 ⇒ non-preemptive scheduling scheme
  - Other cases ⇒ preemptive scheduling scheme



## Preemptive and non-preemptive Scheduling

- Non-preemptive scheduling
  - Process keeps the CPU until it releases the CPU either by
    - terminating
    - switching to the waiting state
    - does not require the special hardware (timer)
  - Example: DOS, Win 3.1, Macintosh



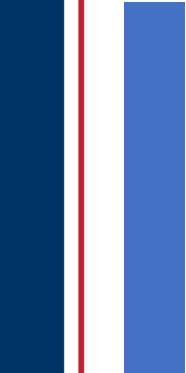
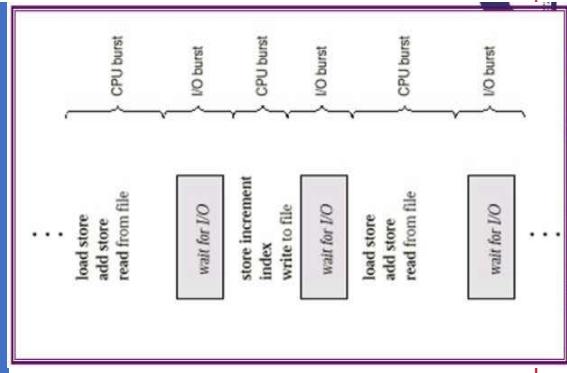
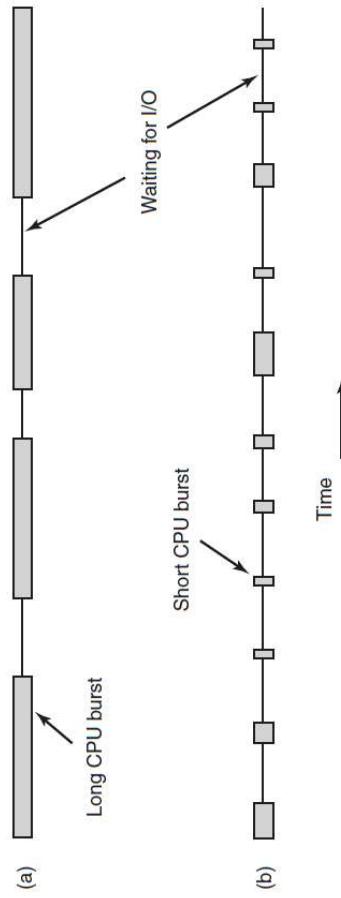
## CPU-I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait
- Begins with a CPU burst
- followed by an I/O burst
- CPU burst → I/O burst → CPU burst → I/O burst → . . .
- End: the last CPU burst will end with a system request to terminate execution
- Process classification
  - Based on distribution of CPU & I/O burst
    - CPU-bound program might have a few very long I/O bursts
    - I/O-bound program would typically have many very short CPU bursts
  - help us select an appropriate CPU-scheduling algorithm



## CPU-I/O Burst Cycle CPU - I/O

- Distinguish the types of processes
  - Based on time allocation for CPU & I/O cycles
    - CPU-bound process has several long CPU cycles
    - I/O-bound process has many short CPU cycles
  - To choose the appropriate scheduling algorithm



## Basic Concepts

### Scheduling Criteria

- Scheduling algorithms
- Multi-processor scheduling



## Scheduling Criteria I

### CPU utilization

- keep the CPU as busy as possible
- should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system)

### Throughput

- number of processes completed per time unit
- Long process: 1 process/hour
- Short processes: 10 processes/second

### Turnaround time

- Interval from the time of submission of a process to the time of completion
- Can be the sum of
  - periods spent waiting to get into memory
  - waiting in the ready queue
  - executing on the CPU
  - doing I/O



## Preemptive and non-preemptive Scheduling

### Preemptive scheduling

- Process only allowed to run in specified period
- End of period, time interrupt appear, dispatcher is invoked to decide to resume process or select another process
- Protect CPU from "CPU hungry" processes
- Sharing data problem
  - Process 1 updating data and the CPU is taken
  - Process 2 executed and read data which is not completely updated
- Example: Multiprogramming OS WinNT, UNIX



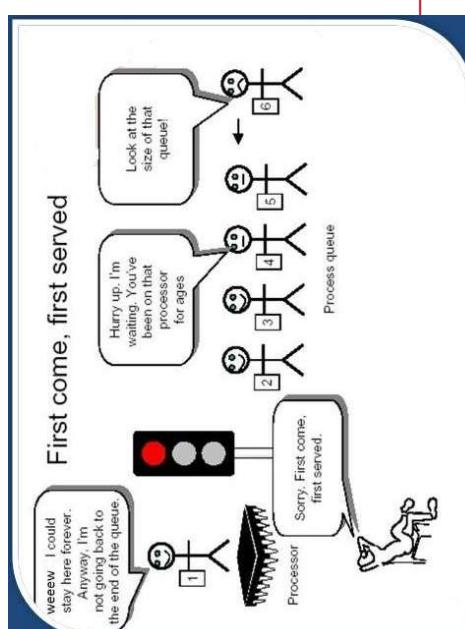
## Preemptive and non-preemptive Scheduling



## FCFS: First Come, First Served

- Rule: process that requests the CPU first is allocated the CPU first

Process owns CPU until it terminates or blocks for I/O request



## FCFS: First Come, First Served

- Rule: process that requests the CPU first is allocated the CPU first
- Process owns CPU until it terminates or blocks for I/O request

- Example

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

	$P_1$	$P_2$	$P_3$
0			

- Pros and cons

- Simple, easy to implement
- Short process must wait like long process
- If  $P_1$  executed last?



- Waiting time
  - sum of the periods spent waiting in the ready queue
  - CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue
- Response time
  - time from the submission of a request until the first response is produced
  - process can produce some output fairly early
  - continue computing new results while previous results are being output to the user

- Assumption: Consider one CPU burst (ms) per process
- Measure: Average waiting time



## Basic Concepts

- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling



## Round Robin Scheduling

### Rule

- Each process is given a time quantum (time slice)  $\tau$  to be executed
- When time's up processor is preemptive, and process is placed in the last position of ready queue
- If there are  $n$  process, longest waiting time is  $(n - 1)\tau$

### Example

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

quantum  $\tau = 4$

### Problem: select $\tau$

- $\tau$  large: FCFS
- $\tau$  small: CPU is switched frequently
- Commonly  $\tau = 10-100ms$



## SJF: Shortest Job First

- Rule: associates with each process the length of the latter's next CPU burst
- process that has the smallest next CPU burst
- Two methods
  - Non-preemptive
  - preemptive (SRPT: Shortest Remaining Time First)

### Example

Process	Burst Time	Arrival Time
$P_1$	8	0.0
$P_2$	4	1.0
$P_3$	9	2.0
$P_4$	5	3.0

### Pros and Cons

- SJF (SRPT) is optimal: Average waiting time is minimum
- It's not possible to predict the length of next CPU burst
- Predict based on previous one

## Priority Scheduling

### Priority Scheduling

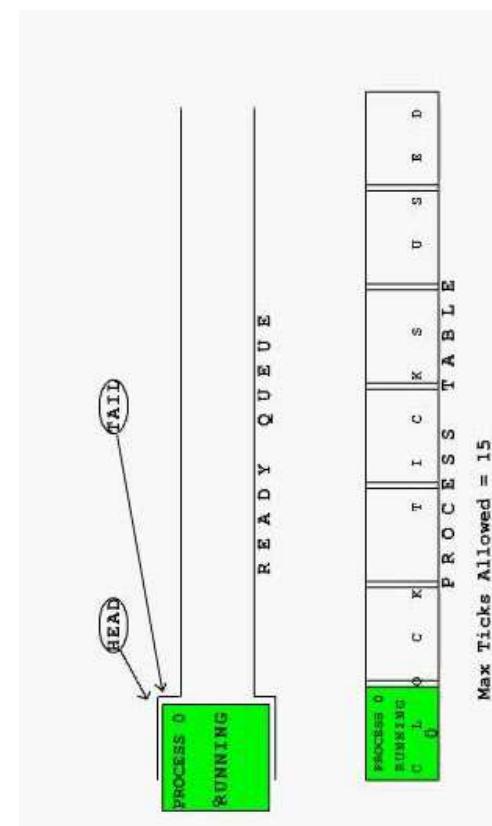
- Each process is attached with a priority value (a number)
- CPU is allocated to the process with the highest priority
- SJF: priority ( $p$ ) is the inverse of the (predicted) next CPU burst
- 2 methods
  - Non-preemptive
  - Preemptive

### Example

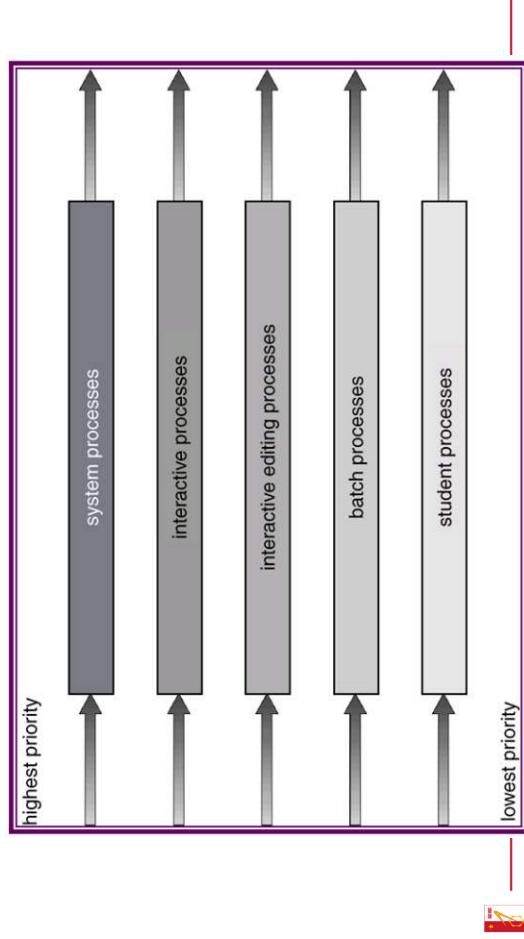
Process	Burst Time	Priority	Arrival Time
$P_1$	10	3	0
$P_2$	1	1	1
$P_3$	2	4	2
$P_4$	1	5	3
$P_5$	5	2	4

"Starvation" problem: low-priority process has to wait indefinitely (even forever)

Solution: increase priority by the time process wait in the system



## Multilevel Queue Scheduling (Example)



## Multilevel Queue Scheduling

- Ready queue is divided into several separate queues
- Processes are permanently assigned to 1 queue
- Based on some properties of the process, such as memory size, priority, or type...
- Each queue has its own scheduling algorithm

## Multilevel Feedback Queue

- Allows a process to move between queues
- separate processes with different CPU-burst characteristics
- If a process uses too much CPU time -> moved to a lower-priority queue
- I/O-bound and interactive processes in the higher-priority queues
- process that waited too long in a lower-priority queue may be moved to a higher-priority queue
- Prevent “starvation”

## Multilevel Queue Scheduling

- among the queues, the scheduler must consider
- Commonly implemented as fixed-priority preemptive scheduling
- Processes in lower priority queue only executed if higher priority queues are empty
- High priority process preemptive CPU from lower priority process
- Starvation is possible
- Time slice between the queues

- foreground process queue, 80% CPU time for RR
- background process queue, 20% CPU time for FCFS

- Basic Concepts
- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling

## Multilevel Feedback Queue

- Queue's **scheduler** is defined by the following parameters:
  - number of queues
  - scheduling algorithm for each queue
  - method used to determine when to upgrade/demote a process to a higher/lower priority queue
  - method used to determine which queue a process will enter when that process needs service



## Problem

- the scheduling problem is correspondingly more complex

### ● Load sharing

#### ● **separate queue** for each processor

- one processor could be **idle**, with an empty queue, while **another** processor was very **busy**

#### ● use a **common ready queue**

#### ● problems :

- two processors choose the **same process** or
  - processes are lost from the queue

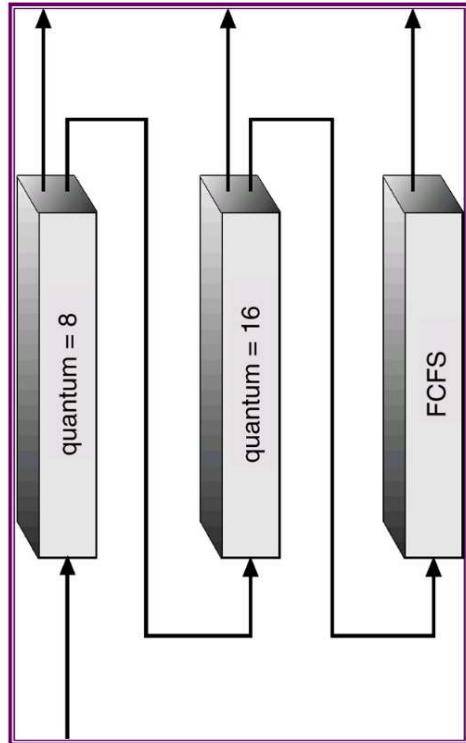
#### ● asymmetric multiprocessing

- **only one processor accesses the system's queue** → no sharing problem

- **I/O-bound processes** may bottleneck on the CPU that is performing all of the operations

## Multilevel Feedback Queue

- Example



- Critical resource
- Internal lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization example
- Monitor



- ① Process
- ② Thread
- ③ CPU scheduling
- ④ Critical resource and process synchronization
- ⑤ Deadlock and solutions



- Write program to simulate multilevel feedback queue



### Example

```

#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
void T1(){
    while(1){ x = y + 1; printf("%d", x); }
}
void T2(){
    while(1){ y = 2; y = y * 2; }
}
int main(){
    HANDLE h1, h2; DWORD Id;
    h1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T1,NULL,0,&Id);
    h2=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T2,NULL,0,&Id);
    WaitForSingleObject(h1,INFINITE);
    WaitForSingleObject(h2,INFINITE);
    return 0;
}

```



## Results



### Producer-Consumer Problem II

counter++

Load R1, counter  
Inc R1  
Store counter,R1

counter--

Load R2, counter  
Dec R2  
Store counter,R2

counter--

Load R1, counter  
Inc R1  
Store counter,R1

counter++

R1 = ?  
counter = 5  
R2 = ?  
counter = 5  
t

### Producer-Consumer Problem II

counter++

Load R1, counter  
Inc R1  
Store counter,R1

counter--

Load R2, counter  
Dec R2  
Store counter,R2

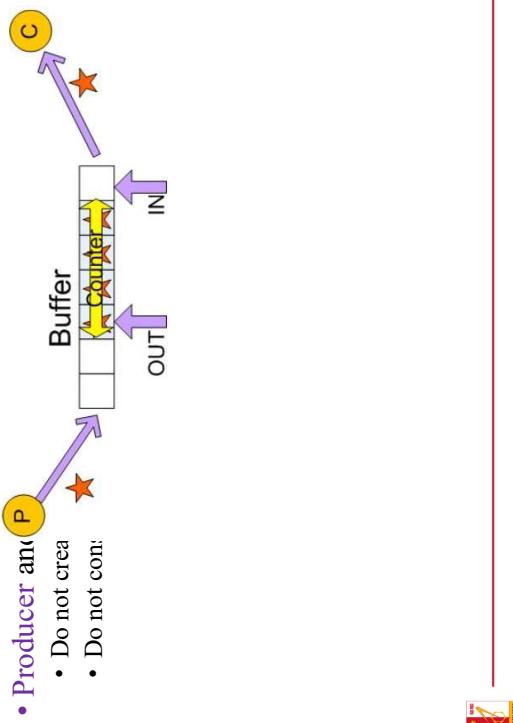
counter--

R1 = 5  
counter = 5  
R2 = ?  
counter = 5  
t

- 2 processes in the system

### producer-consumer I

• Producer and P  
• Do not crea  
• Do not con:



### Producer-Consumer Problem II

Producer

```
while(1){
    while(Counter == 0);
    /* do nothing */
    nextConsumed = Buffer[OUT];
    OUT = (OUT + 1) % BUFFER_SIZE;
    Counter--; /* consume the item in
    nextConsumed */
}
```

Consumer

```
while(1){
    while(Counter == 0);
    /* produce an item in
    nextProduced */
    while (Counter == BUFFER_SIZE);
    /* do nothing */
    Buffer[N] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}
```

### Remark

- Producer creates 1 product
- Consumer consumes 1 product
- ⇒ Number of product inside Buffer does not change

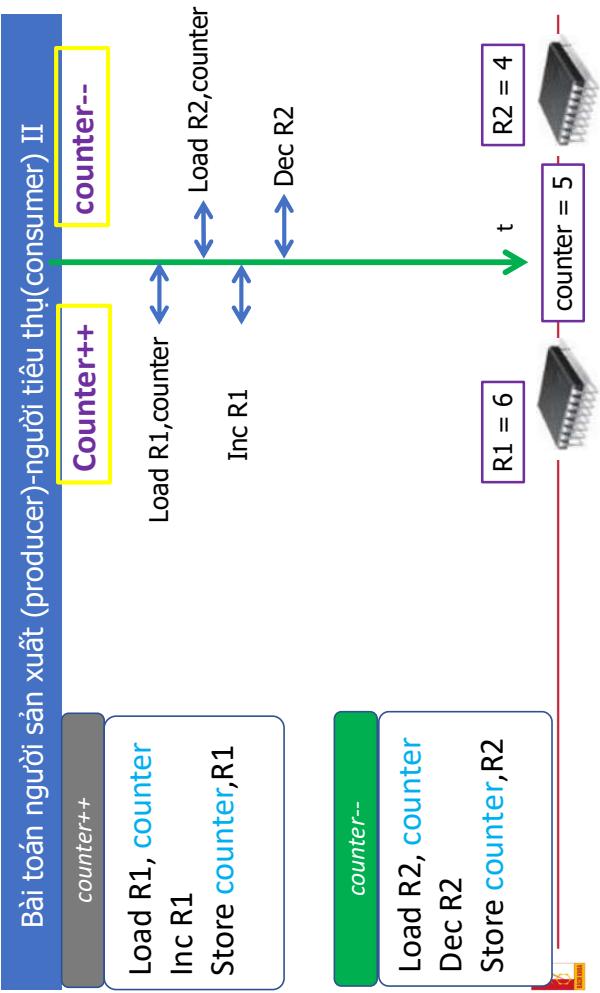


## Chapter 2 Process Management

### 4. Critical resource and process synchronization

## Chapter 2 Process Management

### 4. Critical resource and process synchronization

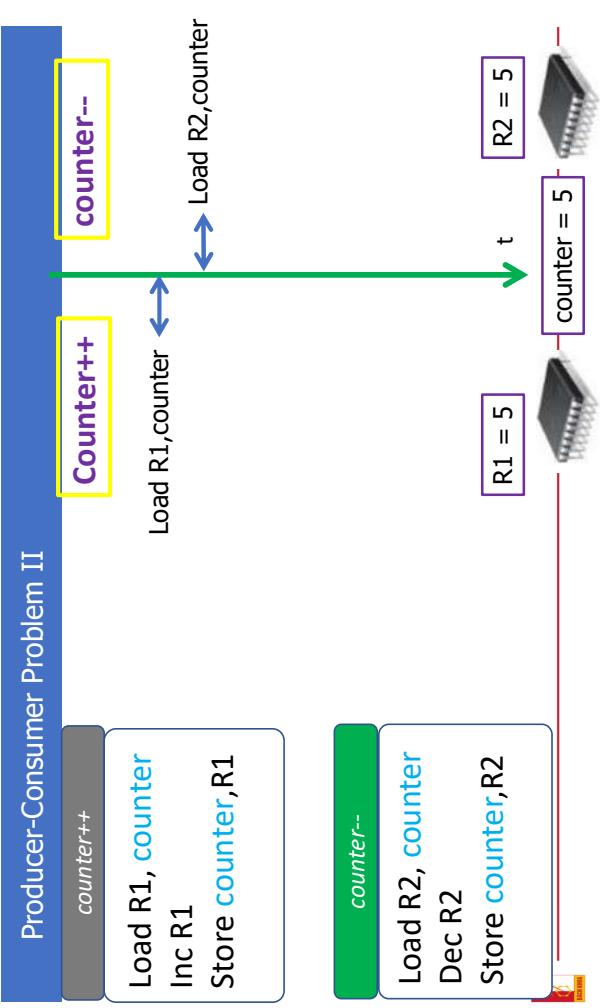


## Chapter 2 Process Management

### 4. Critical resource and process synchronization

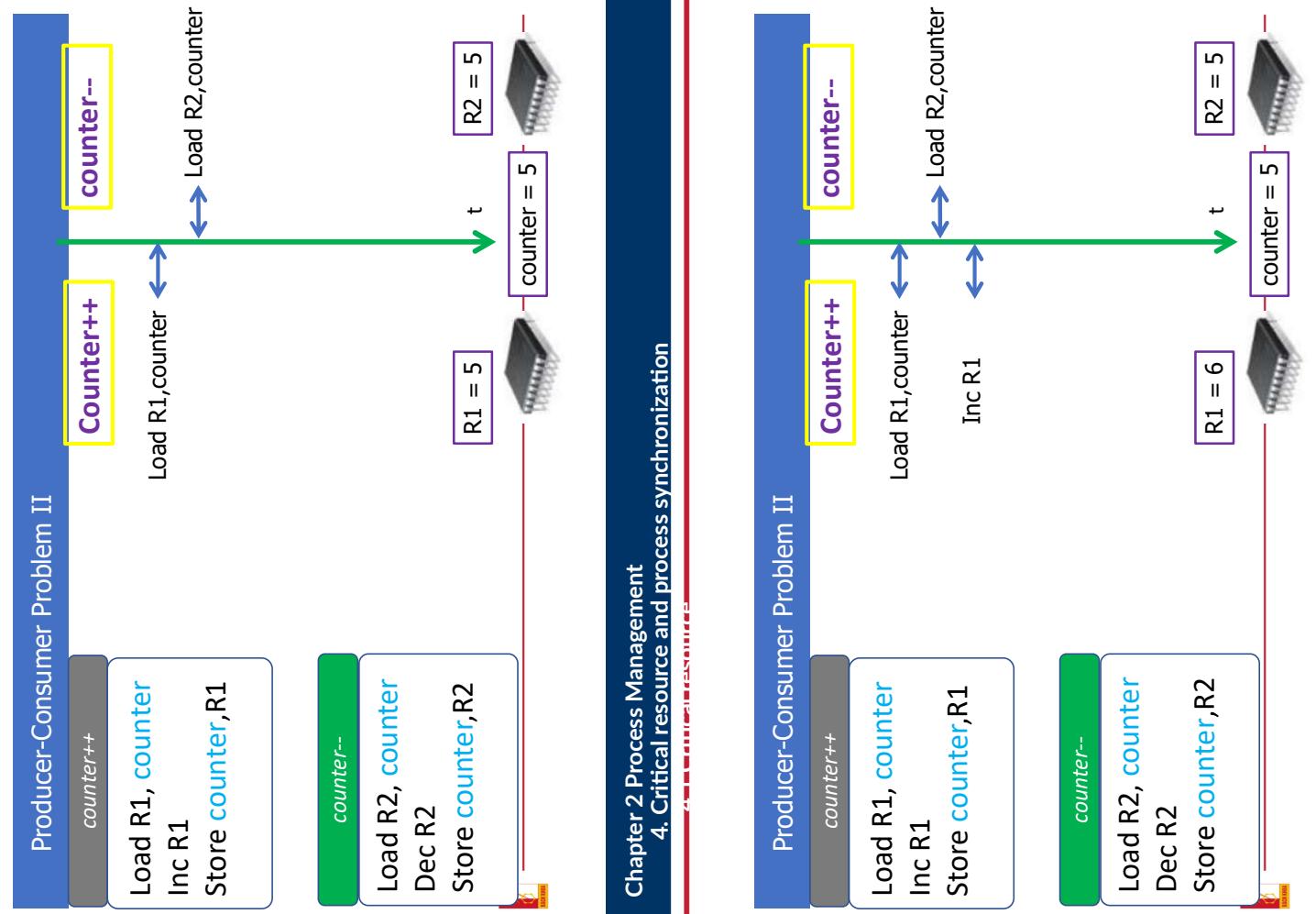
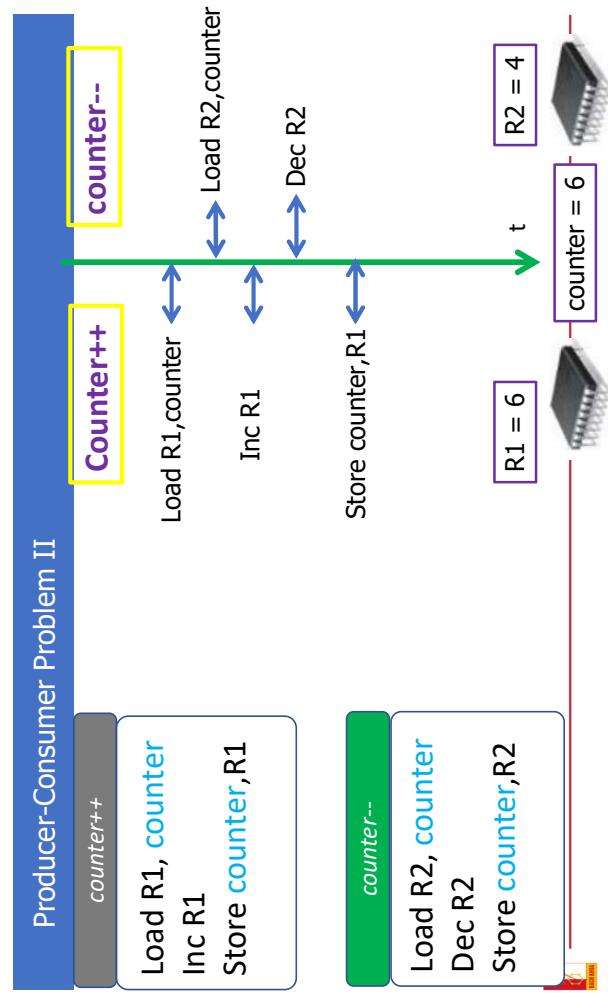
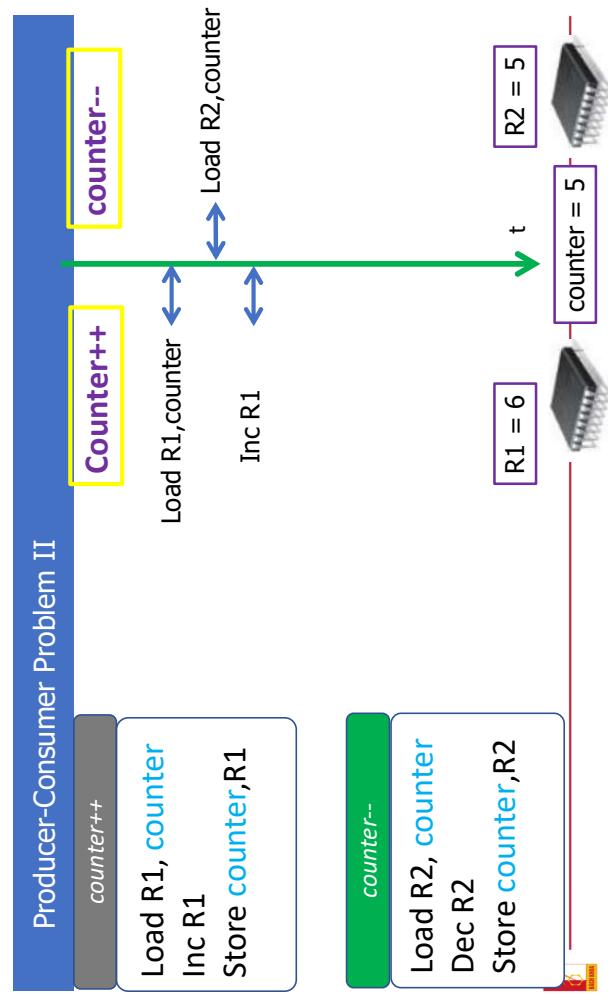
## Chapter 2 Process Management II

### 4. Critical resource and process synchronization

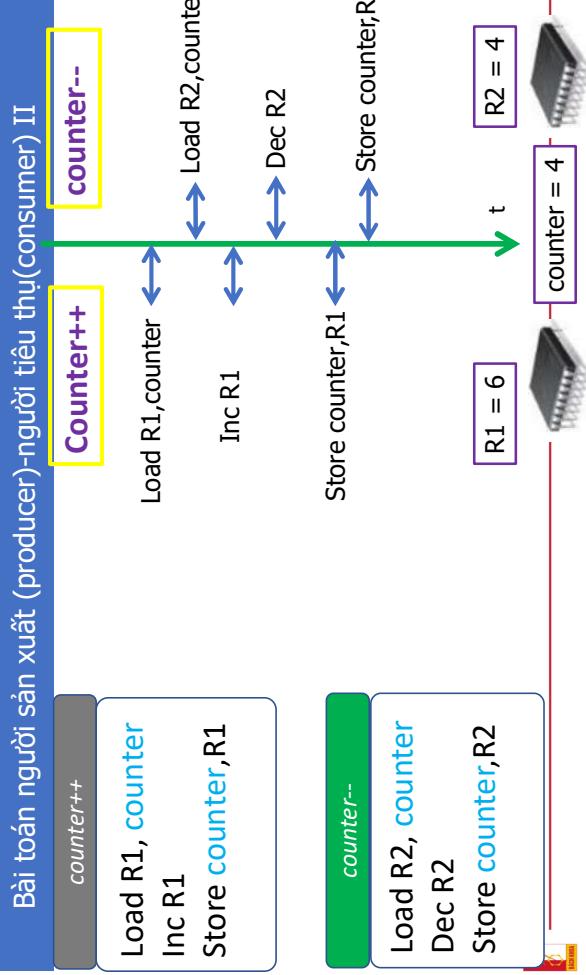
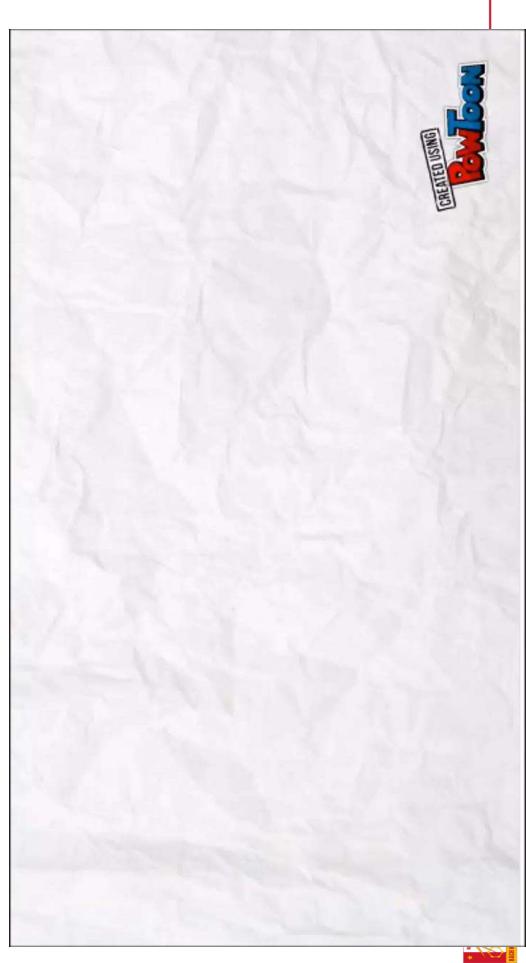


## Producer-Consumer Problem II

### 4. Critical resource and process synchronization

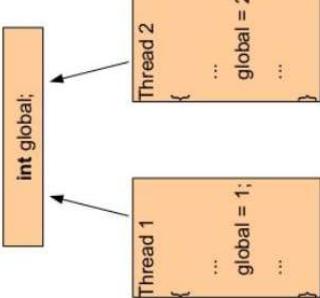


## Race condition



## Race condition

- Situation where the results of many processes access the sharing data depend of the order of these actions
- Make program's result undefinable



## Definition

### Resource

Everything that is required for process's execution

### Critical resource

- Resource that is limited of sharing capability
  - Required concurrently by processes
- Can be either physical devices or sharing data

### Problem

Sharing critical resource may not guarantee data completeness  
⇒ Require process synchronization mechanism



## Conditions to have a good solution

- **Mutual Exclusion:** Critical resource does not have to serve the number of process more than its capability at any time
  - If process **P<sub>i</sub>** is executing in its **critical section**, then **no other processes** could execute in their critical sections



## Race condition

- Prevent race condition by **synchronize concurrent running processes**
  - **Only 1** process can access sharing data **at a time**
    - Variable counter in Producer-Consumer problem
    - The **code segments** that **access sharing data** in the processes must be **executed** in a **defined order**
    - E.g.:  $x \leftarrow y + 1$  instruction in Thread T<sub>1</sub> only both two instruction of Thread T<sub>2</sub> are done



## Conditions to have a good solution

- Mutual Exclusion:
- **Progressive:** If **critical resource still able to serve** and there are process want to be executed in critical section then this process can use critical resource
  - If there are **more than 1** process use **critical resource**, then we must **synchronize** them
    - Object: guarantee that **no more than 1** process can stay **inside** critical section



## Methods' classification

- Low level method
  - Variable lock
  - Test and set
  - Semaphore
- High level method
  - Monitor



## Conditions to have a good solution

- Mutual Exclusion:
  - Progressive:
- Bounded Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted



## Critical resource

### Variable lock method

#### Test and Set method

#### Semaphore mechanism

#### Process synchronization example

#### Monitor

- There are 2 processes P1&P2 concurrently running
- Sharing the same critical resource
- Each process put the critical section at begin and the remainder section is next
  - Process must ask before enter the critical section {entry section}
  - Process perform {exit section} after exiting from critical section
- Program structure

```
do {
```

```
    entry section
```

```
    critical section
```

```
    exit section
```

```
    remainder section
```

```
} while (1);
```



## Algorithm

- Share var C1,C2 Boolean // sharing variable used as lock
- Initialization C1 = C2 = false // Critical resource is free

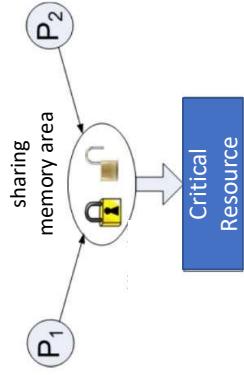
```
Process P1
do{
    while(C2 == true);
    C1 ← true;
    { Process P1's critical section }
    C1 ← false;
    { Process P1's remaining section }
}while(1);
```

Process P2

```
do{
    while(C1 == true);
    C2 ← true;
    { Process P2's critical section }
    C2 ← false;
    { Process P2's remaining section }
}while(1);
```

## Principle

- Share var C1,C2 Boolean // sharing variable used as lock
- Initialization C1 = C2 = false // Critical resource is free



## Principle

## Principle

- Each process uses 1 byte in the sharing memory area as a lock
- enters critical section, lock (byte lock = true)
- exits from critical section, unlock (byte lock= false)
- Process want to enter critical section: check other process's lock's status
- Locking ⇒ Wait
- Not lock ⇒ Has the right to enter critical section

```
Process P1
do{
    C1 ← true;
    while(C2 == true);
    { Process P1's critical section }
    C1 ← false;
    { Process P1's remaining section }
}while(1);
```

Process P2

```
do{
    C2 ← true;
    while(C1 == true);
    { Process P2's critical section }
    C2 ← false;
    { Process P2's remaining section }
}while(1);
```

- Synchronize properly for all cases

Remark

languages

- Complex when the number of processes and resources increase
  - “busy waiting” before enter critical section
  - When waiting, process must check **the right to enter the critical section** => Waste processor’s time

- Not properly synchronize

Remark

- Mutual exclusion problem (Case 1)
  - Progressive problem (Case 2)
- Reason: The following actions are done separately
  - Test the right to enter critical section
  - Set the right to enter critical section

Chapter 2 Process Management  
4. Critical resource and process synchronization

**Chapter 2 Process Management**  
4. Critical resource and process synchronization

## •Critical resource

Variable local method

Test and Set method

- Semaphore mechanism
  - Process synchronization example
  - Monitor

100

Chapter 2 Process Management

Chapter 2 Process Management

- Utilize **turn** variable to show process with priority

Dekker's algorithm

*Process P1*      *Process P2*

do{

```

 $C_2 \leftarrow \text{true};$ 
 $\text{while}(C_1 == \text{true})\{$ 
 $\quad \text{if}(\text{turn} == 1)\{$ 
 $\quad \quad C_2 \leftarrow \text{false};$ 
 $\quad \quad \text{while}(\text{turn} == 1);$ 
 $\quad \quad C_2 \leftarrow \text{true};$ 
 $\}$ 
 $\}$ 

```

```

    { P2's critical section }
turn = 1;
C2 ← false;
{ P2s remaining section }
}while(1);

```

```

 $C_1 \leftarrow \text{true};$ 
 $\text{while}(C_2 == \text{true})\{$ 
 $\quad \text{if}(\text{turn} == 2)\{$ 
 $\quad \quad C_1 \leftarrow \text{false};$ 
 $\quad \quad \text{while}(\text{turn} == 2);$ 
 $\quad \quad C_1 \leftarrow \text{true};$ 
 $\}$ 
 $\}$ 

```

```

    } Process P1's critical section
    turn = 2;
    C1 ← false;
}
{ P1's remaining section
}while(1);

```

#### • Choosing a lock mechanism.

##### Algorithm with TestAndSet instruction

- Free ( $Lock=false$ )
- Initialization:  $Lock = false \Rightarrow$  Resource is free
- Algorithm for process  $P_i$

```
do{  
    while(TestAndSet(Lock));  
    {  
        Process P's critical section  
    }  
    Lock = false;  
    {  
        P's remaining section  
    }  
}while(1);
```

#### • Sharing variable $lock$ causes the resource's static

##### Algorithm with Swap instruction

- Initialization:  $Lock = false \Rightarrow$  Resource is free
- Algorithm for process  $P_j$

```
do{  
    key = true;  
    while(key == true)  
        swap(Lock, Key);  
    {  
        Process P's critical section  
    }  
    Lock = false;  
    {  
        P's remaining section  
    }  
}while(1);
```

#### • Using a lock mechanism.

##### Principle

- Test and change the content of 1 word
- Swap the content of 2 different words
- Instruction is executed atomically
- Code block is uninterruptible when executing
- When called at the same time, done in any order

```
boolean TestAndSet(VAR boolean target) {  
    boolean rv = target;  
    target = true;  
    return rv;  
}  
  
void Swap(VAR boolean , VAR boolean b) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

- An **integer variable**, **initialized by resource sharing capability**
  - Number of available resources (e.g. 3 printers)
  - Number of resource's unit (10 empty slots in buffer)
- Can **only** be changed by 2 operation P and V
- Operation **P(S)** (or **wait(S)**)

```
wait(S) {
    while(S <= 0) no-op;
    S--;
}
```

- Operation **V(S)** (or **signal(S)**)

```
signal(S) {
    S++;
}
....
```

- P and V are uninterruptible instructions



## Remark

- Simple, complexity is **not increase** when number of processes and critical resource increase
- “busy waiting” before enter critical section
  - When waiting, process has to check if sharing resource is free or not  
=> Waste processor's time
- No **bounded waiting guarantee**
  - The next process enters critical section is dependent on the resource release time of current **resource-using** process

⇒ Need to be solved

## Semaphore usage I

- *n*-process critical-section problem
  - processes share a semaphore, mutex
  - initialized to 1.
  - Each process Pi is organized as

do {

```
(wait(mutex));
```

critical section

```
signal(mutex);
```

remainder section

} while () ;

## Critical resource

- Variable lock method
- Test and Set method
- **Semaphore mechanism**
- Process synchronization example
- Monitor



To overcome the need for busy waiting (cont. 1)

- **Block()** Temporarily suspend running process
  - it must wait. (block itself -> not busy waiting)
  - **block operation** places a process into a waiting queue associated with the **semaphore**,
    - **Process's state** is switched to the **waiting**
    - Control is transferred to
      - the CPU scheduler,



To overcome the need for busy waiting (cont.2)

- **Wakeup(P)** Resume process P suspended by block() operation

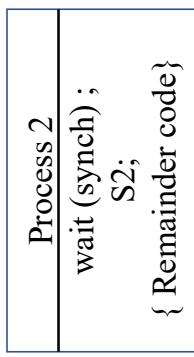
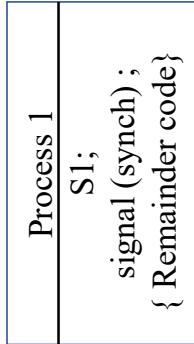
- process that is blocked,
  - waiting on a semaphore S,
  - restarted when some other process executes a signal operation.
- The process is **restarted** by a **wakeup** operation
  - changes the process from the **waiting** state to the ready **state**.
  - process is then placed in the **ready** queue.



Semaphore usage II

- The order of execution inside processes:
  - P1 with a statement S1 , P2 with a statement S2.
    - require that S2 be executed only after S1 has completed

P1 and P2 share a common semaphore **synch**, initialized to 0, Code for each process



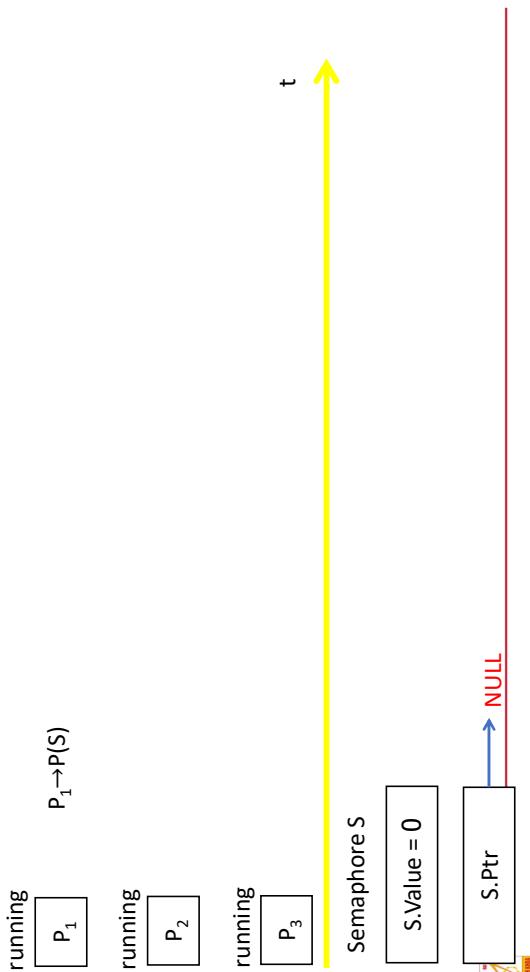
To overcome the need for busy waiting

Use 2 operations

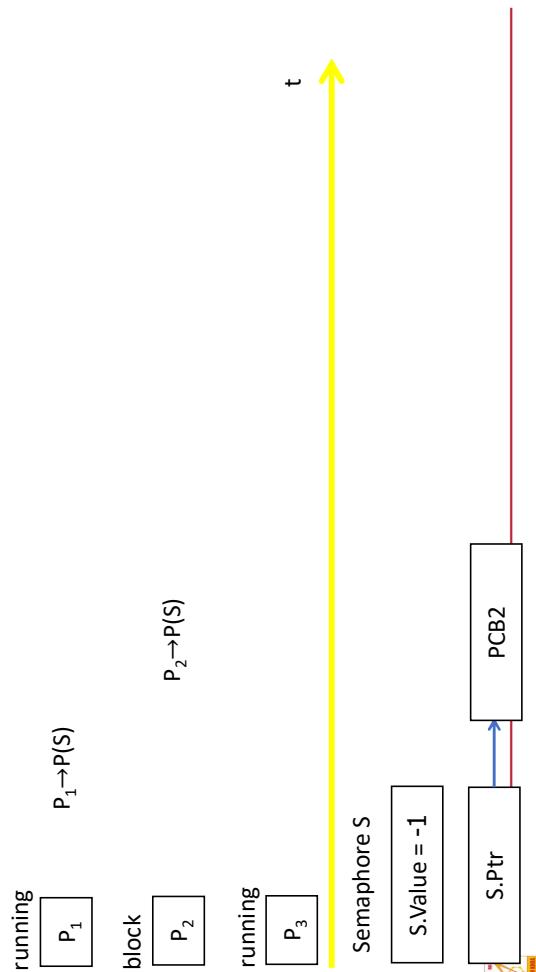
- **Block()** Temporarily suspend running process
  - Wakeup(P) Resume process P suspended by block() operation



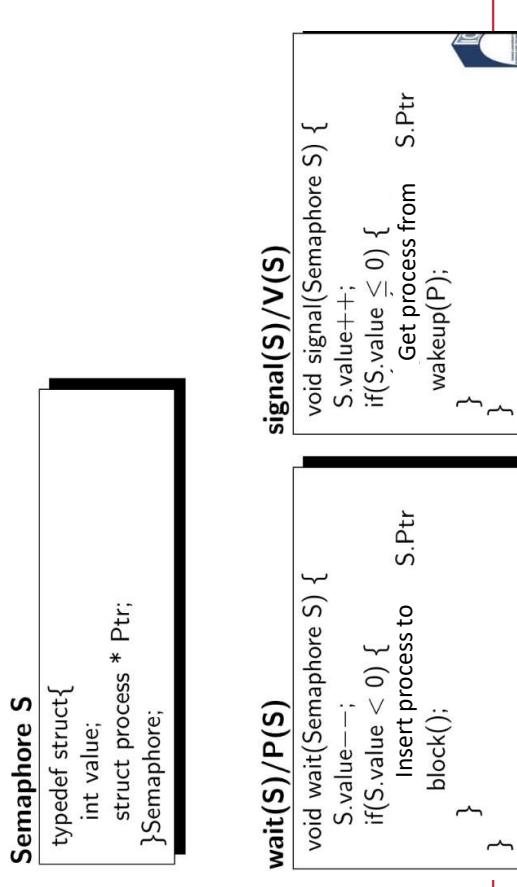
## Synchronization example



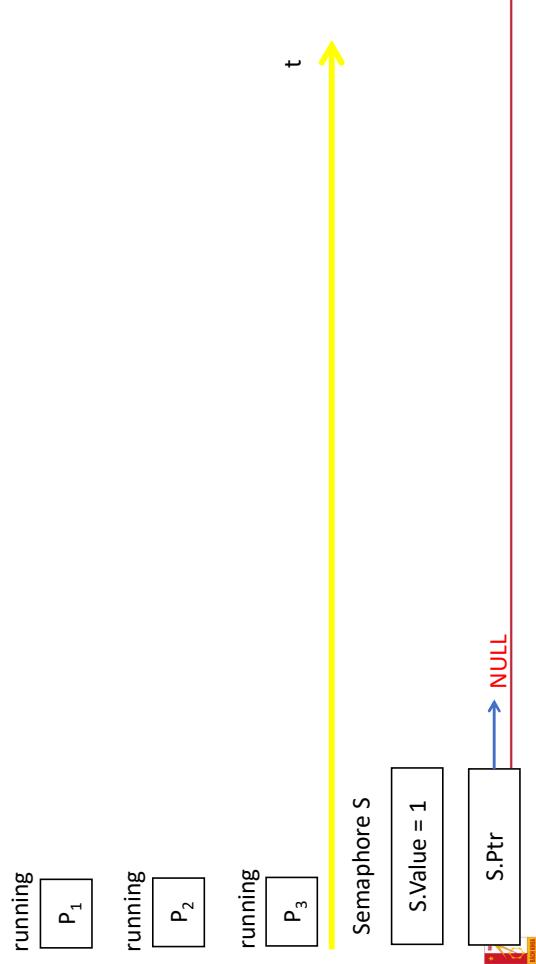
## Synchronization example



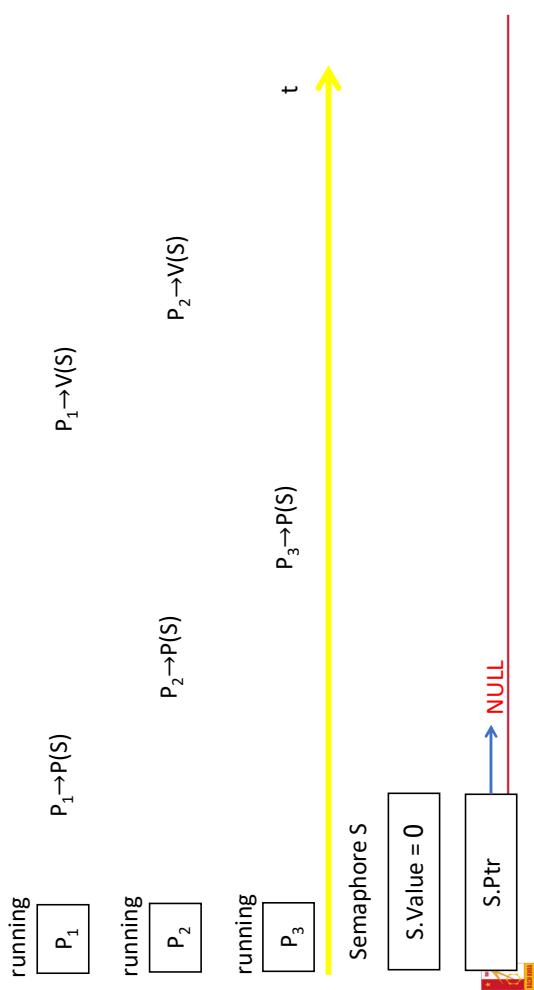
## Semaphore implementation



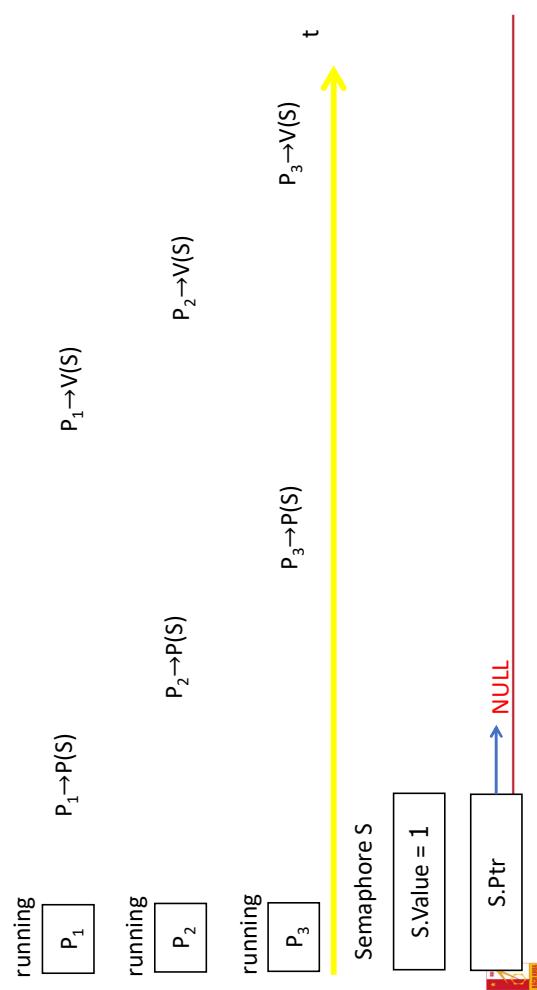
## Synchronization example



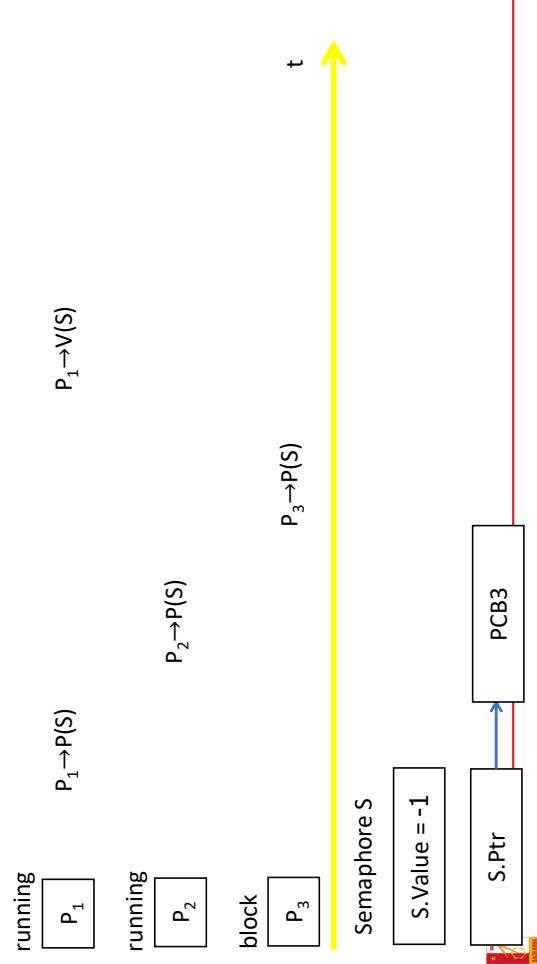
### Synchronization example



### Synchronization example



### Synchronization example



Remark	→ <b>dead synchronization</b>
<ul style="list-style-type: none"><li>● P(S) and V(S) is nonshareable</li></ul>	<ul style="list-style-type: none"><li>● Uniprocessor system: Forbid interrupt when perform wait(), signal()</li><li>● Multiprocessor system<ul style="list-style-type: none"><li>● Not possible to forbid interrupt on other processors</li><li>● Use variable lock method ⇒ busy waiting, however waiting time is short (10 commands)</li></ul></li></ul>

### Remark

卷之三

- Multiprocessor system: Forbid interrupt when perform wait(), signal()
  - Not possible to forbid interrupt on other processors
  - Use variable lock method  $\Rightarrow$  busy waiting, however waiting time is short (10 commands)

**Chapter 2 Process Management**  
**4. Critical resource and process synchronization**

Create Samanhar( ). Create Samanhar

Semaphores are objects in Windows API

- ⇒ pointer to a SECURITY\_ATTRIBUTES structure, handle can be inherited
  - LONG InitialCount**, ⇒ initial count for Semaphore object
  - LONG MaximumCount**, ⇒ maximum count for Semaphore object
  - LPCTSTR lpName** ⇒ Name of Semaphore object

xample: CreateSemaphore(NULL,0,1,NULL);

Return HANDLE of Semaphore object or NULL

```
WaitForSingleObject(HANDLE h, DWORD time)
```

Digitized by srujanika@gmail.com

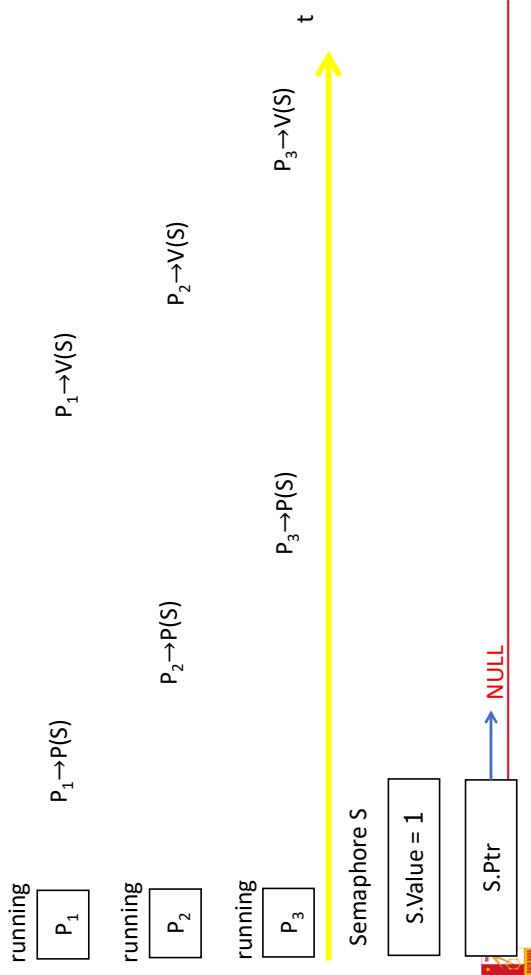
HANDLE `hSemaphore`,  $\Leftarrow$  handle for a Semaph

**LONG** **|ReleaseCount**,  $\Leftarrow$  increase semaphore value

`LPLONG lpPreviousCount` =< pointer to a variable

count

- Example: `ReleaseSemaphore(S, 1, NULL);`



Synchronization example

Digitized by srujanika@gmail.com

- Not possible to forbid interrupt on other processors
  - Use variable lock method  $\Rightarrow$  busy waiting, however short (10 commands)

**Chapter 2 Process Management**  
**4. Critical resource and process synchronization**

- Easy to apply for complex system

Demande

- ## Remark



卷之三

Wrong command  
Wrong order

- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism

## ● Variable lock method

## ● Test and Set method

## ●Semaphore mechanism

## ● Process synchronization examples

Monitor

## Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
HANDLE S1, S2;
void T1();
void T2();
int main()
{
    HANDLE h1, h2;
    DWORD ThreadId;
    S1 = CreateSemaphore(0, 1, 1, "S1");
    S2 = CreateSemaphore(0, 1, 1, "S2");
    h1 = CreateThread(0, 0, T1, 0);
    h2 = CreateThread(0, 0, T2, 0);
    WaitForSingleObject(h1, INFINITE);
    WaitForSingleObject(h2, INFINITE);
    CloseHandle(h1);
    CloseHandle(h2);
    CloseHandle(S1);
    CloseHandle(S2);
}
```



- Producer-Consumer problem
- Dining Philosophers problem
- Readers-Writers
- Sleeping Barber
- Bathroom Problem

- Producer-Consumer problem
- Dining Philosophers problem

## ● Readers-Writers

## ●Sleeping Barber

## •Bathroom Problem

## Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
HANDLE S1, S2;
void T1();
void T2();
int main()
{
    HANDLE h1, h2;
    DWORD ThreadId;
    S1 = CreateSemaphore(0, 1, 1, "S1");
    S2 = CreateSemaphore(0, 1, 1, "S2");
    h1 = CreateThread(0, 0, T1, 0);
    h2 = CreateThread(0, 0, T2, 0);
    WaitForSingleObject(h1, INFINITE);
    WaitForSingleObject(h2, INFINITE);
    CloseHandle(h1);
    CloseHandle(h2);
    CloseHandle(S1);
    CloseHandle(S2);
}
```



Chapter 2 Process Management  
4. Critical resource and process synchronization

Chapter 2 Process Management  
4. Critical resource and process synchronization



Exame

```

void T1() {
    while(1) {
        WaitForSingleObject(S1, INFINITE);
        x = y + 1;
        ReleaseSemaphore(S2, 1, NULL);
        printf ("%d", x);
    }
}

void T2() {
    while(1) {
        y = 2;
        ReleaseSemaphore(S1, 1, NULL);
        WaitForSingleObject(S2, INFINITE);
        y = 2 * y;
    }
}

```



## Producer-Consumer problem

Solution: Utilize 1 semaphore Mutex to synchronize variable Counter  
Initialization: Mutex = 1

```
while(1) {
    /*produce a product in Buffer */
    if(Counter==SIZE) block();
    /*do nothing*/
    nextConsumed = Buffer[OUT];
    OUT=(OUT + 1) % BUFFER_SIZE;
    Counter--;
    /*consume the item in Buffer*/
}
if(Counter==1) wakeup(Producer);
Producer
```

**Problem:** Assume Counter=0

- Consumer check counter => call block()
- Producer increase counter by 1 and call wakeup(Consumer)
- Consumer not blocked yet => wakeup() is skipped

## Chapter 2 Process Management 4. Critical resource and process synchronization

## Producer-Consumer problem

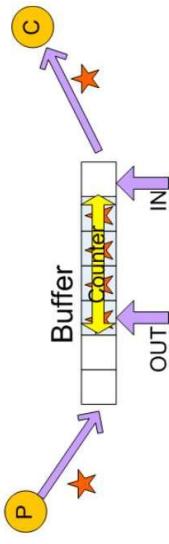
Solution 2: Utilize 2 semaphore full, empty.  
Initialization: full ← 0 : Number of item in buffer  
empty←BUFFER\_SIZE: Number of empty slot in buffer

```
do{
    {Create new product}
    wait(empty);
    {Take out 1 product from Buffer}
    signal(empty);
    {Consume product}
    }while (1);
Producer
```

**Consumer** Running  
**Producer** Running

empty = 0  
full = 5

## Producer-consumer problem



```
while(1){
    while(Counter == 0);
    /*do nothing*/
    nextConsumed = Buffer[OUT];
    OUT=(OUT + 1) % BUFFER_SIZE;
    Counter--;
    /*consume the item in
    nextConsumed*/
}

while(1) {
    /*produce an item in Buffer*/
    while (Counter == BUFFER_SIZE);
    /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}

Producer
```

**Consumer**

## Chapter 2 Process Management 4. Critical resource and process synchronization

## Producer-Consumer problem

```
while(1){
    if(Counter == 0) block();
    /*do nothing*/
    nextConsumed = Buffer[OUT];
    OUT=(OUT + 1) % BUFFER_SIZE;
    Counter--;
    /*consume the item in
    Buffer*/
}

while(1) {
    /*produce an item in Buffer*/
    if(Counter==SIZE) block();
    /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
    if(Counter==SIZE-1) wakeup(Producer);
}

Producer
```

**Consumer**

## Bathroom Problem

- A bathroom is to be used by both men and women, but not at the same time
- If the bathroom is empty, then anyone can enter
- If the bathroom is occupied, then only a person of the same sex as the occupant(s) may enter
- The number of people that may be in the bathroom at the same time is limited

- Problem implementation require to satisfy constraints
  - 2 types of process: male() và female()
  - Each process enter the Bathroom in a random period of time



## Readers and Writers problem

- Many **Reader** processes access the database at the same time
- Several **Writer** processes update the database
- Allow unlimited **Readers** to access the database
  - 1 **Reader** process is accessing the database, new **Reader** process can access the database
  - (Writers have to stay in waiting queue)
- Allow only 1 **Writer** process to update the database at a time
- Non-preemptive problem. Process stays inside critical section without being interrupted



## Dining philosopher problem

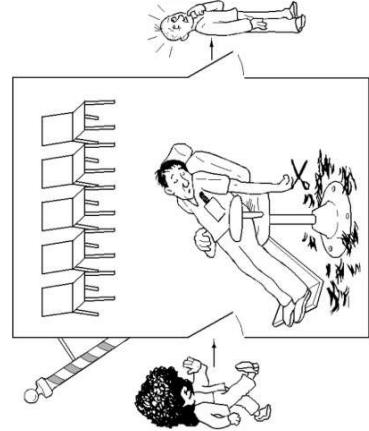
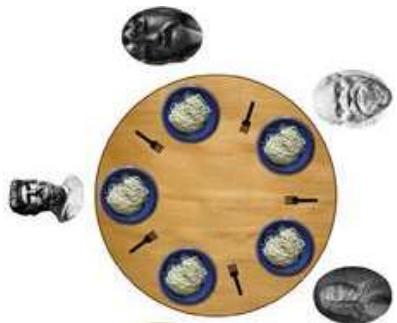
Classical synchronization problem, show the situation where many processes share resources

- 5 philosophers having dinner at a round table
- In front each person is a disk of spaghetti
- Between two disk is a fork
- Philosopher do 2 things : Eat and Think
- Each person need two forks to eat
- Take only one fork at a time
- Take the left fork then the right fork
- Finish eating, return the fork to original place



## Sleeping barber

- N waiting chair for client
- Barber can cut for one client at a time
- No client, barber go to sleep
- When client comes
  - If barber is sleeping ⇒ wake him up
  - If barber is working
    - Empty chair exists ⇒ sit and wait
    - No empty chair left ⇒ Go away



## Dining philosopher problem – Solution 1

- Philosopher take the forks with different order
- Even id philosopher take the even id fork first
- Odd id philosopher take the odd id fork first

```
do{
    j = i%2
    wait(fork[(i + j)%5])
    wait(fork[(i+1 - j)% 5]);
    {Eat}
    signal(fork[(i+1 - j)% 5]);
    signal((i + i)%5);
    {Thinks}
} while (1);
```

- Solve the deadlock problem



- If all the philosophers want to eat
  - Take the left fork (call to: wait(fork[i]))
  - Wait for the right fork (call to: wait(fork[(i+1)%5]))

⇒ **deadlock**

## Dining philosopher problem: Simple method

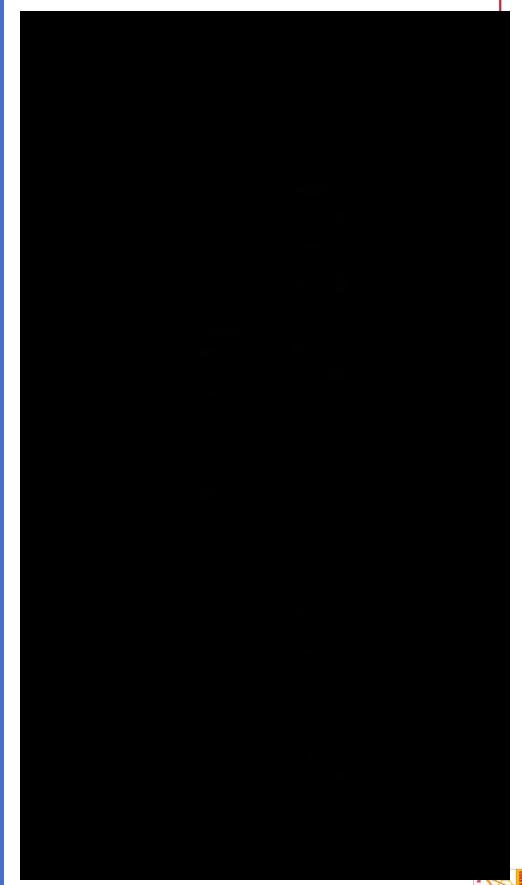
- Each fork is a critical resource, synchronized by a semaphore `fork[i]`
- Semaphore `fork[5] = {1, 1, 1, 1, 1}`;
- Algorithm for philosopher Pi

```
do{
    wait(fork[i])
    wait(fork[(i+1)% 5]);
    {Eat}
    signal(fork[(i+1)% 5]);
    signal(fork[i]);
    {Thinks}
} while (1);
```

- If all the philosophers want to eat
  - Take the left fork (call to: wait(fork[i]))
  - Wait for the right fork (call to: wait(fork[(i+1)%5]))

⇒ **deadlock**

## Demonstration



## Dining philosopher problem – Solution 1

- Allow only one philosopher to take the fork at a time
- Semaphore `mutex ← 1;`
- Algorithm for philosopher Pi

```
do{
    wait(mutex)
    wait(fork[i])
    wait(fork[(i+1)% 5]);
    signal(mutex)
    {Eat}
    signal(fork[(i+1)% 5]);
    signal(i);
    {Thinks}
} while (1);
```

- It's possible to allow 2 non-close philosopher to eat at a time (P1: eats, P2: owns mutex ⇒ P3 waits)



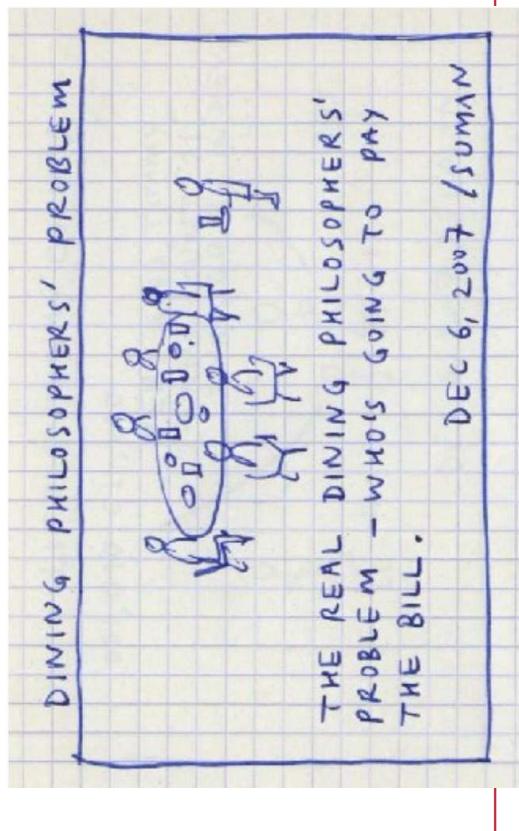
## Introduction

- Special data type, proposed by HOARE 1974
- Combines of procedures, local data, initialization code
- Process can only access variables via procedures of Monitor
- Only one process can work with Monitor at a time
  - Other processes have to wait
  - Allow process to wait inside Monitor
    - Utilize condition variable

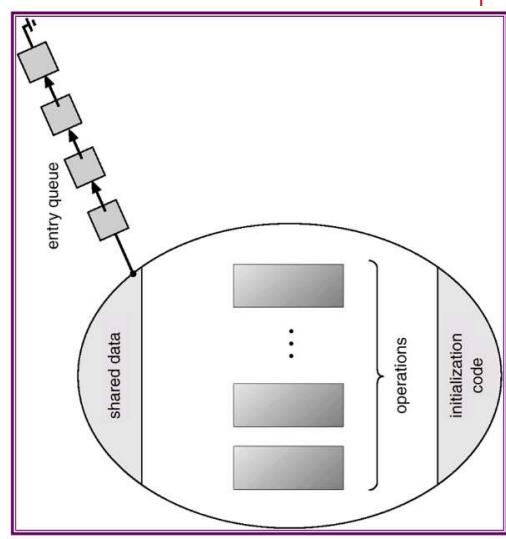
```
monitor monitorName{
    Sharing variables declarations ;
    procedure P1(...){
        ...
    }
    ...
    procedure Pn(...){
        ...
    }
    ...
}
Initialization code
```

Monitor's syntax

## True problem ?



## Model



- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization examples

## ● Monitor

## Monitor's usage: sharing a resource

```
Monitor Resource{
    Condition Nonbusy;
    Boolean Busy
    --- User's part
    void Acquire(){
        if(busy) Nonbusy.wait();
        busy=true;
    }
    void Release(){
        busy=false
        signal(Nonbusy)
    }
    --- Initialization part
    busy= false;
    Nonbusy = Empty;
}
```

Process's structure

```
while(1){
    ...
    Resource.Acquire()
    {
        Using resource
    }
    Resource.Release()
    ...
}
```

## Condition Variable

Actually, **name** of a queue

Declare: **condition** x,y;

Only used by 2 operations  
**wait()** Called by Monitor's procedures (**syntax x.wait() or wait(x)**).

Allow process to be **blocked until activated by other process**

via **signal()** procedure

```
signal() Called by Monitor's procedures (syntax x.signal() or signal(x)). Activate a process waiting at x variable queue.
```

If no waiting process then the operation is skipped



## Producer – Consumer problem

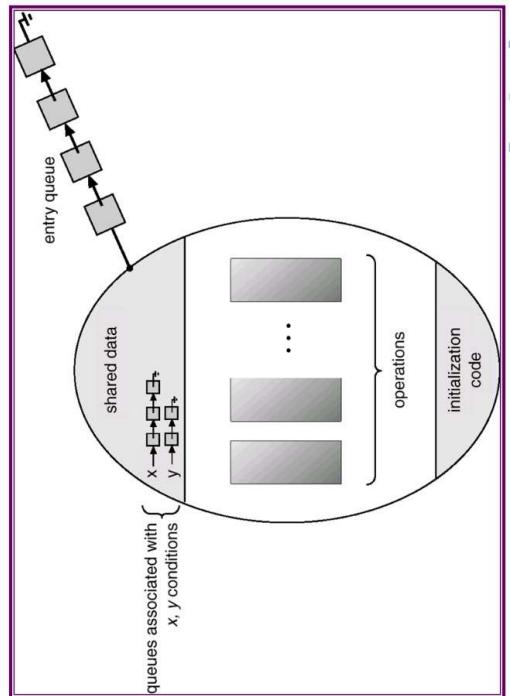
```
Monitor ProducerConsumer{
    Condition Full, Empty;
    int Counter ;
    void Put(Item){
        if(Counter=N) Full.wait();
        { Put item into Buffer };
        Counter++;
        if(Counter=1) Empty.signal()
    }
    void Get(Item){
        if(Counter=0) Empty.wait()
        {Take item from Buffer}
        Counter--;
        if(Counter=N-1) Full.signal()
    }
    Counter=0;
    Full, Empty = Empty;
}
```

ProducerConsumer M;

```
Producer
while(1){
    Item = New product
    M.Put(item)
    ...
}
```

```
Consumer
while(1){
    M.Get(&Item)
    {Use item}
    ...
}
```

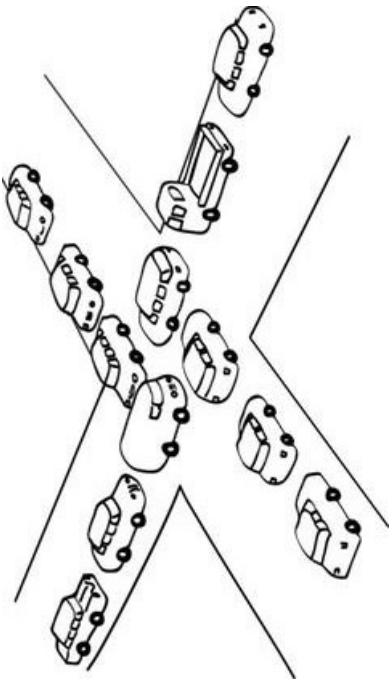
## Model



- ① Process
- ② Thread
- ③ CPU scheduling
- ④ Critical resource and process synchronization
- ⑤ Deadlock and solutions



- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

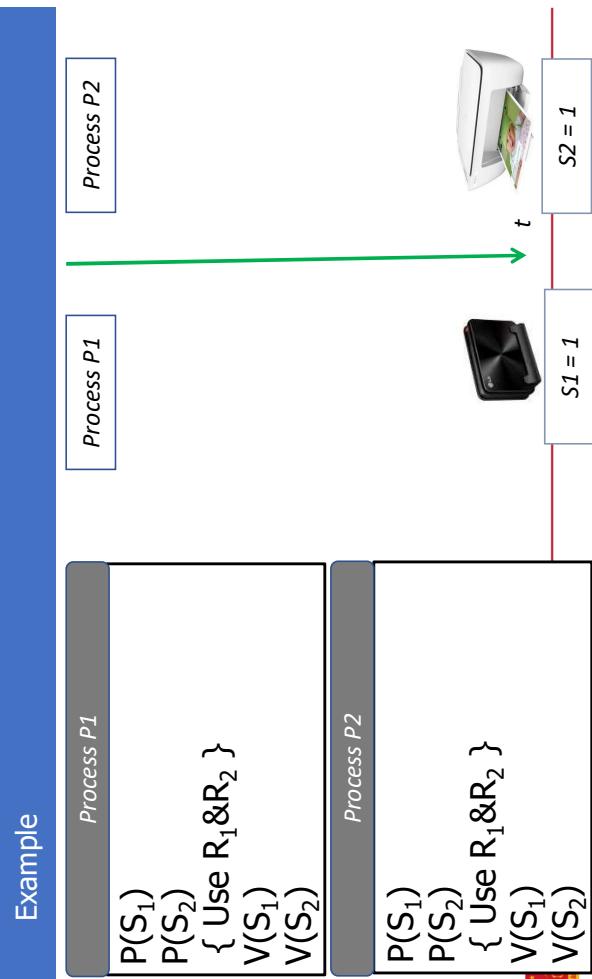


## Chapter 2 Process Management

### 5. Dead lock and solutions

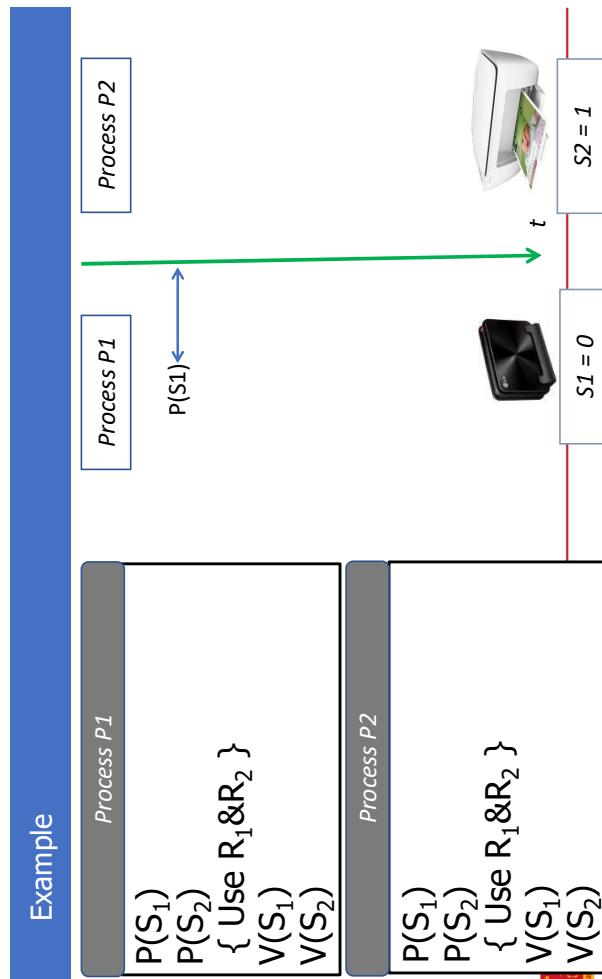
## Chapter 2 Process Management

### 5. Dead lock and solutions



## Chapter 2 Process Management

### 5. Dead lock and solutions



## Chapter 2 Process Management

### 5. Dead lock and solutions

## Deadlock conception

- System combines of **concurrently running processes, sharing resources**
  - Resources** have **different types** (e.g.: CPU, memory,..).
  - Each type of resource may has **many unit** (e.g.: 2 CPUs, 5 printers..)
- Each process is combines of sequences of continuous operations
  - Require resource**: if resource is not available (being used by other processes) ⇒ process has to wait
  - Utilize resource** as required (printing, input data...)
  - Release allocated resources**
- When processes share **at least 2 resources**, system may **"in danger"**

## Deadlock conception

```
P(S1)
P(S2)
{ Use R1&R2 }
V(S1)
V(S2)
```

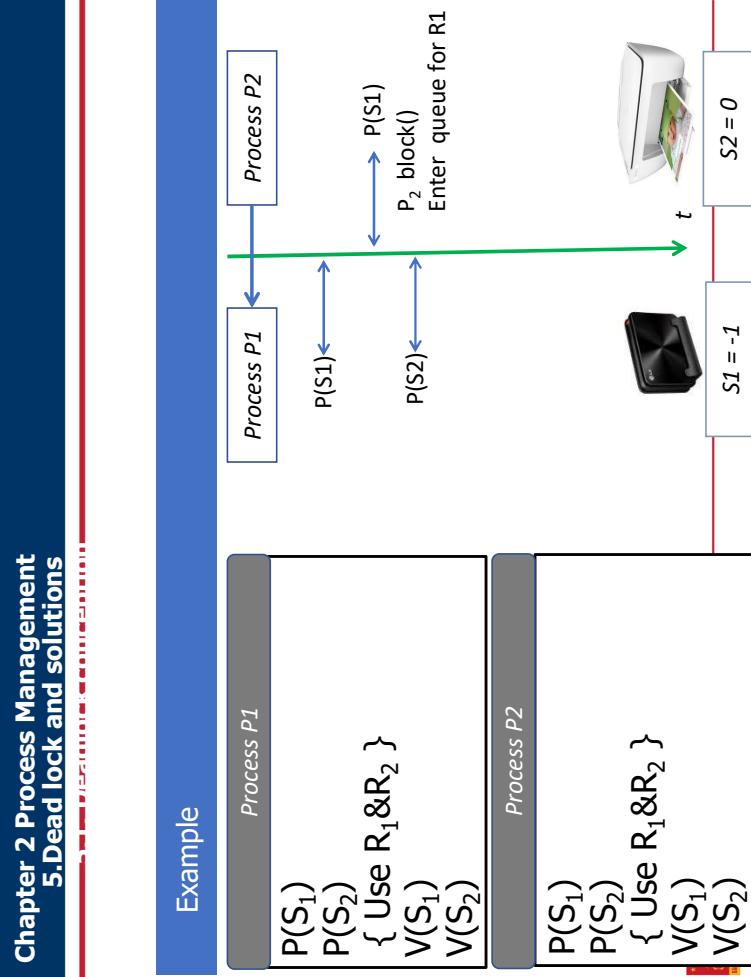
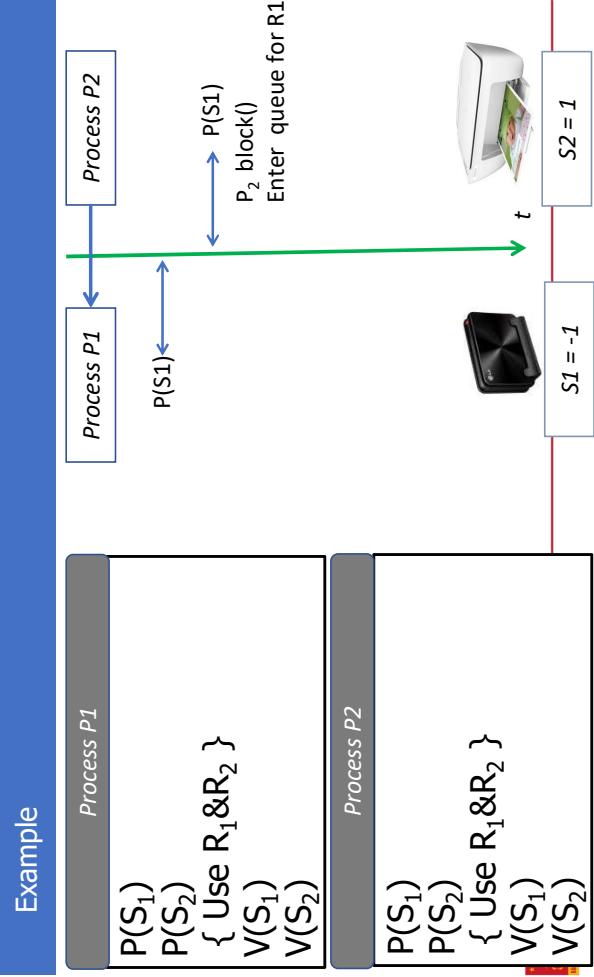
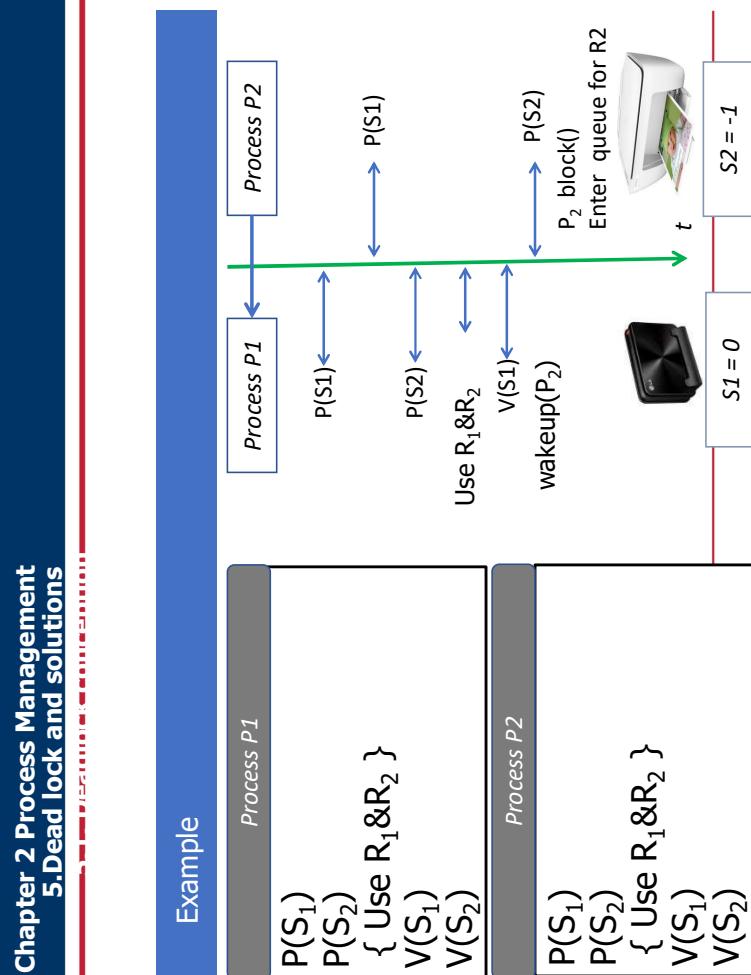
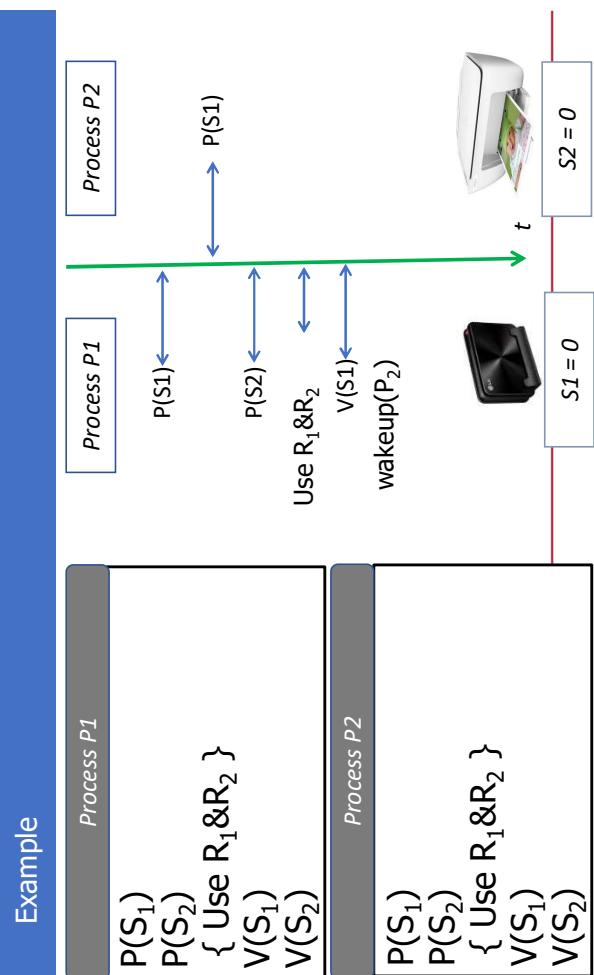


## Chapter 2 Process Management

### 5.Dead lock and solutions

## Chapter 2 Process Management

### 5.Dead lock and solutions

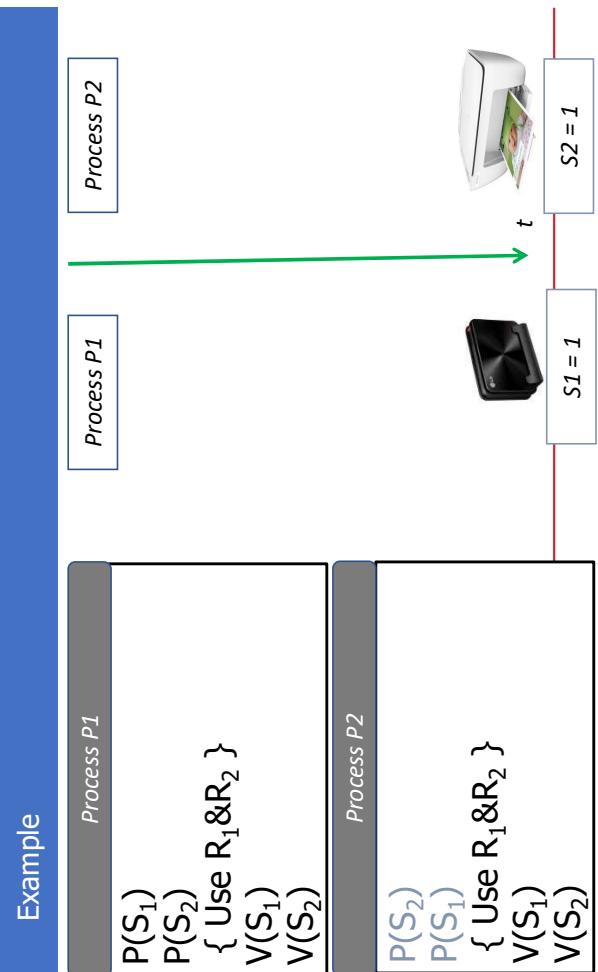


## Chapter 2 Process Management

### 5.Dead lock and solutions

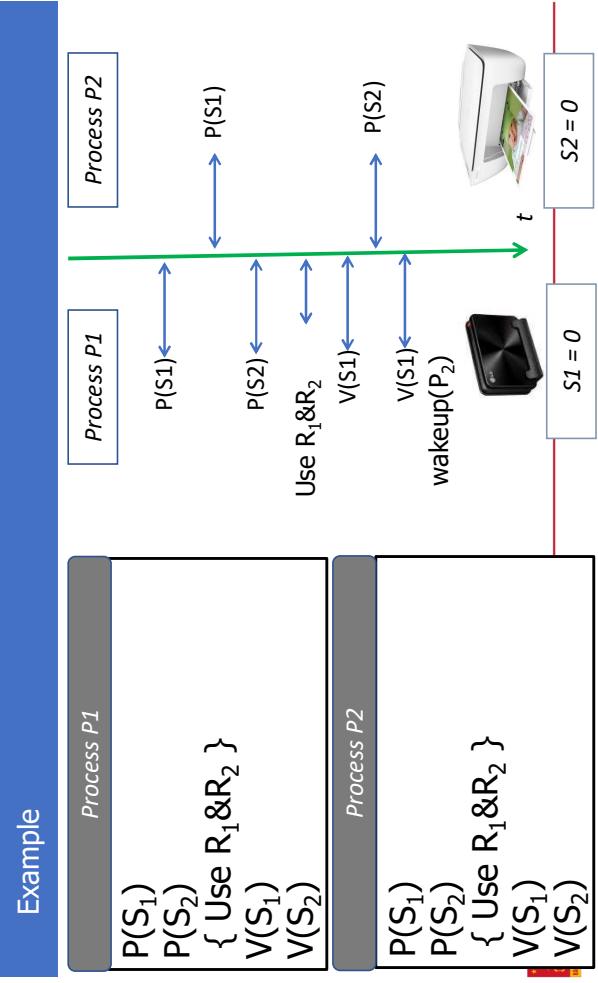
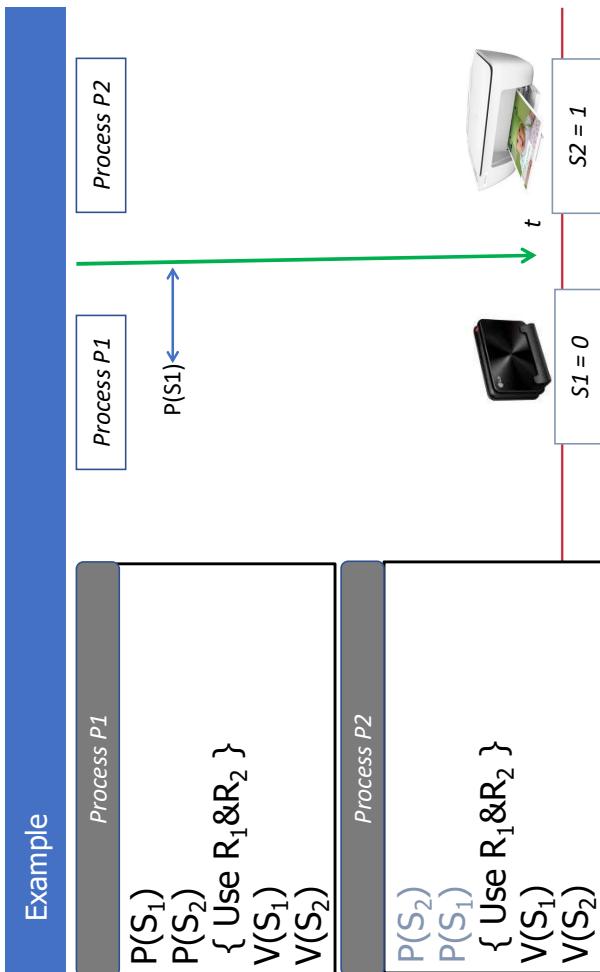
## Chapter 2 Process Management

### 5.Dead lock and solutions



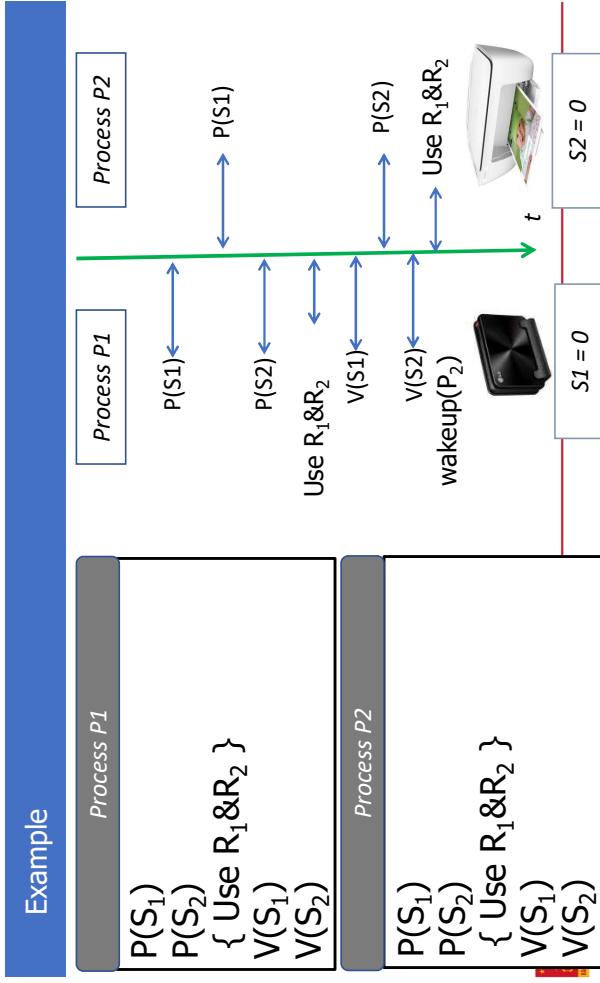
## Chapter 2 Process Management

### 5.Dead lock and solutions



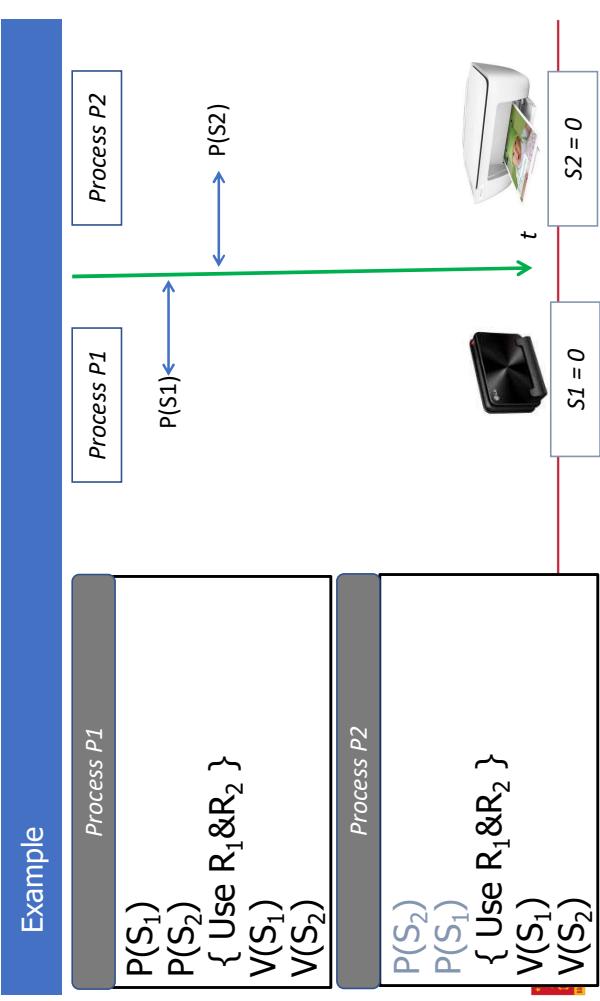
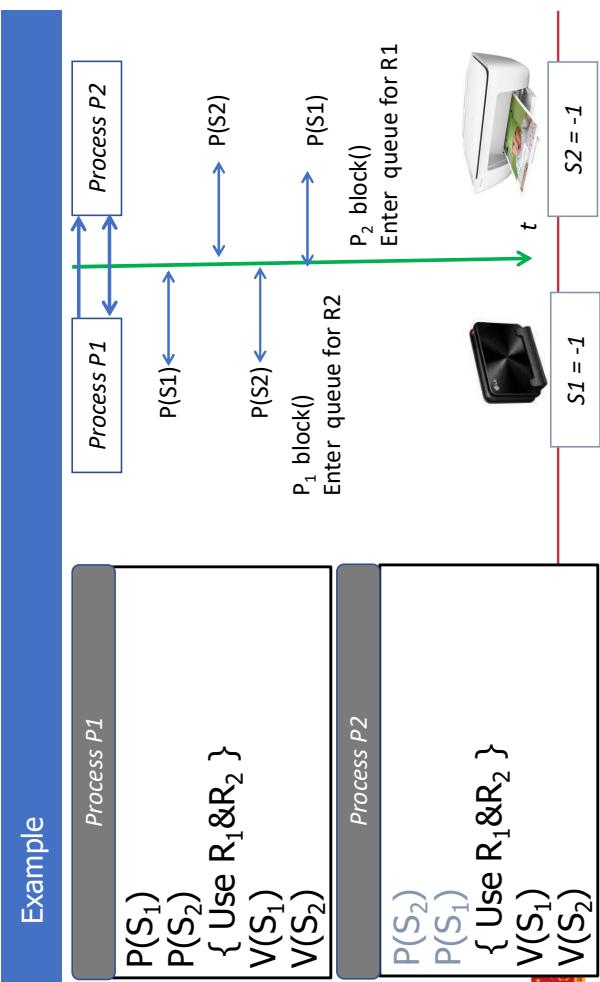
## Chapter 2 Process Management

### 5.Dead lock and solutions

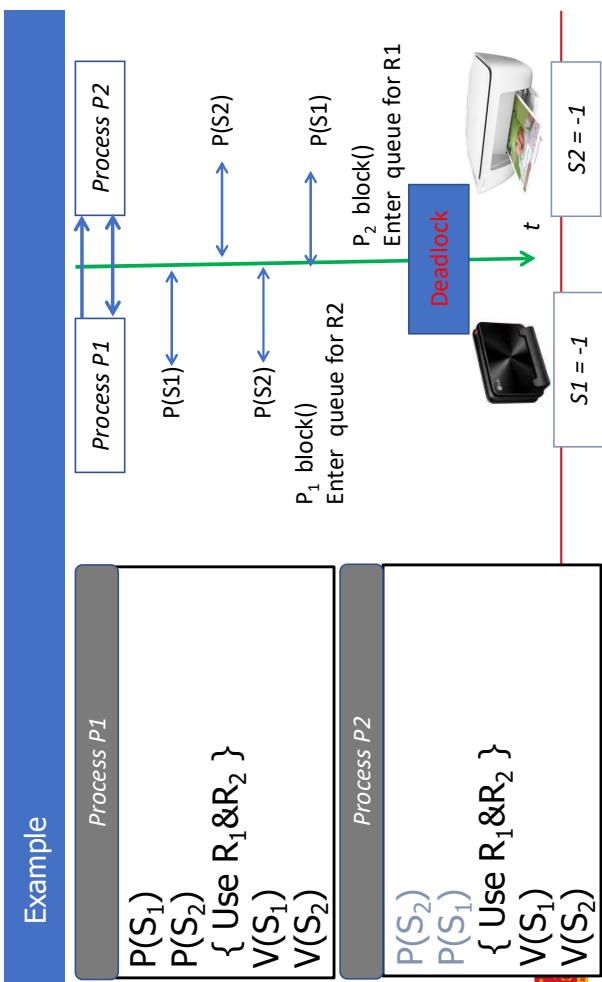


**Chapter 2 Process Management  
5.Dead lock and solutions**

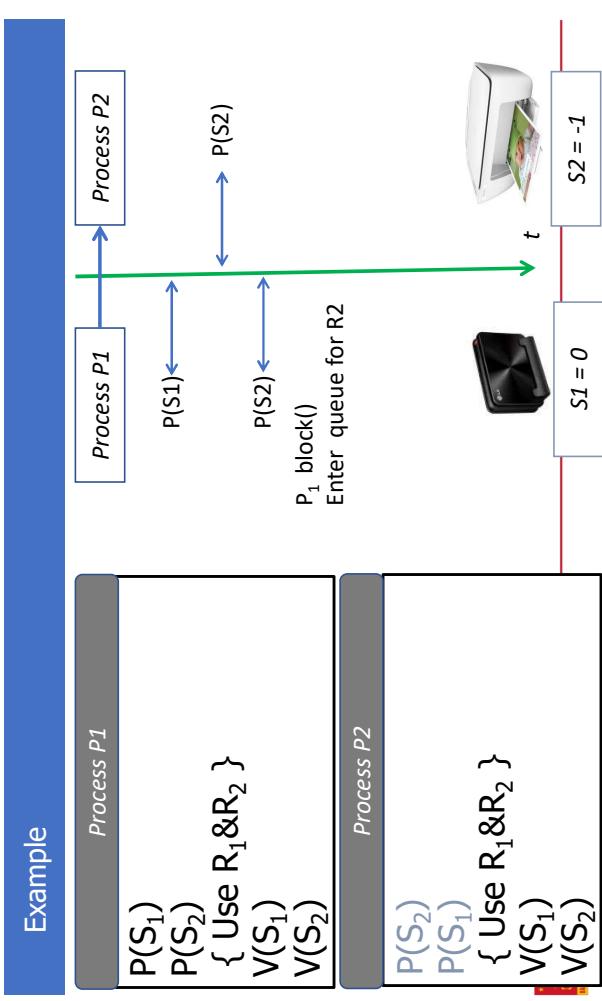
**Chapter 2 Process Management**  
**5.Deadlock and solutions**



**Chapter 2 Process Management  
5.Dead lock and solutions**



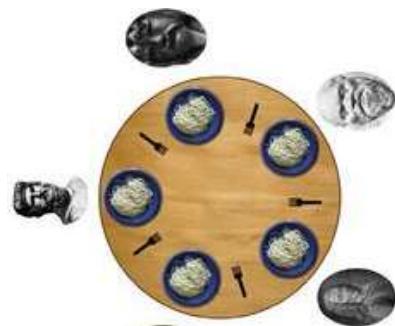
**Chapter 2 Process Management  
5.Dead lock and solutions**



## Conditions

- 4 conditions, must occur at the same time
  - **Critical resource**
    - Resource is used in a non-shareable model
    - Only 1 process can use resource at a time
    - Other process request to use resource  $\Rightarrow$  request must be postponed until resource is released
  - **Wait before enter the critical section**
    - Process can not enter critical section has to wait in queue
    - Still own resources while waiting
  - **No resource reallocation system**
    - Resource is non-preemptive
    - Resource is released only by currently using process after this process finished its task
  - **Circular waiting**
    - Set of processes  $\{P_0, P_1, \dots, P_n\}$  waiting in a order:  $P_0 \rightarrow R_1 \rightarrow P_1; P_1 \rightarrow R_2 \rightarrow P_2; \dots; P_{n-1} \rightarrow R_n \rightarrow P_n; P_n \rightarrow R_0 \rightarrow P_0$
    - Circular waiting create nonstop loop

## Example: Dining philosopher problem



- Critical resource
- Wait before enter critical section
- Non-preemptive resource
- Circular waiting

## Definition

A set of processes is **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause

- Deadlock conception
- **Conditions for resource deadlocks**
- Solutions for deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

## Methods

### Prevention

- Apply methods to guarantee that the system **never** has deadlock
  - Expensive
  - Apply for system that deadlock happens **frequently** and once it happen the **cost is high**
- Avoidance**
- Check each process's resource request and reject request if this request may lead to deadlock
  - Require extra information
  - Apply for system that deadlock happens **least frequently**
  - and once it happen the **cost is high**



## Methods

### Deadlock detection and recovery

- Allow the system work normally  $\Rightarrow$  deadlock may happen
- **Periodically check** if deadlock is happening
- If deadlock apply methods to remove deadlock
- Apply for system that deadlock happens **least frequently** and once it happen the **cost is low**



## Methods

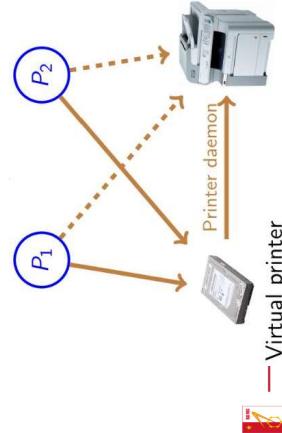
### Prevention

- Prevention
- Avoidance
- Detection and recovery



## Critical resource condition

- Reduce the system's critical degree
  - Shareable resource(read-only file): accessed simultaneously
  - Non-shareable resource: **Cannot** be accessed simultaneously
- SPOOL mechanism(*Simultaneous peripheral operation on-line*)
  - Do not allocate** resource **when it's not necessary**
  - A limited number of processes **can request** resource



## Wait before entering critical section condition

**Rule:** Make sure 1 process **request** resource only when it **doesn't own** any other resources

### Prior allocate

- processes request **all** their resources **before** starting execution and only **run** when required resources are **allocated**
  - Effectiveness** of resource utilization is **low**
  - Process only use resource at the last cycle?
  - Total requested resource higher than the system's capability?
- Resource release**
  - Process **releases** all resource **before** apply(re-apply) new resource
  - Process execution's **speed** is **low**
  - Must guarantee that **data** kept in temporary release resource **won't be lost**



## Rule

Attack 1 of 4 required conditions for deadlock to appear

### Critical resource

Wait before entering critical section

Non-preemptive resource

Circular wait



## Rule

Attack 1 of 4 required conditions for deadlock to appear

### Critical resource

Wait before entering critical section

Non-preemptive resource

Circular wait



## No preemption resource condition

**Rule:** allow process to **preempt** resource **when it's necessary**

- **Process**  $P_i$  apply for **resource**  $R_j$ 
  - $R_j$  is available: allocate  $R_j$  to  $P_i$
  - $R_j$  not available: ( $R_j$  is being used by process  $P_k$ )
    - $P_k$  is waiting for another resource
    - Preempt  $R_j$  from  $P_k$  and allocate to  $P_i$  as requested
    - Add  $R_j$  into the list of needing resource of  $P_k$
    - $P_k$  is execution again when
      - Receive the needing resource
      - Take back resource  $R_j$
  - $P_k$  is running
    - $P_i$  must wait (*no resource release*)
    - Allow resource preempt only when **it's necessary**

## No preemption resource condition

**Rule:** allow preemptive when it's necessary

- Only applied for resources that can be store and recover easily
  - (*CPU, memory space*)
  - Difficult to apply for resource like **printer**
- 1 process is preempted many time?



## Wait before entering critical section condition - Example

- Process combines of 2 **phases**
  - Copy data from **tape** to a file in the **disk**
  - Arrange the data in **file** and bring to **printer**
- Prior allocation method
  - Request both tape, file and printer
  - **Waste** printer in first phase, tape in the second phase

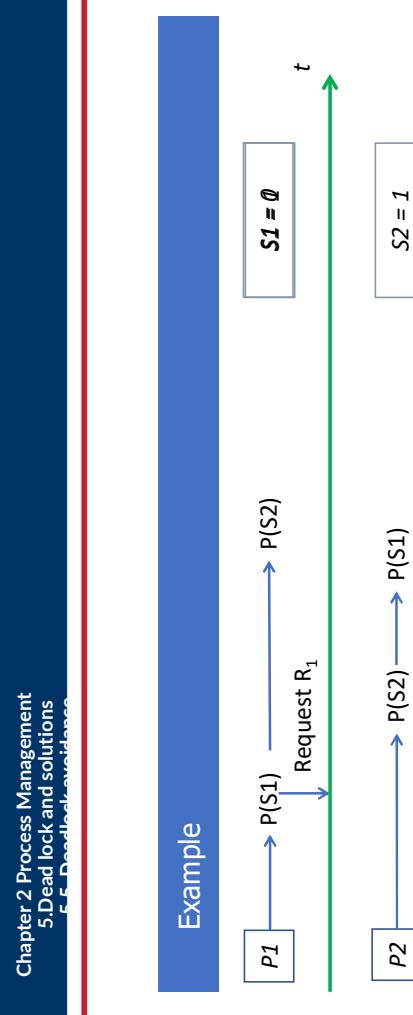


## Wait before entering critical section condition - Example

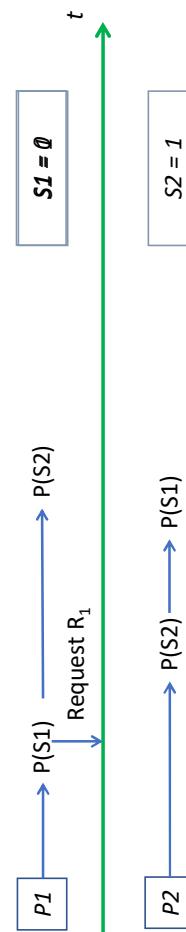
- Process combines of 2 **phases**
  - Copy data from **tape** to a file in the **disk**
  - Arrange the data in **file** and bring to **printer**
- Prior allocation method
  - Request both tape, file and printer
  - **Waste** printer in first phase, tape in the second phase
- **Resource release method**
  - Request tape and file for phase 1
  - Release tape and file
  - **Waste** printer in first phase, tape in the second phase
- **Resource release method**
  - Request tape and file for phase 1
  - Release tape and file
  - **Waste** printer in first phase, tape in the second phase



- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- **Deadlock avoidance**
- Deadlock detection and recovery



### Example



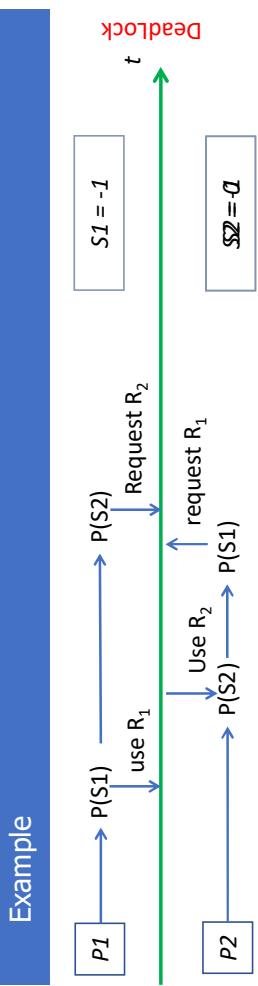
### Circular wait condition

- Provide a global numbering of all type of resources
  - $R = \{R_1, R_2, \dots, R_n\}$  Set of resources
  - Construct an ordering function  $f: R \rightarrow N$ 
    - Function f is constructed based on the order of resource utilization
- $f(\text{Tape}) = 1$ 
  - $f(\text{Disk}) = 5$
  - $f(\text{Printer}) = 12$
- Process can only request resource in an increasing order
  - holding resource type  $R_k$  can only request resource type  $R_j$  satisfy  $f(R_j) > f(R_k)$
  - Process requests resource  $R_k$  has to release all resource  $R_i$  satisfy condition  $f(R_i) \geq f(R_k)$

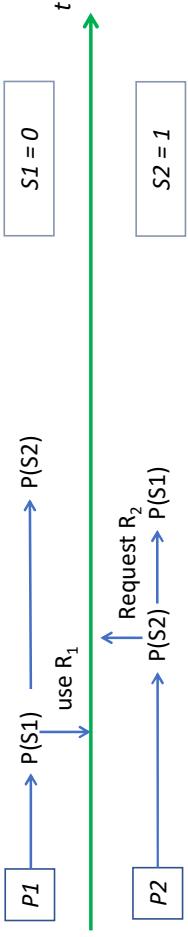
### Circular wait condition

- Process can only request resource in an increasing order
  - Process holding resource type  $R_k$  can only request resource type  $R_j$  satisfy  $f(R_j) > f(R_k)$
  - Process requests resource  $R_k$  has to release all resource  $R_i$  satisfy condition  $f(R_i) \geq f(R_k)$
- **Prove**
  - Suppose deadlock happen between processes  $\{P_1, P_2, \dots, P_m\}$ 
    - $R_1 \rightarrow P_1 \rightarrow R_2 \rightarrow P_2 \Rightarrow f(R_1) < f(R_2)$
    - $R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \Rightarrow f(R_2) < f(R_3) \dots$
    - $R_m \rightarrow P_m \rightarrow R_1 \rightarrow P_1 \Rightarrow f(R_m) < f(R_1)$
    - $f(R_1) < f(R_2) < \dots < f(R_m) < f(R_1) \Rightarrow \text{invalid}$





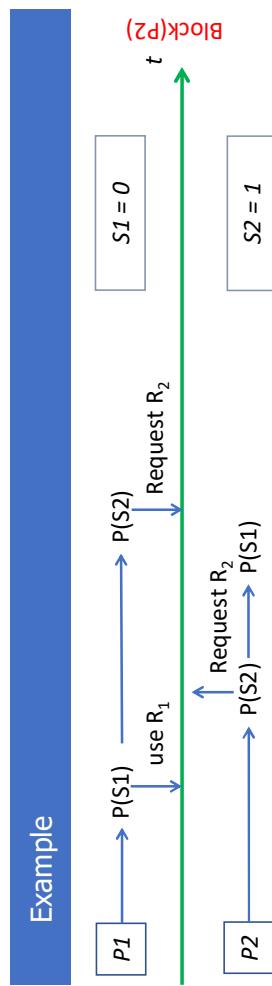
Example



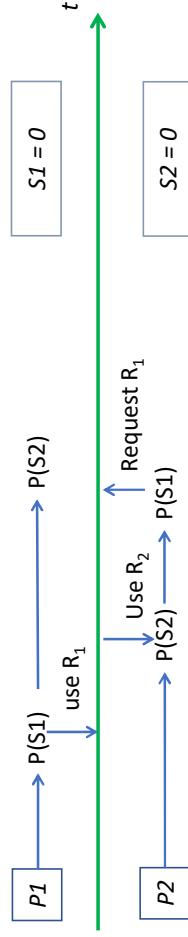
Example



Chapter 2 Process Management  
5. Dead lock and solutions  
5.5. Deadlock avoidance



Example



Example

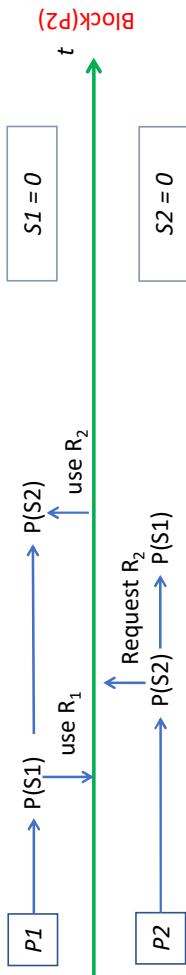
## Safe state

The system's state is safe when

- It's possible to provide resource for each process (to maximum requirement) follow an order such that deadlock will not occur

- A **safe sequence** of all processes is existed

## Example



Conclude:

If processes' orders of request/release resources are known in advance, the system could make a resource allocation decision (accept or block) for all request to let deadlock not occur.

## Safe sequence

**Set of process  $P = \{P_1, P_2, \dots, P_n\}$  is safe if**

- For each process  $P_i$ , each resource request in future is accepted due to
  - The amount of available resource in the system
  - Resource is currently occupied by process  $P_j (j < i)$
  - In **safe sequence**, when  $P_i$  request for resource
    - If **not accept** immediately,  $P_i$  wait until  $P_j$  terminates ( $j < i$ )
    - When  $P_j$  terminate and **release resource**,  $P_i$  receives required resource, executes, **releases allocated resource** and terminate
    - When  $P_j$  terminate and release resource  $\Rightarrow P_{i+1}$  will receive resource and able to finish.
    - All the processes in the safe sequence is able to finish
- Must **know in advance** information of processes and resources
  - Process has to declare the **maximum amount** of **each resources type** that is **required for execution**
  - **Decision** is made based on the result of **Resource-Allocation State check**
    - Resource allocation state is defined by following parameters
      - Number of resource unit **available** in the system
      - Number of resource unit **allocated** for each process
      - Number of **maximum** resource unit each process may **request**
      - If system is **safe**, request is **accepted**
    - Perform checking every time a resource request is received
      - Objective: Guarantee the system's always in safe state
        - At the beginning (resource is not allocated), system is safe
        - Only allocate resource when safety is guaranteed
        - $\Rightarrow$  System changes from current safety state to another safety state

## Rule

- Must **know in advance** information of processes and resources
  - Process has to declare the **maximum amount** of **each resources type** that is **required for execution**
  - **Decision** is made based on the result of **Resource-Allocation State check**
    - Resource allocation state is defined by following parameters
      - Number of resource unit **available** in the system
      - Number of resource unit **allocated** for each process
      - Number of **maximum** resource unit each process may **request**
      - If system is **safe**, request is **accepted**
    - Perform checking every time a resource request is received
      - Objective: Guarantee the system's always in safe state
        - At the beginning (resource is not allocated), system is safe
        - Only allocate resource when safety is guaranteed
        - $\Rightarrow$  System changes from current safety state to another safety state

## Example

- Conclude
  - System is safe  $\Rightarrow$  Processes are able to finish  
 $\Rightarrow$  no deadlock
  - System is not safe  $\Rightarrow$  Deadlock may occur
- Method
  - Verify all resource request
    - If the system is still safe after allocate resource  $\Rightarrow$  allocate
    - If not  $\Rightarrow$  process has to wait
  - The banker algorithm



## Example

- Consider a system includes
  - 3 processes  $P_1, P_2, P_3$  and one resource R has **12 units**
  - $(P_1, P_2, P_3)$  may **request maximum**  $(10, 4, 9)$  unit of resource R
  - At time  $t_0$ ,  $(P_1, P_2, P_3)$  **allocated**  $(5, 2, 2)$  unit of resource R
  - At current time  $(t_0)$ , is the system safe?
  - System allocate  $(5 + 2 + 2)$  units  $=>$  3 units remain
  - $(P_1, P_2, P_3)$  may request  $(5, 2, 7)$  units
  - With current 3 units, all request of  $P_2$  is acceptable  $\Rightarrow P_2$  guaranteed to **finish and will release** 2 unit of R after finished
  - With  $3 + 2$  units,  $P_1$  guaranteed to finish and will release 5 units
  - With  $3 + 2 + 5$  unit  $P_3$  guaranteed to finish
  - At time  $t_0$ ,  $P_1, P_2, P_3$  are guaranteed to finish  $\Rightarrow$  system is safe with **safe sequence**  $(P_2, P_1, P_3)$



## The banker algorithm: Introduction

- Good for systems have resources with many units
- New appearing process has to declare the maximum unit of each resource type
- Not larger than the total amount of the system
- When one process request for resource, system verify if it's safe to accept this requirement
- If system still safe  $\Rightarrow$  Allocate resource for process
- Not safe  $\Rightarrow$  wait



## Example

- Consider a system includes
  - 3 processes  $P_1, P_2, P_3$  and one resource R has **12 units**
  - $(P_1, P_2, P_3)$  may **request maximum**  $(10, 4, 9)$  unit of resource R
  - At time  $t_0$ ,  $(P_1, P_2, P_3)$  **allocated**  $(5, 2, 2)$  unit of resource R
  - A time  $t_1$ ,  $P_3$  request and allocated 1 resource unit. Is the system safe?
  - With current 2 units, all request of P2 is acceptable  $\Rightarrow$  P2 guaranteed to finish and will release 2 unit of R after finished
  - When  $P_2$  finished, the amount of **available resource** in the system is **4**
  - With **4** resource unit,  $P_1$  and  $P_3$  both have to wait when apply for **5** more resource unit
  - Hence, system is not safe with  $(P_1, P_3)$
  - Remark: At time  $t_1$ , if  $P_3$  **has to wait** when request for 1 more resource unit, deadlock will be removed



## Example

- Consider a system includes
  - 3 processes  $P_1, P_2, P_3$  and one resource R has **12 units**
  - $(P_1, P_2, P_3)$  may **request maximum**  $(10, 4, 9)$  unit of resource R
  - At time  $t_0$ ,  $(P_1, P_2, P_3)$  **allocated**  $(5, 2, 2)$  unit of resource R
  - A time  $t_1$ ,  $P_3$  request and allocated 1 resource unit. Is the system safe?
  - With current 2 units, all request of P2 is acceptable  $\Rightarrow$  P2 guaranteed to finish and will release 2 unit of R after finished
  - When  $P_2$  finished, the amount of **available resource** in the system is **4**
  - With **4** resource unit,  $P_1$  and  $P_3$  both have to wait when apply for **5** more resource unit
  - Hence, system is not safe with  $(P_1, P_3)$
  - Remark: At time  $t_1$ , if  $P_3$  **has to wait** when request for 1 more resource unit, deadlock will be removed



### Algorithm for safety checking

```
BOOL Safe( Current Resource-Allocation State){  
    Work<-Available  
    for ( i : 1 → n) Finish[i]←false  
    flag← true  
    While(flag){  
        flag← false  
        for ( i : 1 → n)  
            if(Finish[i]=false AND Need[i] ≤Work){  
                Finish[i]← true  
                Work ← Work+Allocation[i]  
                flag← true  
            } //endif  
        } //endwhile  
        for ( i : 1 → n) if (Finish[i]=false) return false  
    return true;  
} //End function
```

### Algorithm's data I

System       $n$  number of process in the system  
 $m$  number of resource type in the system

Data structures

- Available** Vector with length  $m$  represents the number of available resource in the system. ( $\text{Available}[3] = 8 \Rightarrow ?$ )
- Max** Matrix  $n * m$  represents each process maximums request for each type of resource. ( $\text{Max}[2,3] = 5 \Rightarrow ?$ )
- Allocation** Matrix  $n * m$  represents amount of resource allocated for processes ( $\text{Allocation}[2,3] = 2 \Rightarrow ?$ )
- Need** Matrix  $n * m$  represents amount of resource is needed for each process ( $\text{Need}[2,3] = 3 \Rightarrow ?$ )

$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$



### Example

- Consider system with 5 processes  $P_0, P_1, P_2, P_3, P_4$  and 3 resources  $R_0, R_1, R_2$
- $R_0$  has 10 units,  $R_1$  has 5 units,  $R_2$  has 7 units
- Maximum resource requirement and allocated resource for each process

	R0	R1	R2	
P0	7	5	3	P0
P1	3	2	2	P1
P2	9	0	2	P2
P3	2	2	2	P3
P4	4	3	3	P4
				Allocation
				Max

### Algorithm's data I

Chapter 2 Process Management  
5.Dead lock and solutions  
5.5. Deadlock avoidance

Rule

- $X, Y$  are vectors with length  $n$
- $X \leq Y \Leftrightarrow X[i] \leq Y[i] \forall i = 1, 2, \dots, n$
- Each row of  $\text{Max}, \text{Need}, \text{Allocation}$  is processed like vector
- Algorithm calculated on vectors

Local structures

- Work** vector with length  $m$  represents how much each resource still available
- Finish** vector with Boolean type, length  $n$  represents if a process is guaranteed to finish or not

$\text{Work}$



Is the system safe?

- $P_1$  requests 1 unit of  $R_0$  and 2 unit of  $R_2$ ?
- $P_4$  requests 3 unit of  $R_0$  and 3 unit of  $R_1$ ?
- $P_0$  requests 2 unit of  $R_1$ . allocate

### Example: Check for safety

Number of available resource in the system Available(R<sub>0</sub>, R<sub>1</sub>, R<sub>2</sub>) = (3, 3, 2)

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>
Allocation	T	T	T	T	T
Max	4	3	3	0	0
Need	0	0	2	1	1

System is safe  
(P<sub>1</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>0</sub>, P<sub>2</sub>)



### Example: Check for safety

- Number of available resource in the system Available(R<sub>0</sub>, R<sub>1</sub>, R<sub>2</sub>) = (3, 3, 2)
- Remaining request of each process (Need = Max - Allocation)

	R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>			R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>			R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>		
	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>0</sub>	P <sub>1</sub>		
P <sub>0</sub>	7	5	3	0	1	0	7	4	3	0	1	0	7	4	3
P <sub>1</sub>	3	2	2	2	0	0	1	2	2	2	0	0	1	2	2
P <sub>2</sub>	9	0	2	3	0	2	6	0	0	3	0	2	3	0	2
P <sub>3</sub>	2	2	2	2	1	1	0	1	1	2	0	0	2	0	0
P <sub>4</sub>	4	3	3	0	0	2	4	3	1	2	1	1	0	1	1
Max	4	3	3	0	0	2	4	3	3	0	0	2	4	3	1
Need	0	0	2	1	1	1	0	0	2	1	1	0	0	2	1



### Example: Check for safety

- Number of available resource in the system Available(R<sub>0</sub>, R<sub>1</sub>, R<sub>2</sub>) = (3, 3, 2)
- Remaining request of each process (Need = Max - Allocation)

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>		R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>		R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>
Allocation	T	T	T	T	T	Allocation	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	Allocation	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Work						( <del>TTTTTT</del> )Y							
Finish													



Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>		R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>		R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>
Allocation	T	T	T	T	T	Allocation	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	Allocation	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Work						( <del>TTTTTT</del> )Y							
Finish													



### Example: Check for safety

- Number of available resource in the system Available(R<sub>0</sub>, R<sub>1</sub>, R<sub>2</sub>) = (3, 3, 2)
- Remaining request of each process (Need = Max - Allocation)

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>		R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>		R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>
Allocation	F	T	T	T	F	Allocation	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	Allocation	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Work						( <del>TTTTTT</del> )Y							
Finish													



Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>		R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>		R <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>
Allocation	F	T	T	T	F	Allocation	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	Allocation	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Work						( <del>TTTTTT</del> )Y							
Finish													



### Algorithm for resource request

- Request[i] Resource requesting vector of process P<sub>i</sub>

- Request[3,2] = 2: P<sub>3</sub> requests 2 units of resource R<sub>2</sub>

- When Pi make a resource request, the system checks

① if(Request[i] > Need[i])

**Error**(Request higher than declared number)

② if(Request[i] > Available)

**Block**(Not enough resource, process has to wait)

③ Set the new resource allocation for the system

- Available = Available - Request[i]

- Allocation[i] = Allocation[i] + Request[i]

- Need[i] = Need[i] - Request[i]

④ Allocate resource based on the result of new system safety check

**if**(Safe(New Resource Allocation State))

        Allocate resource for Pi as requested

**else**  
        Pi has to wait  
        Recover former state (Available, Allocation,Need)



### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	F	F	F	F	F
Work					(5,3,2)		

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	F	F	F	F	F
Work					(5,3,2)		

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	F	F	F	F	F
Work					(2,3,0)		



### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	F	F	F	F	F
Work					(5,3,2)		

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	F	F	F	F	F
Work					(2,3,0)		



### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	F	F	F	F	F
Work					(5,3,2)		



### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	F	F	F	F	F
Work					(2,3,0)		



### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	F	F	F	F	F
Work					(2,3,0)		



### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	T	T	F	T	T	F	T
Work					(7,5,5)		

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	T	T	F	T	T	F	T
Work					(7,5,5)		

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	T	F	T	F	T
Work					(7,4,3)		

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	T	F	T	F	T
Work					(7,4,3)		



### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	T	T	F	T	T	F	T
Work					(10,5,7)		

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	T	F	T	F	T
Work					(7,4,5)		



### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	T	F	T	F	T
Work					(7,4,5)		

Request is accepted

### Example: P<sub>1</sub> request (1, 0, 2)

- Request[1] ≤ Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2, 3, 0)

	R0	R1	R2		R0	R1	R2
Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>
Finish	F	T	T	F	T	F	T
Work					(7,4,5)		



## Introduction

- Do not apply deadlock prevention or avoidance method, allow deadlock to occur
- Timely check if the system has deadlock or not if yes then find a solution
- To function properly, system has to provide
  - Algorithm to check if the system is deadlock or not
  - Algorithm to recover from deadlock
- Deadlock detection
  - Algorithm for showing the deadlock
  - Deadlock recovery
  - Terminate process
  - Resource preemptive



## Algorithm to point out deadlock: Introduction

- Apply for system that has resource types with many units
- Similar to [banker algorithm](#)
- Data structures
  - **Available** Vector with length m: Available resource in the system
  - **Allocation** Matrix n \* m: Resources allocated to processes
  - **Request** Matrix n \* m: Resources requested by processes
- Local structures
  - **Work** Vector with length m: available resource
  - **Finish** Vector with length n: process is able to finish or not
- Rule
  - $\leq$  relations between Vectors
  - Rows in matrix n \* m are processed similar to vectors



## Example: (continue)

- $P_4$  request 3 units of  $R_0$  and 3 units of  $R_2$ 
  - Request[4] = (3, 0, 3)
  - Available = (2, 3, 0)
  - ⇒ Resource is not enough,  $P_4$  has to wait
- $P_0$  request 2 units of  $R_1$ 
  - Request[0] ≤ Available ((0, 2, 0) ≤ (2, 3, 0)) ⇒ It's possible to allocate
  - If allocate: Available = (2, 1, 0)
  - Perform Safe algorithm
    - ⇒ All processes may not finish
    - ⇒ if accepted, system may be unsafe
    - ⇒ Resource is sufficient but do not allocate,  $P_0$  has to wait



- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
  - Deadlock prevention
  - Deadlock avoidance
- **Deadlock detection and recovery**



### Example

	R0	R1	R2		R0	R1	R2
Allocation	0	1	0		0	0	0
Request	2	0	0		2	0	2
P0	0	1	0		P0	0	0
P1	2	0	0		P1	2	0
P2	3	0	3		P2	0	0
P3	2	1	1		P3	1	0
P4	0	0	2		P4	6	0

● Available resource ( $R_0, R_1, R_2$ ) = (0, 0, 0)

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>
Finish	F	F	F	F	F
Work			(0,0,0)		

### Algorithm to point out deadlock

```

BOOL Deadlock(Current Resource-Allocation State){
    Work<-Available
    for (i : 1 → n)
        if(Allocation[i]≠0) Finish[i]←false
        else Finish[i]←true; //Allocation = 0 not in waiting circular
    flag← true
    While(flag){
        flag←false
        for (i : 1 → n)
            if (Finish[i] = false AND Request[i] ≤ Work){
                Finish[i]← true
                Work ← Work+Allocation[i]
                flag← true
            }
        } //Endwhile
        for (i : 1 → n) if (Finish[i] = false) return true;
        return false; //Finish[i] = false, process Pi is in deadlock
    } //End function

```

### Example

	R0	R1	R2		R0	R1	R2
Allocation	0	1	0		0	0	0
Request	2	0	0		2	0	2
P0	0	1	0		P0	0	0
P1	2	0	0		P1	2	0
P2	3	0	3		P2	3	0
P3	2	1	1		P3	2	1
P4	0	0	2		P4	6	0

● Available resource ( $R_0, R_1, R_2$ ) = (0, 0, 0)

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>		R0	R1	R2
Finish	T	F	F	F	F		P0	0	0
Work			(0,1,0)				P1	2	0

### Example

- 5 processes P<sub>0</sub>, P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>; 3 resources R<sub>0</sub>, R<sub>1</sub>, R<sub>2</sub>
- Resource R<sub>0</sub> has 7 units, R<sub>1</sub> has 2 units, R<sub>2</sub> has 6 units
- Resource allocation status at time t<sub>0</sub>

	R0	R1	R2		R0	R1	R2
Allocation	0	1	0		0	0	0
Request	2	0	0		2	0	2
P0	0	1	0		P0	0	0
P1	2	0	0		P1	2	0
P2	3	0	3		P2	3	0
P3	2	1	1		P3	2	1
P4	0	0	2		P4	6	0

- Available resource ( $R_0, R_1, R_2$ ) = (0, 0, 0)



### Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Allocation



● Available resource ( $R_0, R_1, R_2$ ) = (0, 0, 0)

Process	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$
Finish	T	F	T	T	F
Work		(5,2,4)			

Request

### Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Allocation



● Available resource ( $R_0, R_1, R_2$ ) = (0, 0, 0)

Process	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$
Finish	T	T	T	T	F
Work		(7,2,4)			

Request



### Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Allocation



● Available resource ( $R_0, R_1, R_2$ ) = (0, 0, 0)

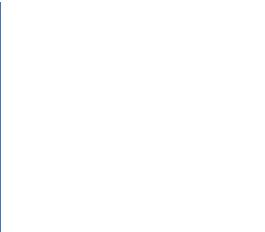
Process	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$
Finish	T	F	T	T	F
Work		(3,1,3)			

Request

### Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Allocation



● Available resource ( $R_0, R_1, R_2$ ) = (0, 0, 0)

Process	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$
Finish	T	F	T	T	F
Work		(5,2,4)			

Request

### Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Allocation



● Available resource ( $R_0, R_1, R_2$ ) = (0, 0, 0)

Process	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$
Finish	T	F	T	T	F
Work		(5,2,4)			

Request

### Example

	R0	R1	R2		R0	R1	R2
P0	0	1	0		0	0	0
P1	2	0	0		2	0	0
P2	3	0	3		3	0	2
P3	2	1	1		2	1	1
P4	0	0	2		0	0	2

Allocation      Request

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>
Finish		F	F	F	F
Work		(0,0,0)			



### Example

	R0	R1	R2		R0	R1	R2
P0	0	1	0		0	1	0
P1	2	0	0		2	0	0
P2	3	0	3		3	0	2
P3	2	1	1		2	1	1
P4	0	0	2		0	0	2

Allocation      Request

- Available resource (R<sub>0</sub>, R<sub>1</sub>, R<sub>2</sub>) = (0, 0, 0)

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>
Finish		F	F	F	F
Work		(7,2,6)			



System has no deadlock  
(P<sub>0</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>1</sub>, P<sub>4</sub>)

### Example

	R0	R1	R2		R0	R1	R2
P0	0	1	0		0	1	0
P1	2	0	0		2	0	0
P2	0	0	1		3	0	3
P3	1	0	0		2	1	1
P4	6	0	2		0	0	2

Allocation      Request

- At time t<sub>1</sub>: P<sub>2</sub> request 1 more resource unit of R<sub>2</sub>
- Resource allocation status

	R0	R1	R2		R0	R1	R2
P0	0	0	0		0	0	0
P1	2	0	0		2	0	0
P2	3	0	3		3	0	3
P3	2	1	1		2	1	1
P4	0	0	2		0	0	2

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>
Finish		T	F	F	F
Work		(0,1,0)			



- Available resource (R<sub>0</sub>, R<sub>1</sub>, R<sub>2</sub>) = (0, 0, 0)



## Deadlock recovery: Resource preemption method

Preempt continuously several resources from a deadlocked processes and give to other processes until deadlock is removed

Need to consider:

- ① Victim's selection
  - Which resource and which process is selected?
  - Preemption's order for smallest cost
  - Amount of holding resource, usage time . . .

### ② Rollback

- Rollback to a safe state before and restart
- Require to store state of running process

### ③ Starvation

- One process is preempted many times  $\Rightarrow$  infinite waiting
- Solution: record the number of times that process is preempted

## Example

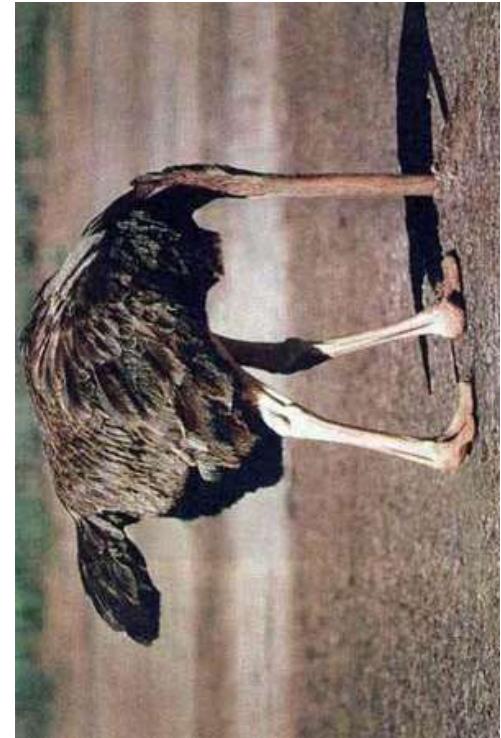
	R0	R1	R2		R0	R1	R2
P0	0	1	0		P0	0	0
P1	2	0	0		P1	2	0
P2	3	0	2		P2	0	0
P3	2	1	1		P3	1	0
P4	0	0	2		P4	6	0
				Allocation			
				Request			

Process	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>
Finish	T	F	F	F	F
Work	(0,1,0)				

- P<sub>0</sub> may finish but the system is deadlock.
- Processes are waiting for each other (P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>)



## Another solution ?



## Deadlock recovery: Process termination method

Rule: Terminate processes in deadlock and take back allocated resource

- Terminate all processes
  - Quick to eliminate deadlock
  - Too expensive
- Killed processes may be almost finished
- Terminate processes consequently until deadlock is removed
  - After process is terminated, check if deadlock is still exist or not
  - Deadlock checking algorithm complexity is  $m * n^2$
  - Need to point out the order of process to be terminated
  - Process's priority
  - Process's turn around time, how long until process finish
  - Resources that process is holding, need to finish
  - . . .
- Process termination's problem
  - Process is updating file  $\Rightarrow$  File is not complete
  - Process is using printer  $\Rightarrow$  Reset printer's status



# Hệ Điều Hành

(Nguyên lý các hệ điều hành)

Đỗ Quốc Huy  
huydq@soict.hust.edu.vn  
Bộ môn Khoa Học Máy Tính  
Viện Công Nghệ Thông Tin và Truyền Thông

## Chap 3 Memory Management

- ① Introduction
- ② Memory management strategies
- ③ Virtual memory
- ④ Memory management in Intel's processors family

### Chapter Summary

- Deadlock is the situation when 2 or more processes are independently waiting for an event that can only be caused by these processes
- Deadlock occurs when the 4 following condition exist
  - Critical resource
  - Wait before entering the critical region
  - No resource-reallocation system
  - Circular waiting
- 3 Approaches to handle Deadlock
  - Prevention
  - Attack the deadlock's happening conditions
  - Avoidance
  - Make the system avoid situations that may cause the deadlock to happen
  - Detection and handling
  - Allow deadlock to happen, indicate deadlock and handle later

- 5 tiến trình  $P_0, P_1, P_2, P_3, P_4$ ; 3 tài nguyên  $R_0, R_1, R_2$
- Tài nguyên  $R_0$  có 6 đơn vị,  $R_1$  có 4 đơn vị,  $R_2$  có 7 đơn vị
- Trạng thái cung cấp tài nguyên tại thời điểm  $t_0$

	R0	R1	R2		R0	R1	R2	
P0	0	1	1		P0	0	0	0
P1	1	0	0		P1	2	1	2
P2	3	0	2		P2	0	0	2
P3	2	1	1		P3	1	0	0
P4	0	2	2		P4	6	0	2
								Request

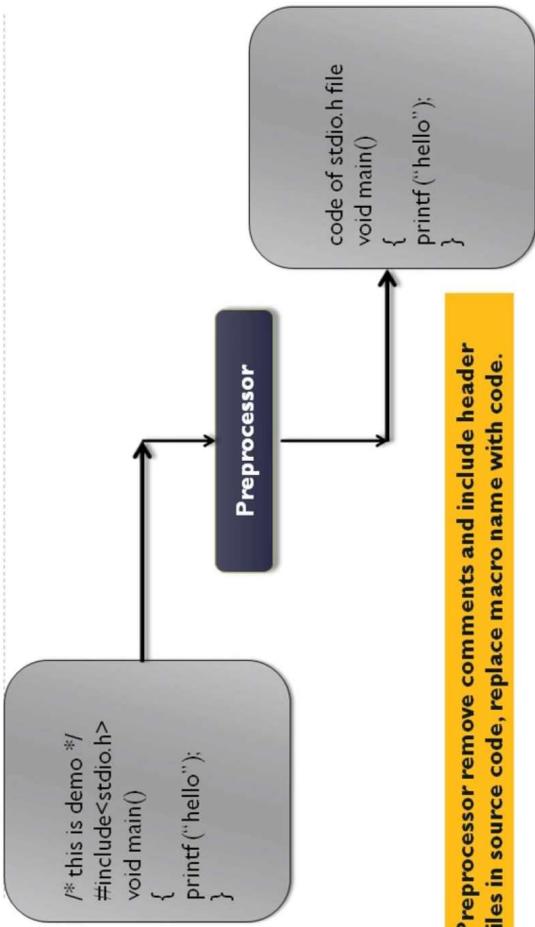


## Chapter 3 Memory management

### 1. Introduction

#### 1.1 Example

## Preprocessor



## Chapter 3 Memory management

### 1. Introduction

#### ● Example

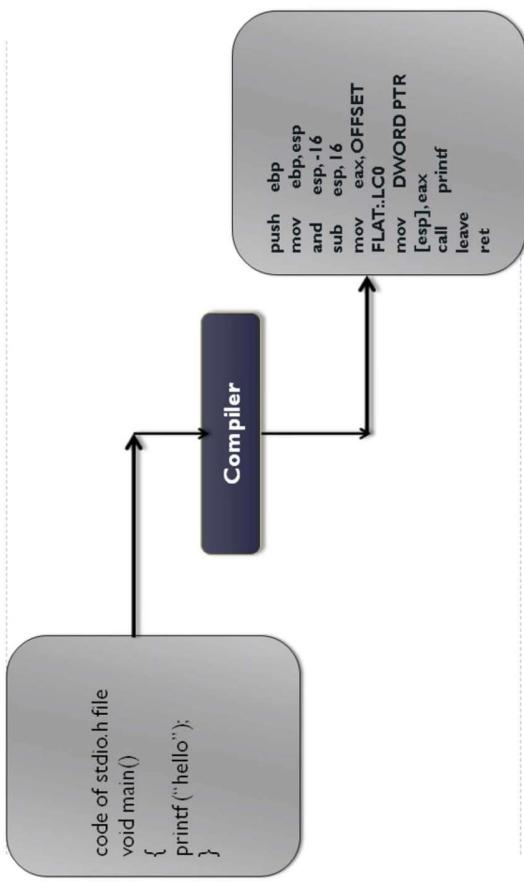
- Memory and program
- Address binding
- Program's structures

## Chapter 3 Memory management

### 1. Introduction

#### 1.1 Example

## C compilation process

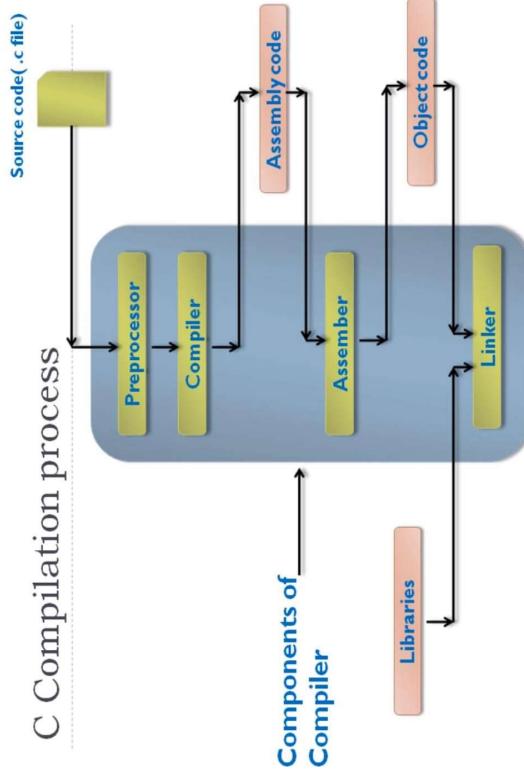


## Chapter 3 Memory management

### 1. Introduction

#### 1.1 Example

## C compilation process



## Chapter 3 Memory management

### 1. Introduction

#### 1.1 Example

Example: Generate programs from multi modules

### Toto project

```
file main.c
#include <stdio.h>
extern int x, y;
extern void toto();
int main(int argc, char *argv[]) {
    toto();
    printf("KQ: %d\n", x * y);
    return 0;
}
```

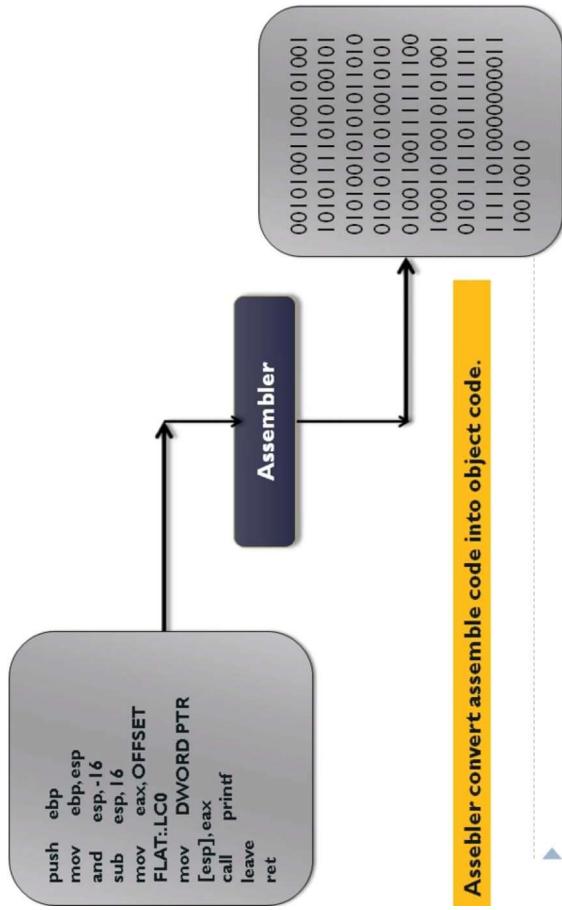
Result  
KQ: 1000

## Chapter 3 Memory management

### 1. Introduction

#### 1.1 Example

C compilation process

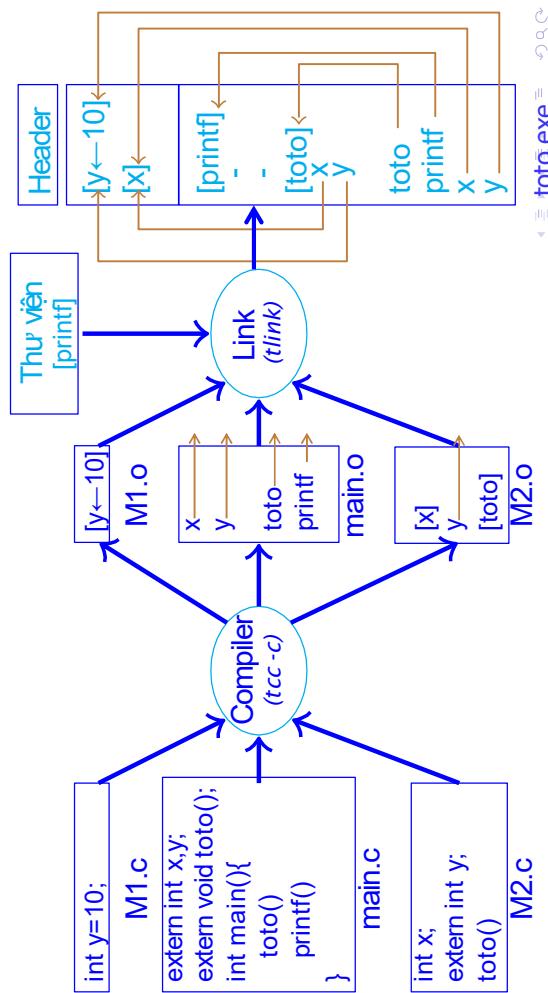


## Chapter 3 Memory management

### 1. Introduction

#### 1.1 Example

toto project compilation process

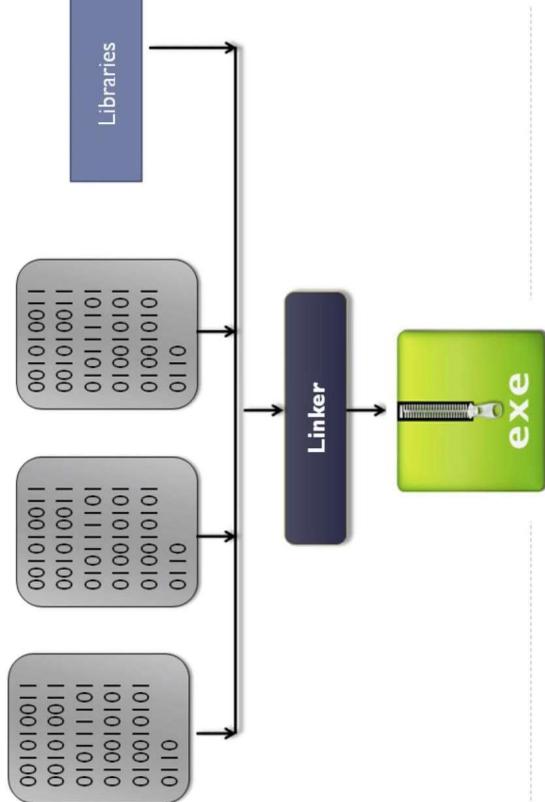


## Chapter 3 Memory management

### 1. Introduction

#### 1.1 Example

C compilation process

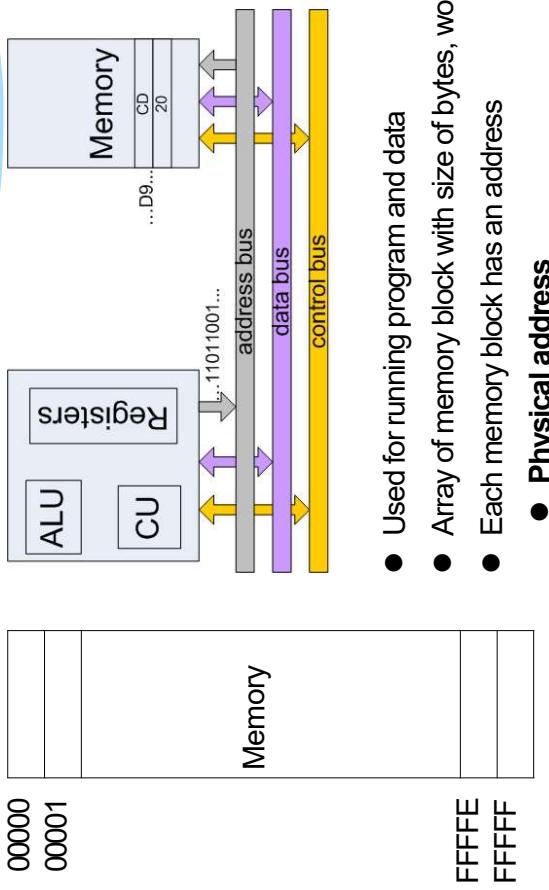


## Chapter 3 Memory management

### 1. Introduction

#### 1.2. Memory and program

##### Main memory



## Chapter 3 Memory management

### 1. Introduction

#### 1.2. Memory and program

- Example
- **Memory and program**
- Address binding
- Program's structures

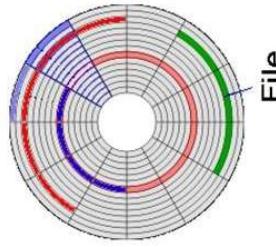
## Chapter 3 Memory management

### 1. Introduction

#### 1.2. Memory and program

##### Program

- Store on external storage devices
- Executable binary files
- File's parameters
- Machine instruction (binary code), . . .
- Data area (global variable), . . .
- Must be brought into internal memory and put inside a process to be executed (process executes program)
- Input queue
- Set of processes kept in external memory (*normally: disk*)
- Wait to be brought into internal memory and execute



01010110011110010101010101101

Header	Data	Code
--------	------	------

## Chapter 3 Memory management

### 1. Introduction

#### 1.2. Memory and program

##### Memory leveling

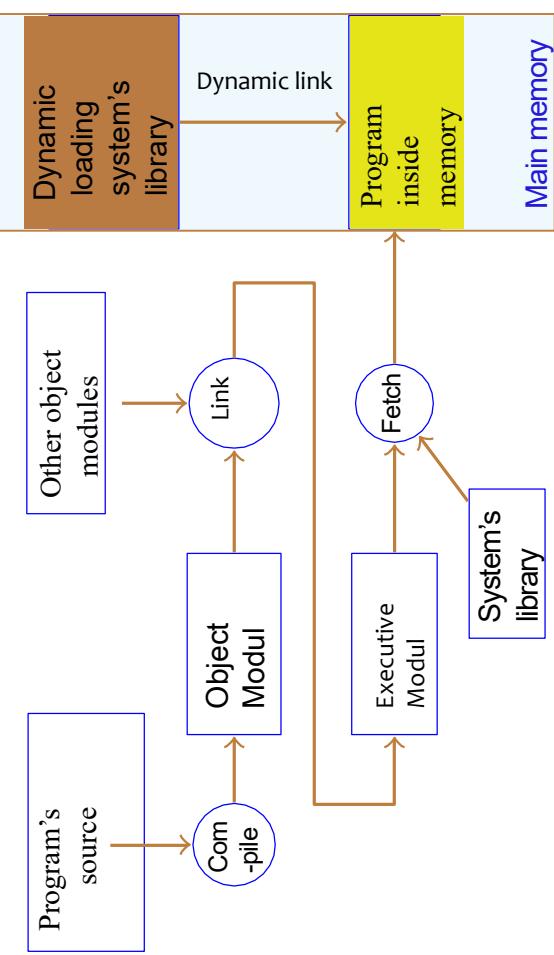
- Memory is an important system's resource
- Program must be kept inside memory to run
- Memory is characterized by size and access speed
- Memory leveling by access speed

Memory type	Size	Speed
Registers	bytes	CPU speed ( $\mu s$ ) 10
Cache on processor	Kilo Bytes	nano seconds
Cache level 2	KiloByte-MegaByte	100 nanoseconds
Main memory	MegaByte-GigaByte	Micro-Seconds
Secondary storage(Disk)	GigaByte-Terabytes	Mili-Seconds 10
Tape, optical disk	Unlimit	Seconds

## Chapter 3 Memory management

1. Introduction
2. Address binding

### Application program processing steps



## Chapter 3 Memory management

1. Introduction
2. Memory and program

### Execute a program

- Load the program into main memory
  - Read and analysis executable file (e.g. \*.com, file \*.exe)
  - Ask for a memory area to load program from disk
  - Set values for parameters, registers to a proper value
- Execute the program
  - CPU reads instructions in memory at location determined by program counter
    - 2 registers CS:IP for Intel's family processor (e.g. : 80x86)
    - CPU decode the instruction
    - May read more operand from memory
    - Execute the instruction with operand
    - If necessary, store the results into memory at a defined location
  - Finish executing
    - Free the memory area that allocated to program
    - Problem
  - Program may be loaded into any location in the memory
    - When program is executed, a sequence of addresses are generated
    - How to access memory?

## Chapter 3 Memory management

1. Introduction
2. Address binding

### Types of address

- Symbolic**
  - Name of object in the source program
  - Example: counter, x, y, ...
- Relative address**
  - Generated from symbolic address by compiler
  - Relative position of an object from the module's first position
  - Example: Byte number 10 from the begin of module
- Absolute address**
  - Generate from relative address when program is loaded into memory
  - For IBM PC: relative address <Seg :Ofs> → Seg \* 16+Ofs
  - Object's address in physical memory – physical address
  - Example: JMP 010Ah ⇒ jump to the memory block at 010Ah at the same code segment (CS)
    - if CS=1555h, jump to location: 1555h \* 10h + 010Ah = 1560Ah

## Chapter 3 Memory management

1. Introduction
2. Memory and program

- Example**
- Memory and program**
- Address binding**
- Program's structures**

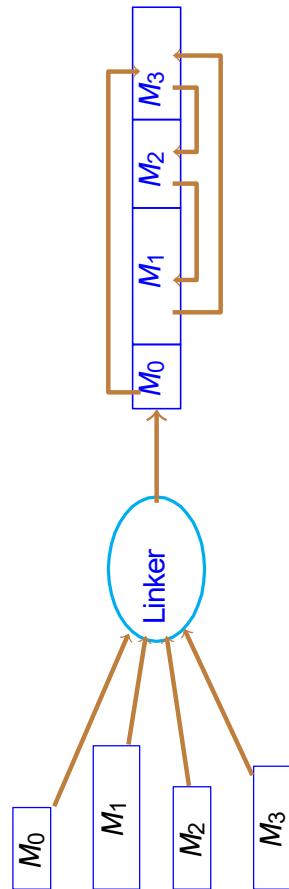
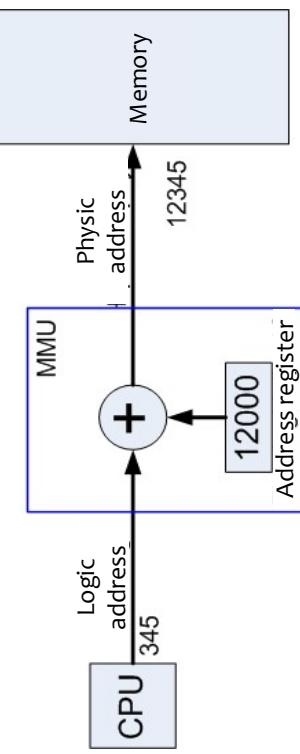
## ① Linear structure

## ② Dynamic loading structure

## ③ Dynamic link structure

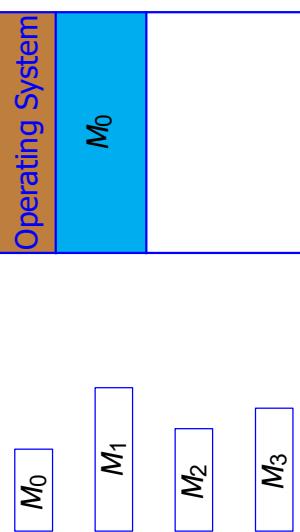
## ④ Overlays structure

- Logic address
  - Generate from process, (CPU brings out)
  - Converted to physical address when access to object in the program by the Memory management unit (MMU)
- Physical address
  - Address of an element (byte/word) in main memory
  - Correspond to the logical address bring out by CPU
- Program work with logical address



- After linking, modules are merged into a complete program
  - Contain sufficient information to be able to execute
  - External pointers are replaced by defined values
  - To execute, required only one time to fetch into the memory

- Example
- Memory and program
- Address binding
- Program's structures



- Each module is edited separately
- When executing, system will load and localize the main module

- Advantages
  - Simple, easy to link and localizing the program
  - Fast to execute
  - Highly movable

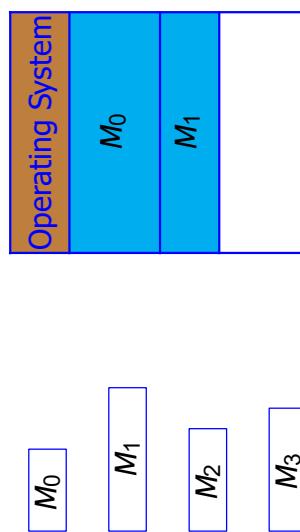
- Disadvantages

- Waste of memory
  - Not all parts of the program are necessary for the program's execution
- It's not possible to run the program that larger than physical memory's size

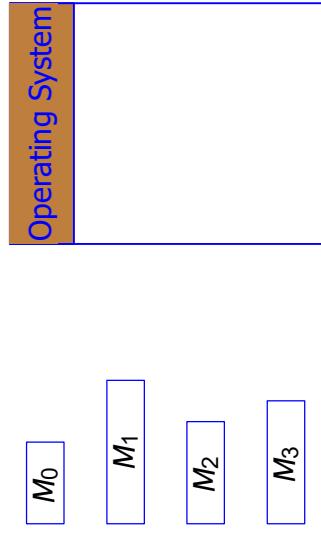
- Advantages
  - Simple, easy to link and localizing the program
  - Fast to execute
  - Highly movable

- Disadvantages

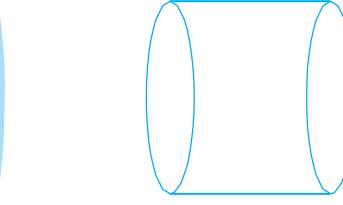
- Waste of memory
  - Not all parts of the program are necessary for the program's execution
- It's not possible to run the program that larger than physical memory's size

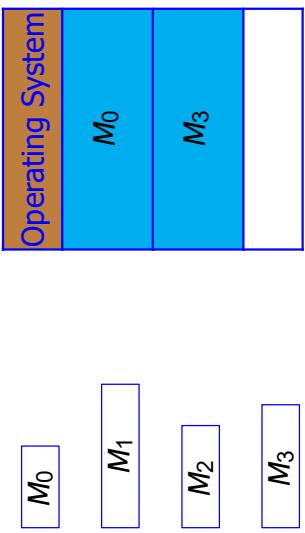


- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory

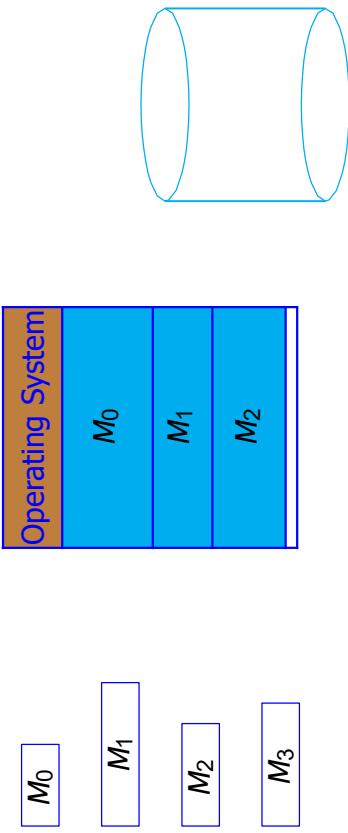


- Each module is edited separately



**Dynamic loading structure**

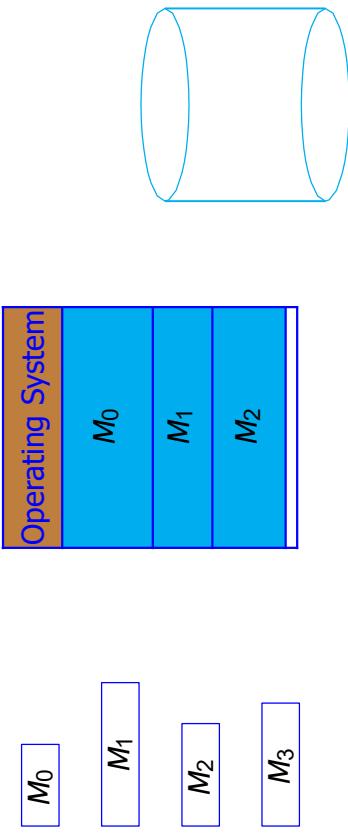
- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory
- When a module is finished using or not enough memory, bring unnecessary modules out

**Dynamic loading structure**

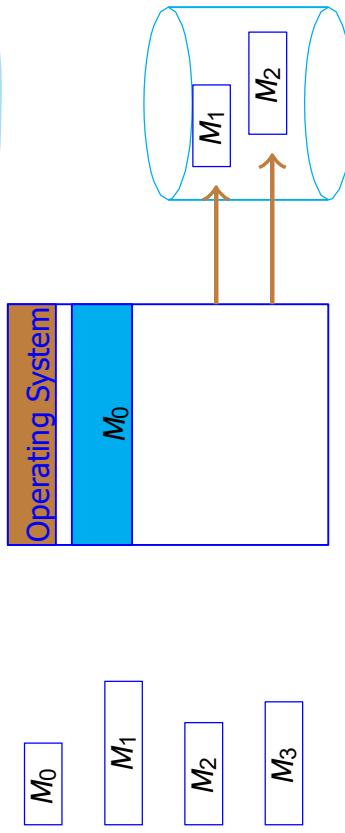
- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory

**Dynamic loading structure**

- Advantage**
- Can use memory are smaller than the program's size
  - High memory usage effectiveness if program is managed well
- Disadvantage**
- Slow when execution
  - Mistake may cause waste of memory and increase execution time
  - Require user to load and remove modules
  - User must understand clearly about the system
  - Reduce the program's flexible

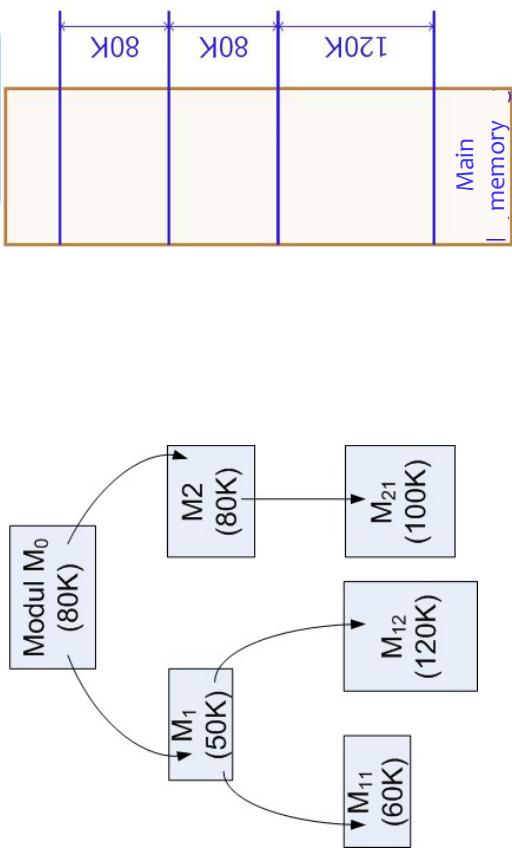
**Dynamic loading structure**

- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory

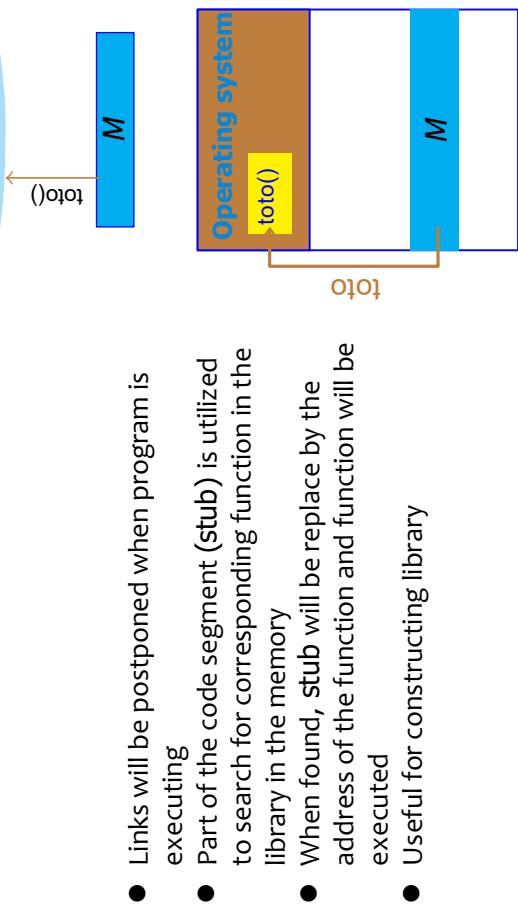
**Dynamic loading structure**

- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory
- When a module is finished using or not enough memory, bring unnecessary modules out

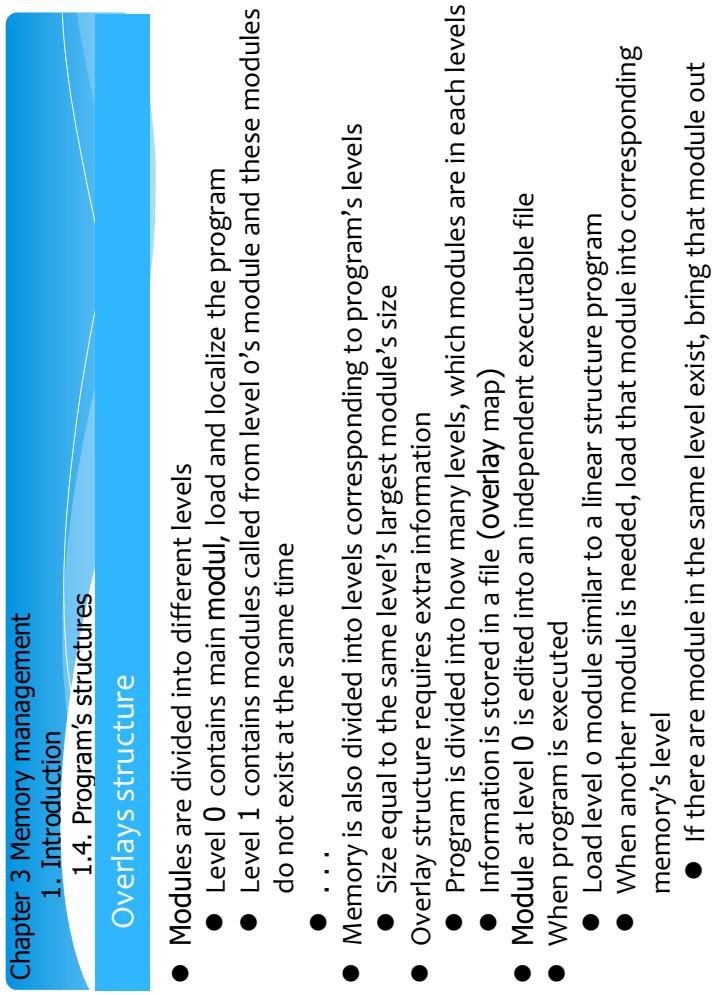
## Overlays structure



## Dynamic-link structure



## Overlays structure



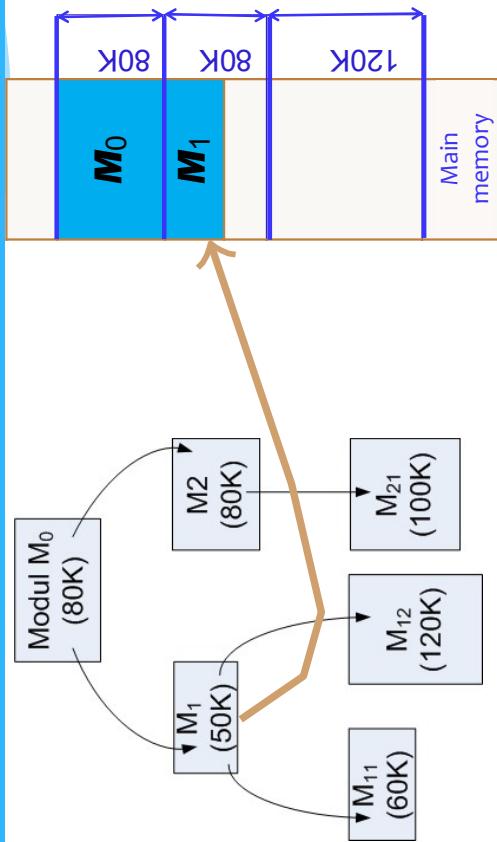
- Modules are divided into different levels
  - Level 0 contains main module, load and localize the program
  - Level 1 contains modules called from level 0's module and these modules do not exist at the same time
  - ...
  - Memory is also divided into levels corresponding to program's levels
  - Size equal to the same level's largest module's size
  - Overlay structure requires extra information
  - Program is divided into how many levels, which modules are in each levels
  - Information is stored in a file (overlay map)
  - Module at level 0 is edited into an independent executable file
  - When program is executed
  - Load level 0 module similar to a linear structure program
  - When another module is needed, load that module into corresponding memory's level
  - If there are module in the same level exist, bring that module out

## Chapter 3 Memory management

### 1. Introduction

### 1.4. Program's structures

#### Overlays structure

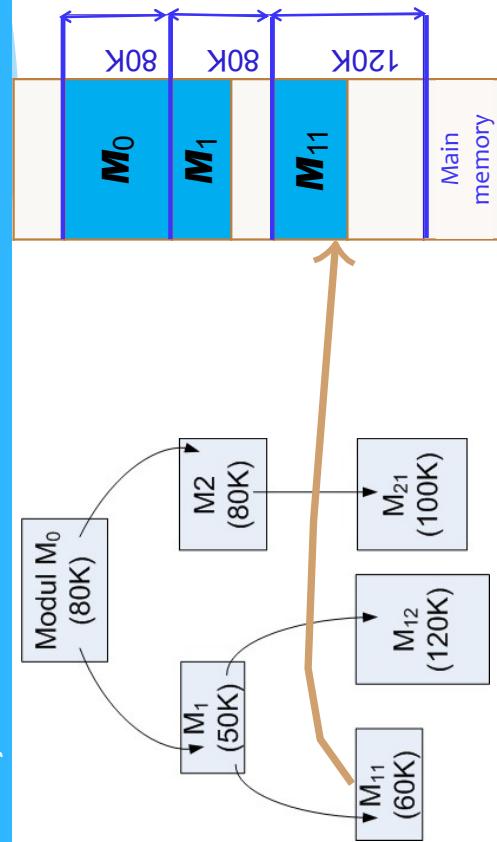


## Chapter 3 Memory management

### 1. Introduction

### 1.4. Program's structures

#### Overlays structure

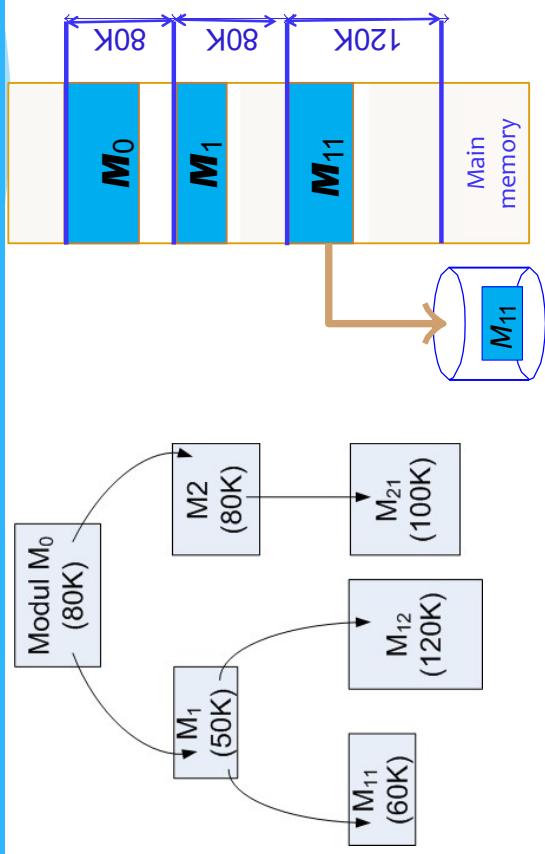


## Chapter 3 Memory management

### 1. Introduction

### 1.4. Program's structures

#### Overlays structure

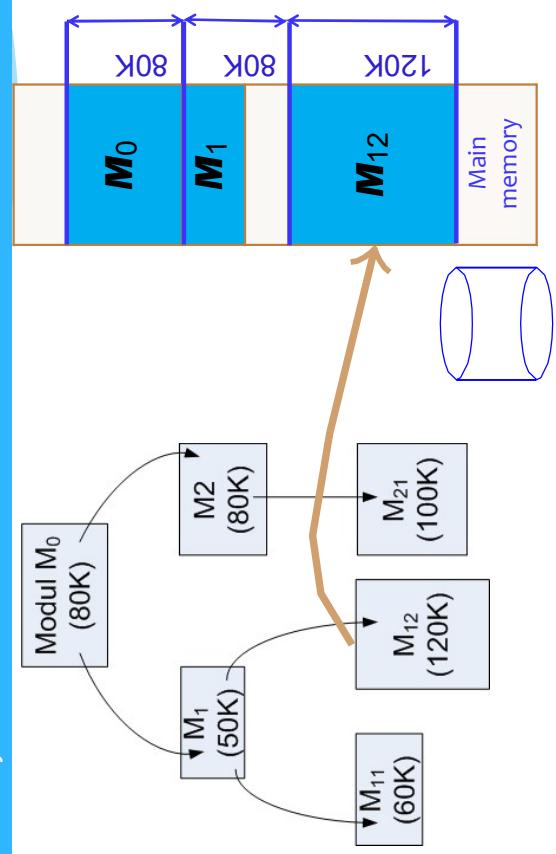


## Chapter 3 Memory management

### 1. Introduction

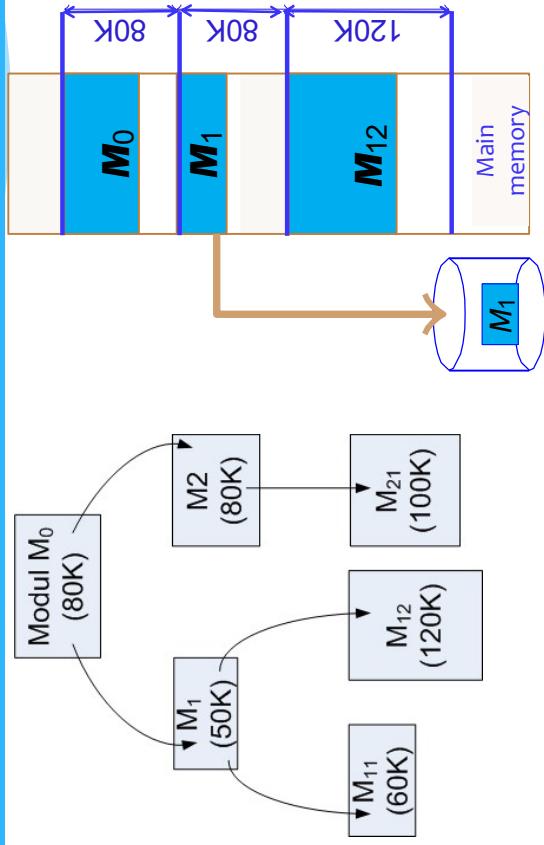
### 1.4. Program's structures

#### Overlays structure



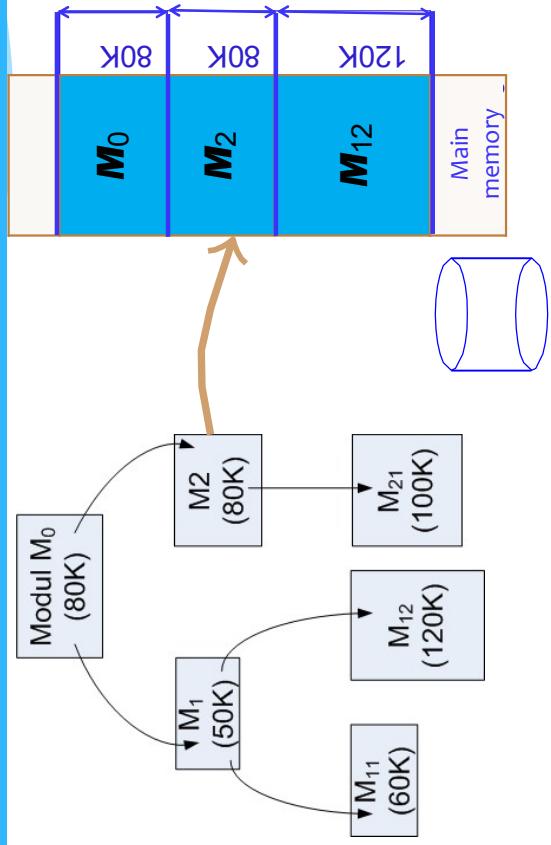
### Overlays structure - Conclusion

- Allow program with larger size than memory area size allocated by the operating system
- Require extra information from user
- Effectiveness is depend on provided information
- Memory usage effectiveness is depend on how program's modules are organized
  - If there are exist modules that larger than other modules in the same level ⇒ the effective is reduced
- Module loading process is dynamic but program's structure is static ⇒ not change at each time running
  - Provide more memory, the effectiveness does not increase



# Chap 3 Memory Management

- ① Introduction
- ② Memory management strategies
- ③ Virtual memory
- ④ Memory management in Intel's processors family



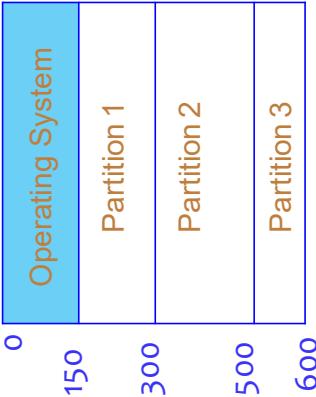
### Conclude

- Simple, easy for memory protection
  - Program and memory area have a protection lock
  - Compare 2 locks when program is loaded
  - Reduce searching time
- Must copy controlling module into many versions and save at many places
- Parallel cannot be more than  $n$
- Memory is segmented
  - Program's size is larger than the largest partition's size
  - Total free memory is large enough but can not load any program  
⇒ Fix partition structure, merge neighboring partition
- Application
  - Large size disk management
  - IBM OS/360 operating system

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

- Memory is divided into  $n$  parts
- Each part is called a *partition*
  - size: can be unequal
  - Utilized as and independent memory area
  - At a single time, only one program is allowed to exist
  - Programs lie inside memory until finish
- Example: Consider the following system



Process	Size	time
$P_1$	120	20
$P_2$	80	15
$P_3$	70	5
$P_4$	50	5
$P_5$	140	12

Queue
-------

### Free memory area selection strategy

**Strategies to select free area for process's request**

**First Fit** : First free area satisfy request

**Best Fit** : Most fitted area

**Worst Fit** : Largest area that satisfy request

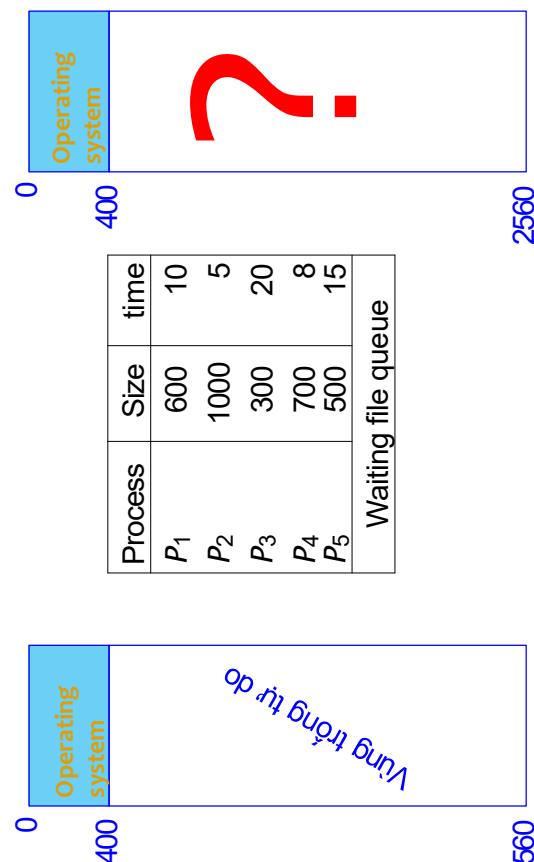
### Rule

Only one management list for free memory

- At the start, the whole memory is free for processes  $\Rightarrow$  largest *hole*
- When a process requests for memory
  - Search in the list for a large enough hole for request
  - If found
    - Hole is divided into 2 parts
    - One part allocate to process as requested
    - One part return to the management list
  - If not found
    - Wait until there is a hole large enough
    - Allow another process in the queue to execution (if the priority is guaranteed)
- When the process finish
  - Allocated memory area is returned to the free memory management list
  - Combine with other neighboring holes if necessary

### Free memory area selection strategy

### Main memory



- Suppose free memory area have the size 100K, 500K, 200K, 300K, and 600K (consequently),
- First-fit, Best-fit, and Worst-fit
- How will process with size 212K, 417K, 112K, and 426K loaded?

Using fitting fit %

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

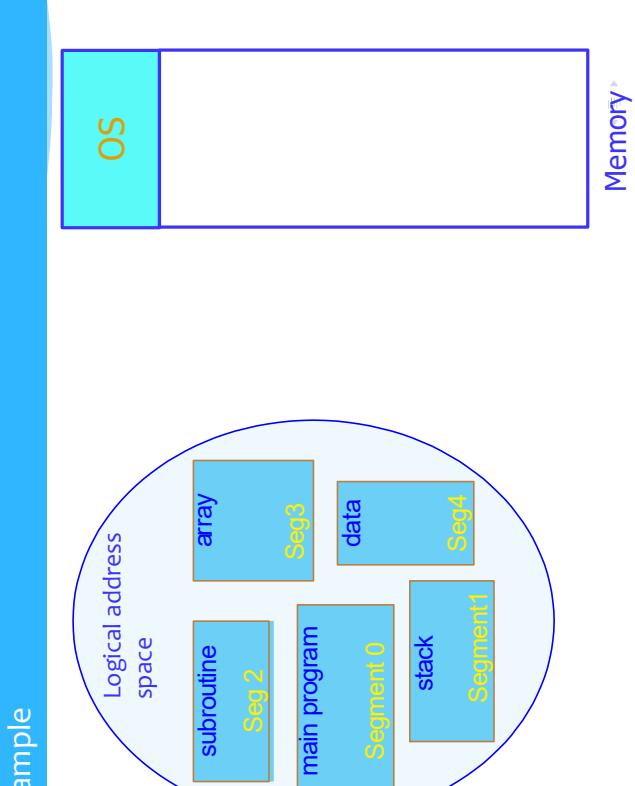
- After a long working time, the free holes are distributed and caused memory lacking phenomenon  $\Rightarrow$  Need to rearrange memory
- Move process
    - Problem: internal objects when move to new place will have new address
      - Use relocation register to store process's relocation value
      - Select method for lowest cost
        - Move all process to one side  $\Rightarrow$  largest free holes
        - Move processes to create a sufficient free hole immediately
    - Swapping process
      - Select a right time to suspend process
        - Bring process and corresponding state to external memory
          - Free allocated memory area and combine with neighboring areas
      - Reallocate to former place and restore state
        - Use relocation register if process is moved to different places

- Chapter 3 Memory management  
2. Memory management strategies  
2.3 Segmentation strategy
- ### Program
- In general, a program includes following modules
    - main program
    - Set of sub-routine
    - Variables, data structures, . . .
    - Modules, objects in program are defined by name
      - function sqrt(), procedure printf() . . .
      - x, y, counter, Buffer . . .
      - Member inside a module is determined based on the distance from the head of the module
        - Instruction 10th of function sqrt() . . .
        - Second element of array Buffer . . .

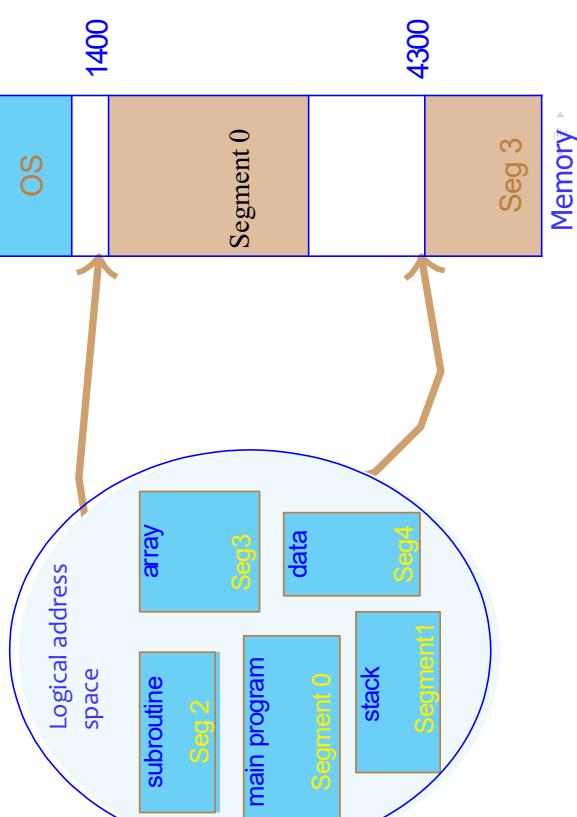
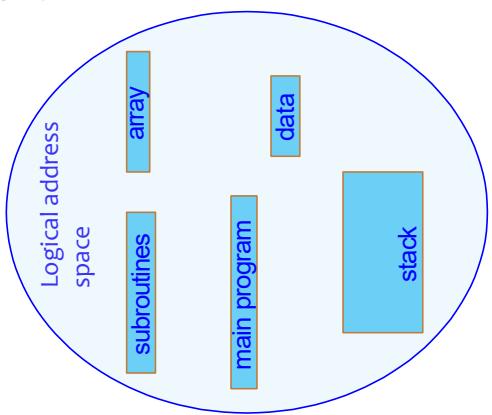
How program is placed inside memory?

- Stack stay at the higher area or Data stay at the higher area?
- Object's physical address . . . ?
  - Users donot care

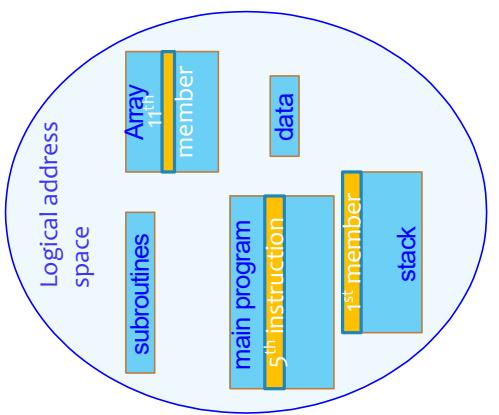
- No need to copy the controlling modules to different places
- Increase/decrease parallel factor depend on the number and size of programs
- Cannot run program with size larger than the physical memory size
- Cause memory waste phenomenon
  - Memory area is not used and not in the memory management list
    - Cause by the operating system error
    - By malicious software
- Cause external memory defragment phenomenon
  - Free memory area is managed but distributed  $\rightarrow$  cannot used
  - Cause internal memory defragment phenomenon
  - Memory allocated to process but not used by process



- When program is loaded into memory to execute. Program is combined of several segments
  - Each segment is a logic block, corresponding to a module
  - Code:** main(), procedure, function . . .
  - Data:** Global objects
  - Other segments: **stack, array . . .**
  - Each segment occupy an contiguous memory area
  - Has a **start position and size**
  - Can be located at **any place in the memory**



- Object in a segment is defined based on the relative distance from the start of the segment
  - 5<sup>th</sup> instruction of the main program
  - 1<sup>st</sup> member of the stack . . .
  - Where is the **location of these objects in the memory?**

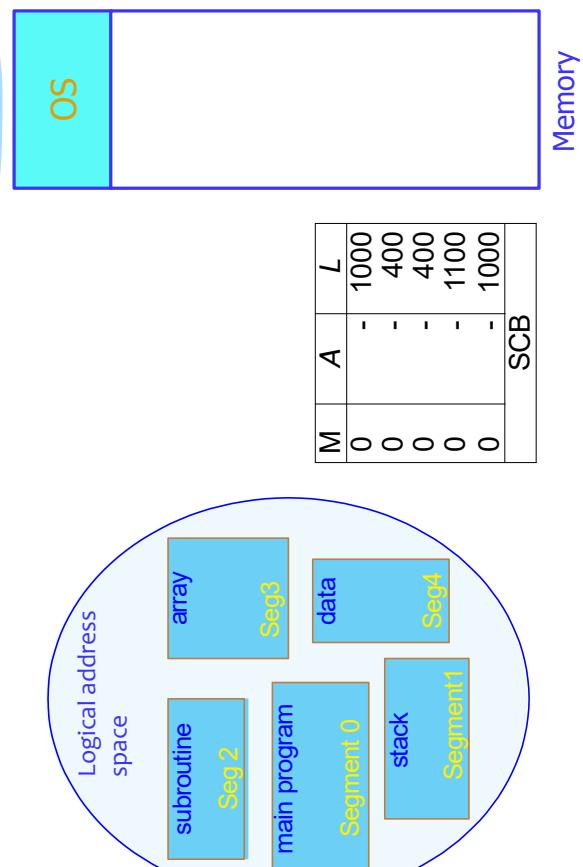


## Chapter 3 Memory management

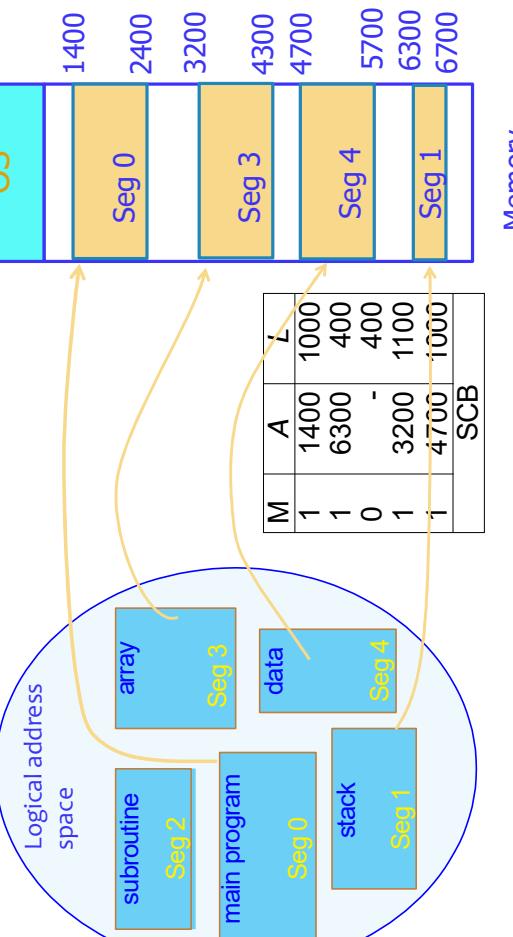
### 2. Memory management strategies

#### 2.3 Segmentation strategy

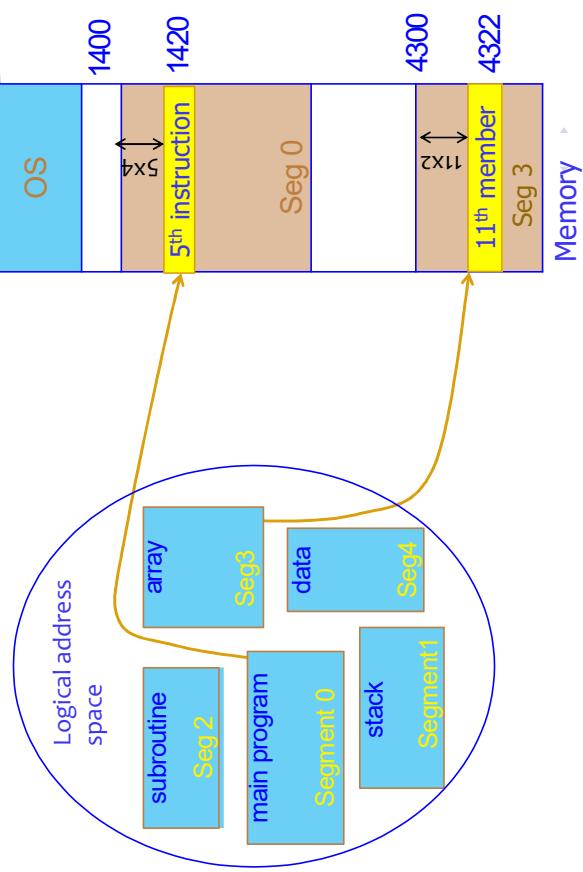
## Example



## Example



## Example



## Segmentation structure

- Program is a combination of modul/segment
- Segment number, segment's length
- Each segment can be edited independently.
- Compile and edit program -> create SCB (Segment Control Block)
- Each member of SCB is corresponding to a program's segment

Mark	Address	Length
0		
...		
n	...	...
		...

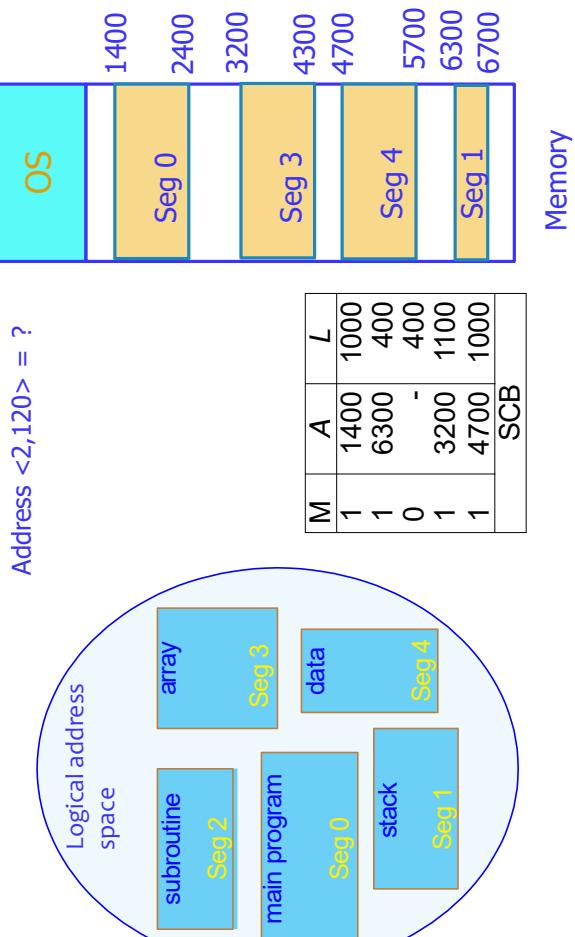
- Mark(0/1) : Corresponding segment is already inside memory
- Address: Segment's base location in memory
- Length: Segment's length
- Accessing address: segment's name (number) and offset

Problem: Convert from 2 dimension address to 1 dimension address

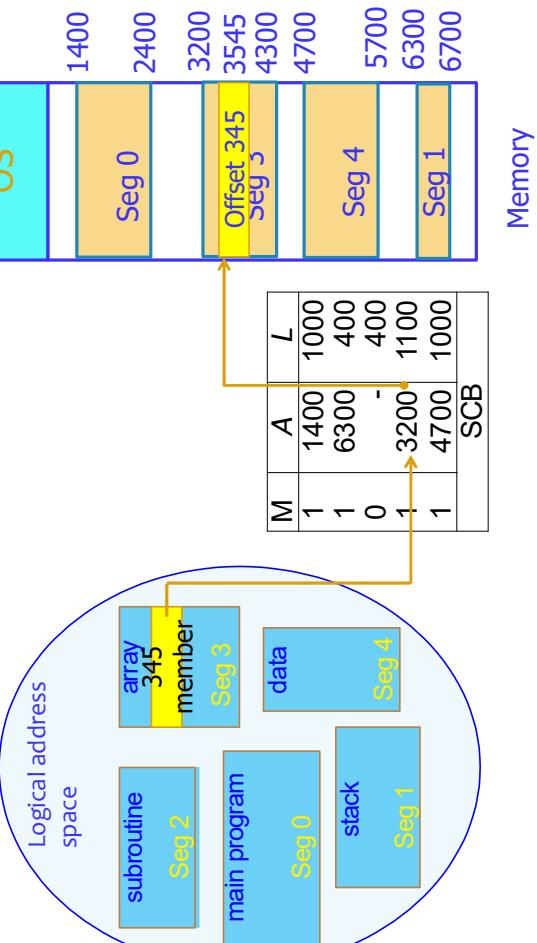
Chapter 3 Memory management  
2. Memory management strategies  
2.3 Segmentation strategy

Chapter 3 Memory management  
2. Memory management strategies  
2.3 Segmentation strategy

Example



Address <3,345> = ?



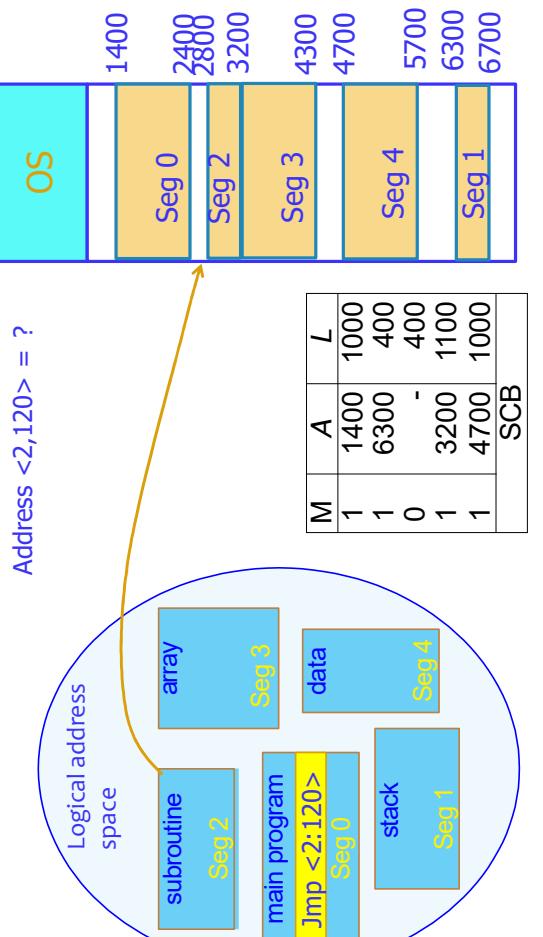
OS

1400  
2400  
3200  
3545  
4300  
4700  
5700  
6300  
6700

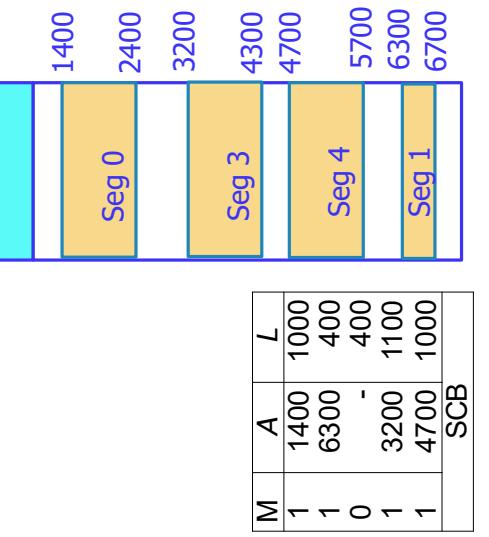
Memory

Chapter 3 Memory management  
2. Memory management strategies  
2.3 Segmentation strategy

Example



Address <4,185> = ?



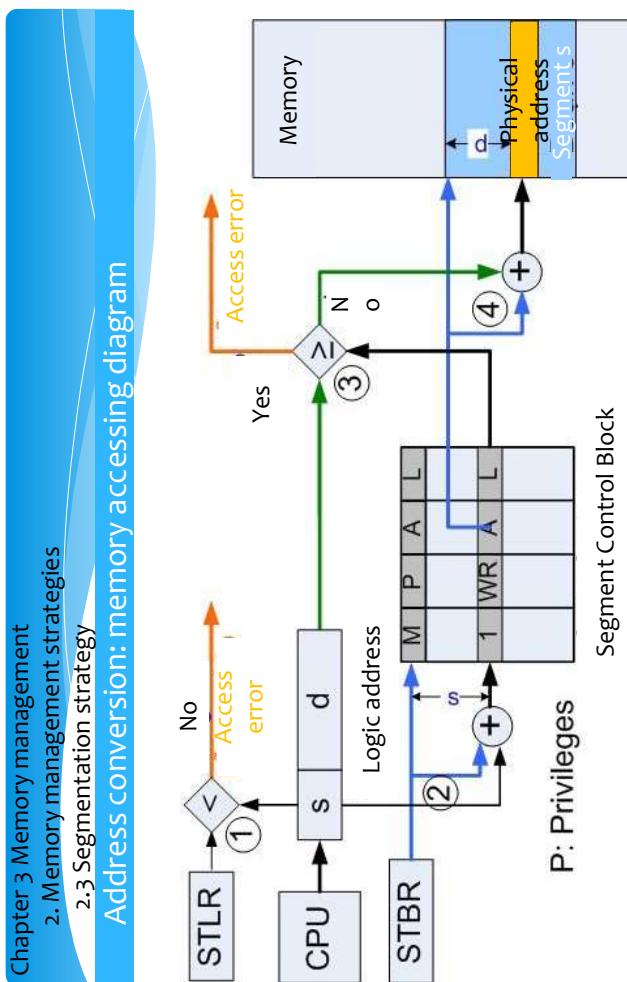
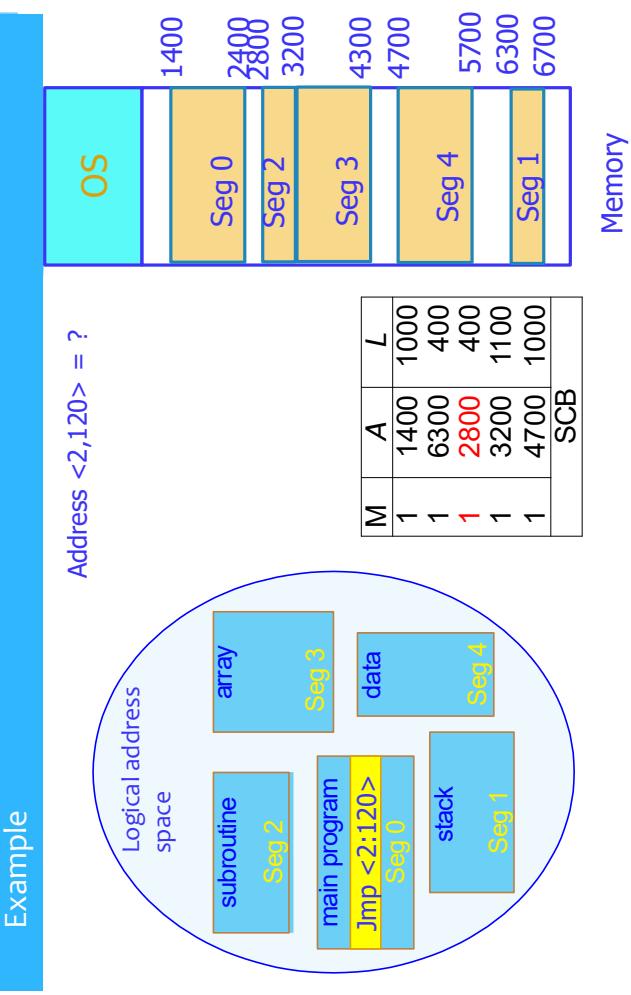
OS

1400  
2400  
3200  
3545  
4300  
4700  
5700  
6300  
6700

Memory

Chapter 3 Memory management  
2. Memory management strategies  
2.3 Segmentation strategy

Chapter 3 Memory management  
2. Memory management strategies  
2.3 Segmentation strategy



## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.3 Segmentation strategy

Conclude: cons

- Effective is depended on the program's structure
- Memory is fragmented
  - Memory allocated by methods first fit /best fit...
  - Require memory rearrangement (relocation, swapping)
  - Easier with the help of SCB
    - $M \leftarrow 0$  : Segment is not in memory
    - Memory's area defined by A and L is returned to the free memory management list
  - Select which module to bring out
    - Longest existed module
    - Last recently used module
    - Least frequently used module  $\Rightarrow$  Require media to record number and time that module is accessed
  - Solution: allocate memory for equal size segment ([page](#))?

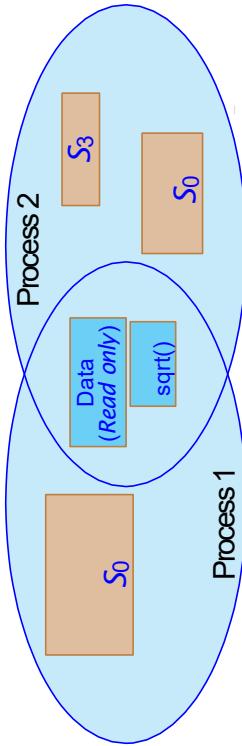
## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.3 Segmentation strategy

Conclusion: pros

- Module loading diagram does not require user 's participation
- Easy to protect segments
  - Check memory accessing error
  - Invalid address : more than segment's length
  - Check accessing's property
  - Code segment: read only  $\rightarrow$  Write into code segment: accessing error
  - Check the right to access module
  - Add accessing right (user/system) into SCB
- Allow segment sharing (Example: Text editor)



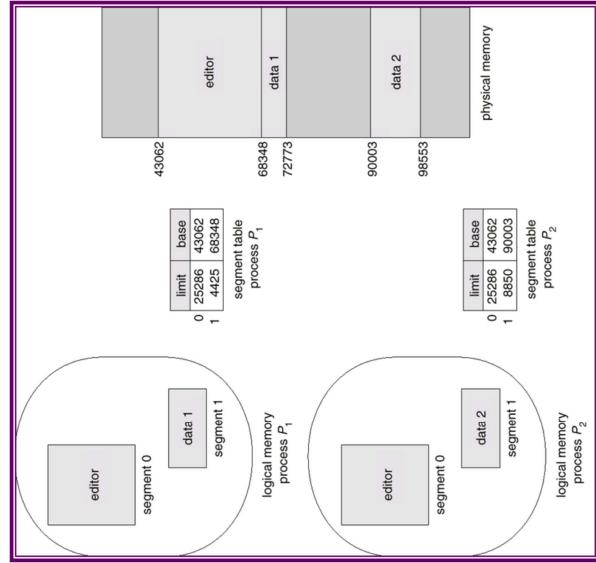
## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.3 Segmentation strategy

### Segment sharing: Main problem

- Sharing segment
  - Call (0, 120) ?
  - Read (1, 245) ?
- => must has the **same index number** in the SCB



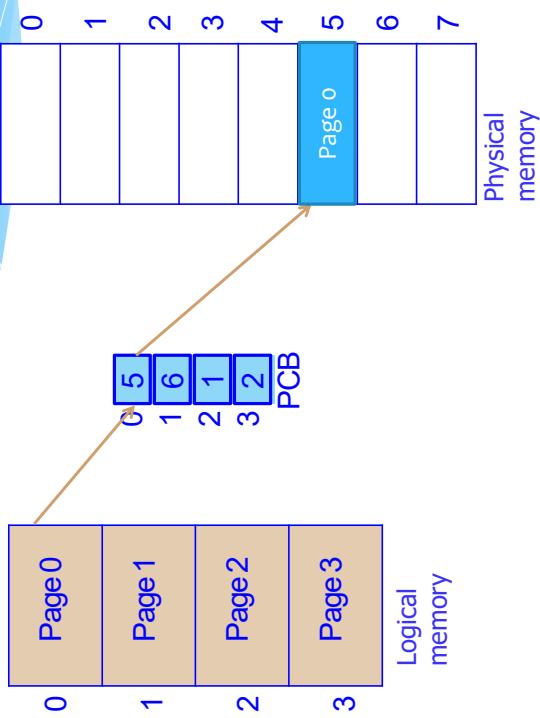
- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.4 Paging strategy

##### Example



## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.4 Paging strategy

##### Rule

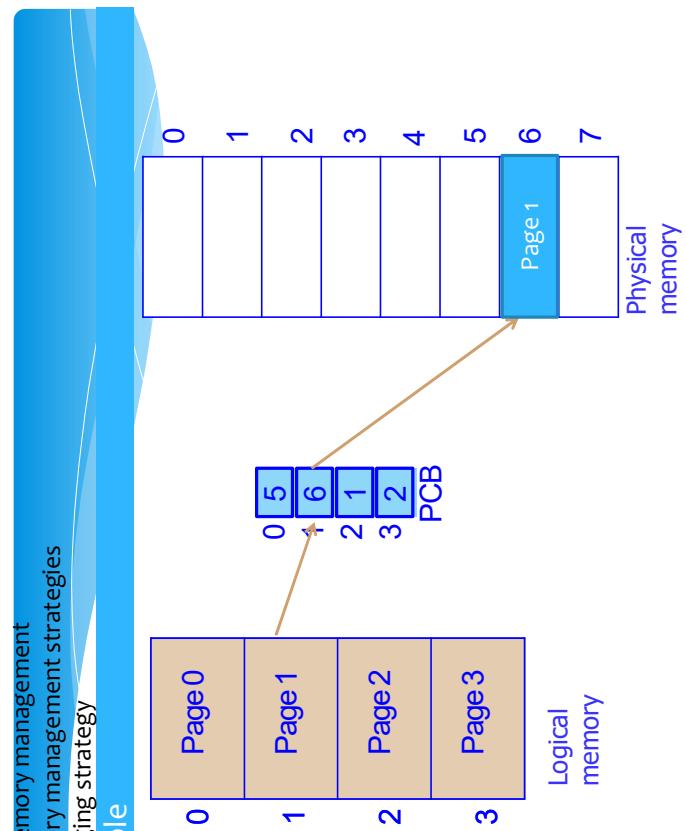
- Physical memory is divided into **equal size blocks**: **page frames**
  - Physical frame is addressed by number **0, 1, 2, . . .**:
  - frame's physical address**
  - Frame is the **unit for memory allocation**
  - Program is divided into blocks that have **equal size** with frame (**pages**)

## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.4 Paging strategy

##### Example



## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.4 Paging strategy

##### Rule (cont)

- When program is executed
  - Load logical **page** (from external memory) **into page's frame**
- Construct a PCB( Page Control Block) to determine the relation between **physical frame** and **logical page**
- Each element of PCB is corresponding to **a program's page**
  - Show which frame is holding corresponding page
  - Example  $PCB[8] = 4 \Rightarrow ?$
- Accessing Address is combined of
  - Page's number (p) : **Index** in PCB to find page's **base address**
  - Displacement** in page (d): Combined with **base address** to find the **physical address**

## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.4 Paging strategy

##### Note

- Frame size is always power of 2
  - Allow connection between frame number and displacement
  - Example: memory is addressed by n bit, frame's size  $2^k$
- frame displacement
 

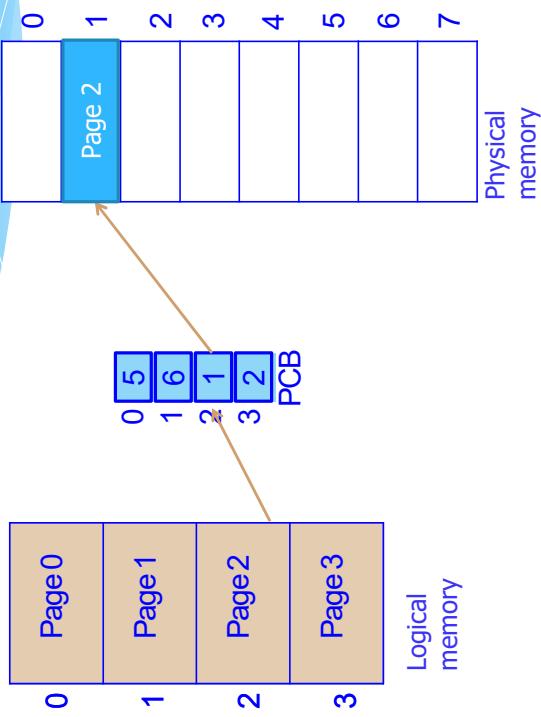
$n - k$	$k$
---------	-----
- Not necessary to load all page into memory
- Number of frame is limited by memory's size, number of page can be unlimited
  - PCB need Mark filed to know if page is already loaded into memory
  - $M = 0$  Page is not loaded
  - $M = 1$  Page is loaded
- Distinguish between paging and segmentation
  - Segmentation
  - Module is depend on program's structure
  - Paging
    - Block's size is independent from program
    - Block size is depend on the hardware (e.g.:  $2^9 \rightarrow 2^{13}$  bytes)

## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.4 Paging strategy

##### Example

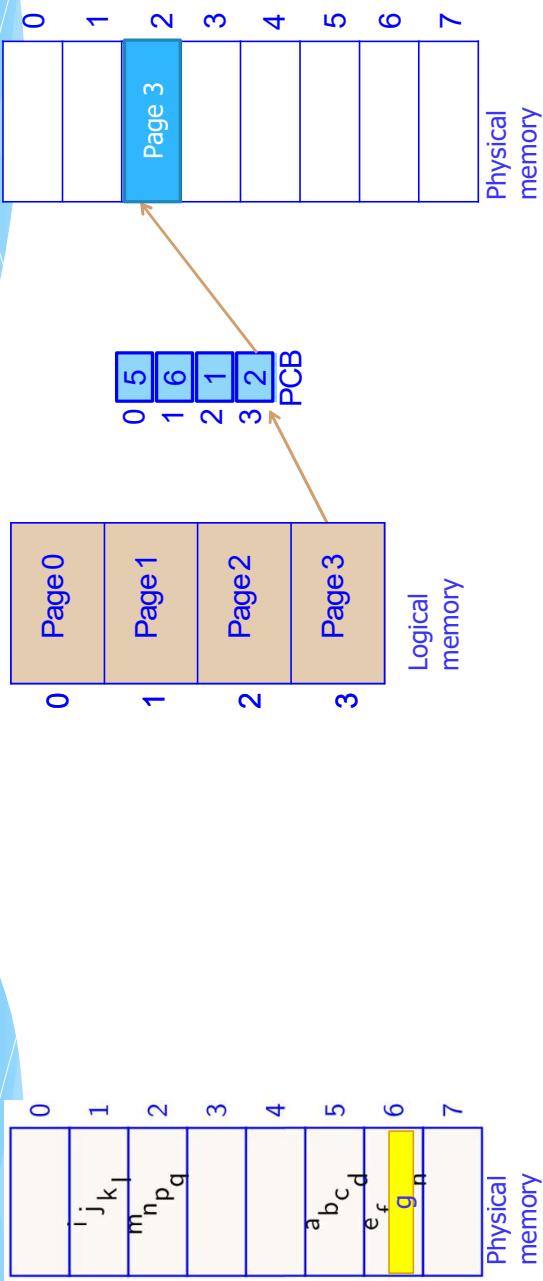


## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.4 Paging strategy

##### Example

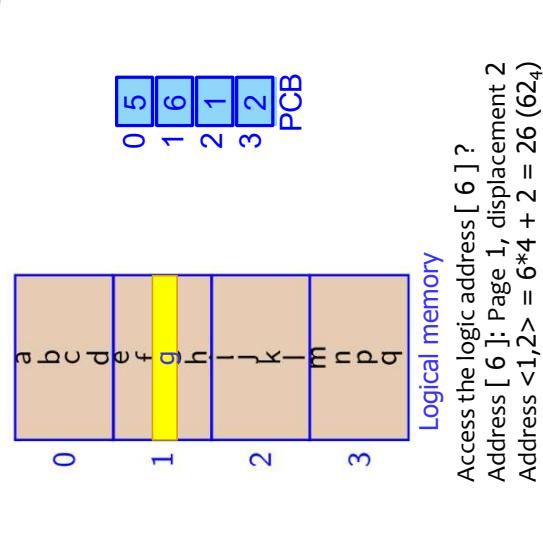


## Chapter 3 Memory management

### 2. Memory management strategies

#### 2.4 Paging strategy

##### Example

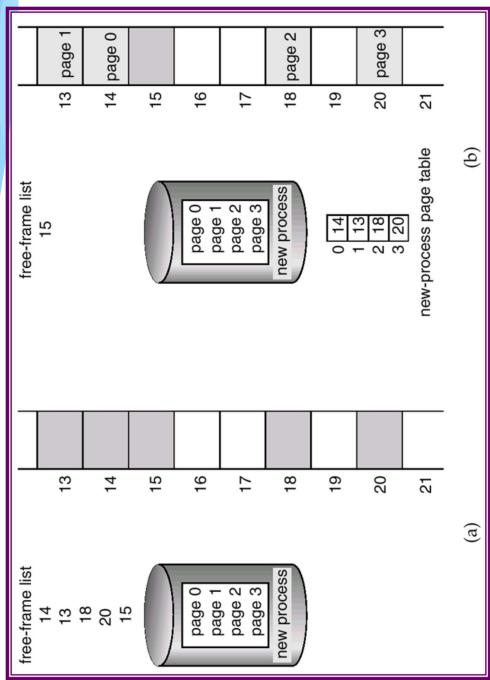


Access the logic address [ 6 ] ?  
 Address [ 6 ]: Page 1, displacement 2  
 Address <1,2> = 6\*4 + 2 = 26 ( $62_4$ )

## Load and replace page

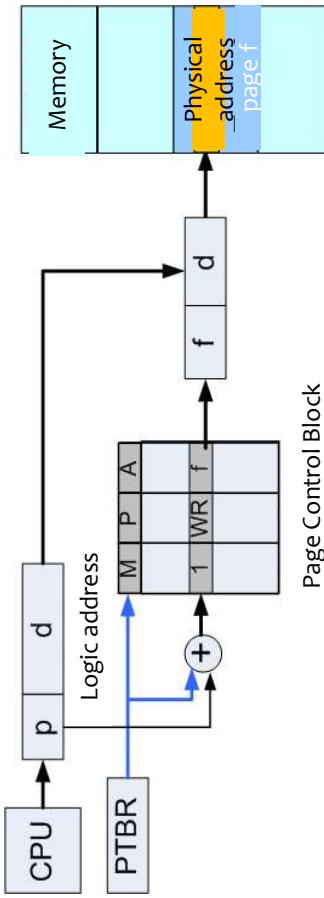
- Remark
  - Number of frame allocated to program
    - Large => Faster execution speed but parallel factor decrease
    - small => High parallel factor but execution speed slow because page is not inside memory
  - Effectiveness is depend on the page loading or page replacing strategy
    - Page loading strategy
      - Load all page: Load all program
      - Prior loading: predict next page will be used
      - Load on demand: Only load page when it's necessary
    - Page replacing strategy
      - FIFO First In First Out
      - LRU Least Recently Used
      - LFU Least Frequently Used
      - ...

## Program is running → Load program into memory

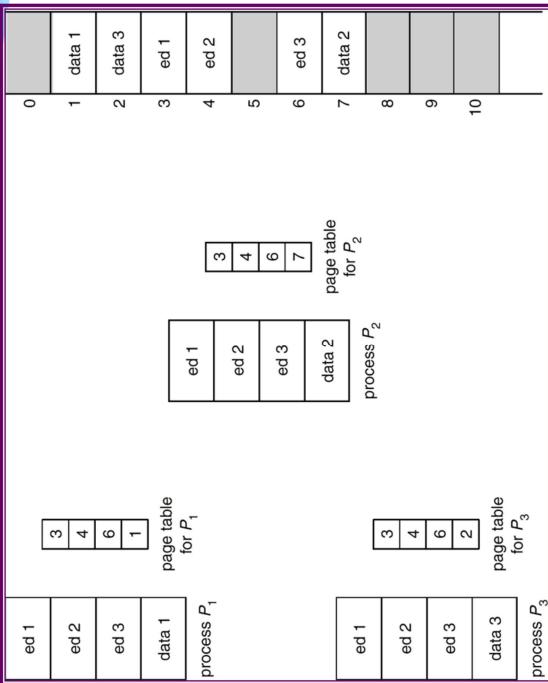


- If number of unused frame is enough => load all page
- If not enough => load parts of pages

- Increase memory access speed
  - Access memory 2 times (PCB and required address)
  - Perform connecting instead of adding operation
  - No external fragmentation phenomenon
  - High parallel factor
    - Only need several program's page inside memory
    - Program can have any size
  - Easy to perform memory protection
    - Legally access address (not more than page size)
    - Access property (read/write)
    - Access right (user/system)
  - Allow sharing page between processes



- Have external memory fragmentation
  - Always appear at the last page
  - Reduce memory fragmentation by reduce page size ?
  - Page fault more frequent
    - Large page control table
    - Require support from hardware
    - Cost for paging is high
  - When the program is large, page control block has many members
    - Program size  $2^{30}$ , page size  $2^{12} \rightarrow$  PCB has  $2^{20}$  members
    - Spend more memory for store PCB
    - Solution: multi level page



### Page sharing: Text editor

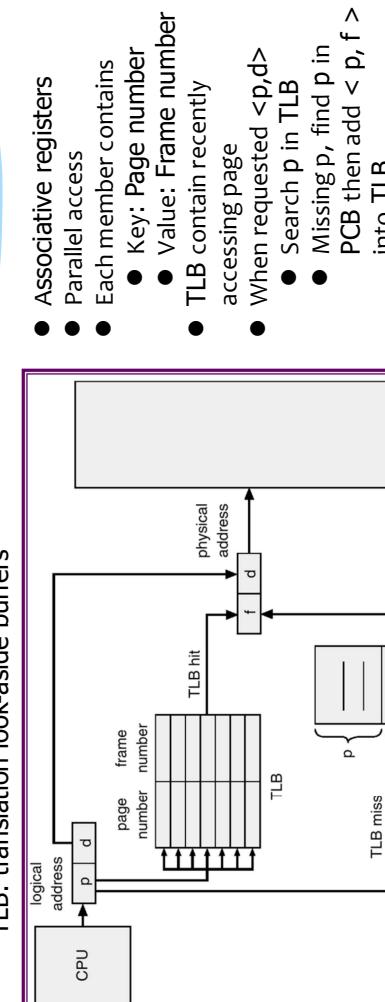
- Each page size 50K
- 3 pages for code
- 1 page for data
- 40 user
- No sharing
- Need 8000K
- Sharing
- Require 2150K

**Rule:** Divide PCB into pages

- Example: 2 level paging**
- Computer use 32 bit for addressing ( $2^{32}$ ); Page size 4K ( $2^{12}$ )
    - Page number - 20 bit
    - Offset in page -12 bit
    - PCB is paged. Page number is divided into
      - Outer page table (page directory) - 10 bit
      - Offset in a page directory – 10 bit
      - Access address has the form  $< p_1, p_2, d >$

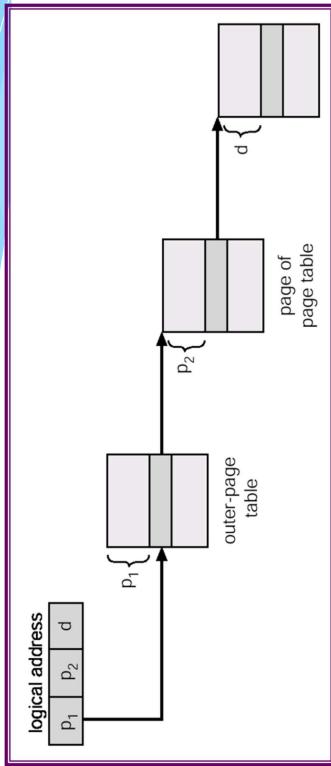
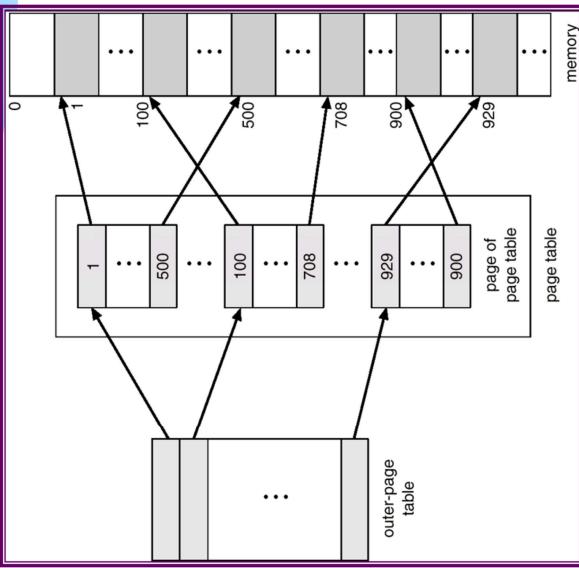
- Necessary while working in sharing environment
- Reduce the size of memory area for all processes
- Sharing code
  - Only one copy of sharing page in the memory
    - Example: text editor, compiler....
  - Problem: Sharing code can not change
    - Sharing page must be in the same logic address of all address
      - Same page id in the PCB
- The code and data are separately
  - Separate for each process
  - Can be in any position in the logic memory of the process

## Address translation buffer



98% memory access is done via TLB

## Multi-level page: Example 2 level paging



## Segmentation and paging combination strategy

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy

- When access: System load page directory into memory
- Unused page table and unused page are not necessary loaded into memory
- Problem: For 64 bit system
- 3, 4, ... Level paging
- Require access memory 4, 5, ... times  $\Rightarrow$  slow
- Solution: address translation buffer

### Conclusion

$M_0$	2340 B
$M_2$	4264 B
$M_1$	5730 B
$M_3$	1766 B

Segmentation		
M	A	L
0	-	2340
1	2140	5730
0	-	4264
0	-	1766

SCB

Segmentation and paging combination		
M	A	L
0	-	3
0	5	6
0	-	5
0	-	2

SCB

$PCB_2$

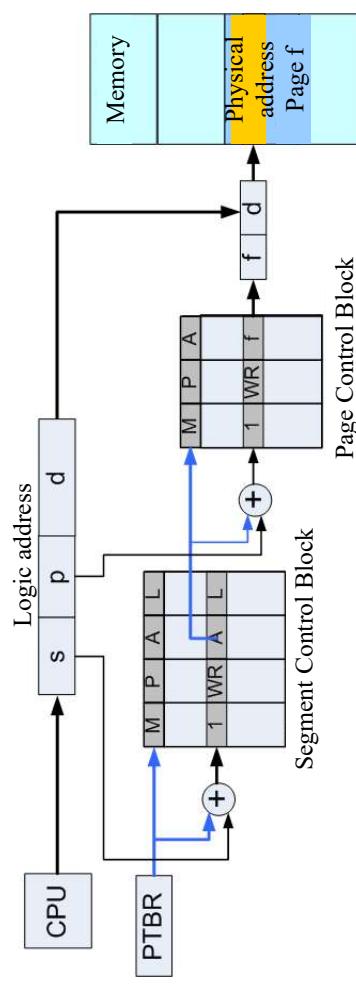
### Rule

- Program is edited as in segmentation strategy
- Create **SCB**
- Each member of SCB correspond to one segment, has 3 fields **M, A, L**
- Each segment is edited separately as in paging strategy
- Create **PCB** for each segment
- Memory accessing address: combination of  $3 < s, p, d >$
- Perform accessing address
- $STBR + s \Rightarrow$ : address of **s** member
- Check value of **MS**, load PCBs if it's necessary
- $As + p \Rightarrow$  Load the address of member **p** in PCBs
- Check value of **Mp**, load page **p** if it's necessary
- Connect Ap with **d**  $\Rightarrow$  physical address if it's necessary
- Utilized in processor Intel 80386, MULTICS . . .

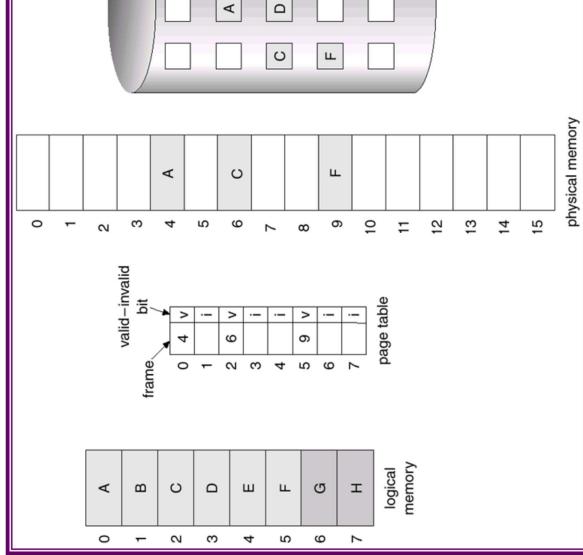
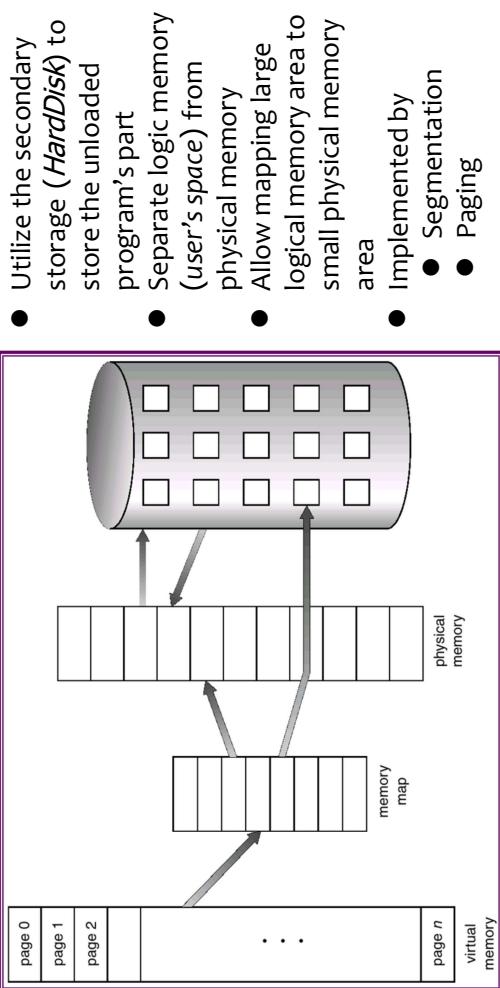
# Chap 3 Memory Management

- ① Introduction
- ② Memory management strategies
- ③ Virtual memory
- ④ Memory management in Intel's processors family

### Memory access diagram



## Concept of virtual memory

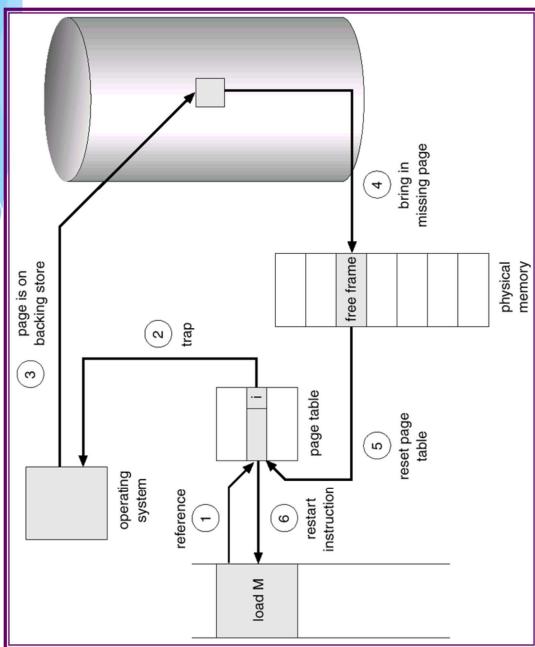


- Instruction must be placed in memory when executed !
- Whole program must stay inside memory ?
  - Dynamic loading, Overlays structures... : Partly loaded
  - Require special notice from programmer
  - Not necessary
  - Program's segment for handling errors
  - Errors occur least frequently, least frequently executed
  - Unused declared data
  - Declare a matrix 100x100, use 10x 10
  - Run program with one part inside memory will allow
    - Unlimited size
    - Write program in virtual address space
- Many program concurrently existed
  - Increase CPU's productivity
  - Reduce I/O request for loading and swapping programs
  - The size of the swapped part is smaller

- ## ① Introduction
- ## ② Page replacement strategy

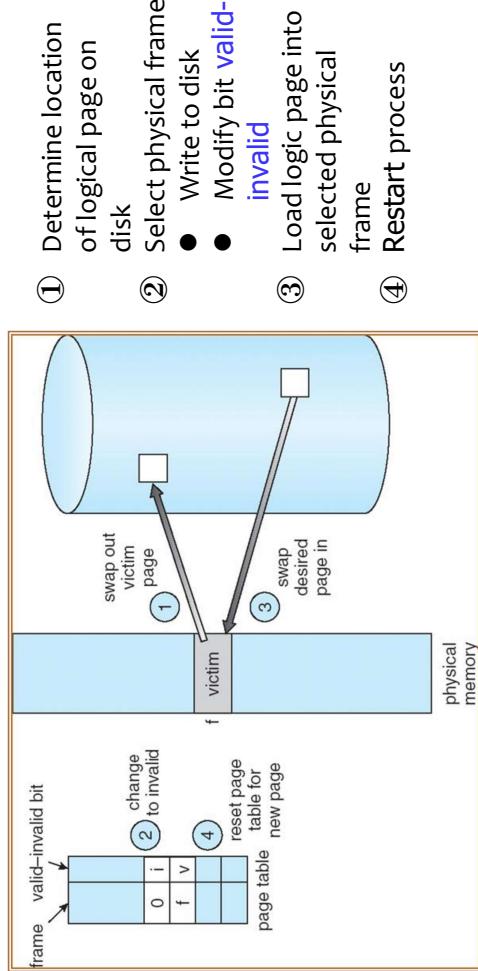
## ① Introduction

## ② Page replacement strategy



If there are no free frames, need to replace pages

- FIFO: First In First Out
- OPT/MIN: Optimal page replacing algorithm
- LRU: page that is Least Recently Used
- LFU: page that is Least Frequently Used
- MFU: page that is Most Frequently Used
- . . .



LRU

**Rule:** Replace page that is least recently used

- Effective for page replacing strategy
  - Guarantee that page fault is reduced when increase physical frame
    - Set of pages in memory with  $n$  frames is always a subset of pages in memory that have  $n + 1$  frames
  - Require support to know the last recently accessed time
  - How to implement?

IRI: Implementation

- Counter
    - Add one more field to record the accessed time into each member of PCB
    - Add a clock/counter to the Control Unit
    - Where there is a page access request
      - Increase counter
      - Copy the counter content into the newly added field in PCB
      - Need a procedure to update PCB (write to the added field) and procedure to search for a smallest accessed time value
    - Number Overflow phenomenon !?
  - List or Stack
    - Use a list or stack to record page number
    - Access to a page, put corresponding member to the top position
    - Replace: member in the last position
    - Usually implemented as a 2 dimension linked list
      - 4 pointer assigning operation ⇒ time consuming

100

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	0	1	2	0	1	3	0	2	1	0	3	2	0	1	7	0	2	1
7	7	0	1	2	0	1	3	0	2	1	0	3	2	0	1	7	0	2	1
0	1	2	0	1	3	0	2	1	0	3	2	1	0	3	2	1	0	1	2
0	1	2	0	1	3	0	2	1	0	3	2	1	0	3	2	1	0	1	2

### Remark

- Effective when the program has the linear structure
  - Least effective when the program has different modules call
    - Easy to implement
  - Utilize a queue to store program's pages inside memory
    - Insert into last position in the queue, replace page at the first position
    - Increase physical page, no guarantee of reducing page fault
      - Accessing sequence: 1 2 3 4 1 2 5 1 2 3 4 5
      - 3 frames: 9 page faults; 4 frames: 10 page faults

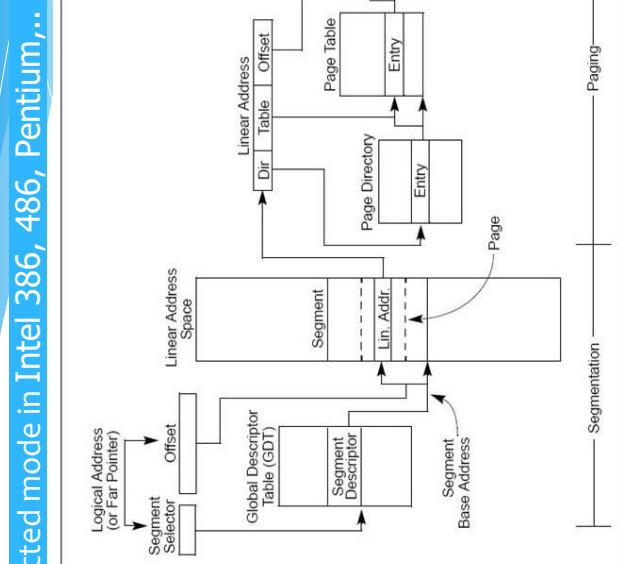
OPT

**Rule:** Replace page that has longest next used time

- Number of page fault is smallest
  - Problem: it's hard to predict program sequence

- Intel 8086, 8088
  - Only one management mode: Real Mode
  - Managed memory area up to 1MB (20bit)
  - Xác định địa chỉ ô nhớ bằng 2 giá trị 16 bit: Segment, Offset
  - Segment register: CS, SS, DS, ES,
  - Offset register: IP, SP, BP...
  - Physical address: Seg SHL 4 + Ofs
- Intel 80286
  - Real mode, compatible with 8086
  - Protected mode
  - Utilize segmentation method
    - Exploit physical memory up to 16M (24bit)
  - Intel 80386, Intel 80486, Pentium,..
  - Real mode, compatible with 8086
  - Protected mode : Combination of segmentation and paging
  - Virtual mode
    - Allow to run 8086 code in protected mode

- Use counter (one field of PCB) to record the last time page is accessed
- LFU: Page that has smallest counter will be replaced
  - Page that is frequently used
  - Important page ⇒ reasonable
  - Initialization page, only used at start ⇒ unreasonable ⇒ Shift right the counter by 1 bit (time div 2)
- MFU: Replace page with largest counter
  - Page with smallest value, just recently loaded and not used much



## Chap 3 Memory Management

- ① Introduction
- ② Memory management strategies
- ③ Virtual memory
- ④ Memory management in Intel's processors family

## Chapter 4 File system management

- ① File system
- ② File system's implementation
- ③ Information organization on disk
- ④ FAT system

### Operating System (*Principles of Operating Systems*)

- Đỗ Quốc Huy
- huydq@soict.hust.edu.vn
- Department of Computer Science
- School of Information and Communication Technology

HUST

#### Chapter 4: File system management 1. File system

##### • File's concept

- Directory structure

#### Chapter 4 file system management

- To keep information for **long time** and **reuse** later -> store on external memory (disk, magnetic tape, optical disk,...) => **file**
  - **Data or program file**
- Many files ⇒ file system
  - Files system combined of 2 parts
    - files: Contain data/program
    - directory : provide information about file
- Files system is large ⇒ How to manage?
  - File's properties, required operation?
  - How to store and access file on storage devices?
    - Storage space allocation, free memory management

HUST

## File's attributes

- **Name:** sequence of character (e.g. "hello.c")
  - only information kept in human-readable form
  - Can be distinguished by upper-case/lower-case
  - Guarantee the independence of file from process, user...
  - A created file hello.c by notepad on Windows
  - B uses emacs on Linux to modify a file specified by name hello.c
- **Identifier:** A tag to define an unique file

## Introduction

- Information is stored on different medias/devices

- Example: Disk, tape, optical disk...

- Storage device is modeled as an array of memory block



- File is a set of information stored on storage devices
  - a storage unit of OS on external devices
  - contains sequence of bits, bytes, lines, records,... Contain meaning defined by creator

## File's attributes

- **Type:** Used in system supports different types
  - Type of file is defined based on 1 part of file name
    - Example: .exe, com/.doc, .txt/.c, .jav, .pas/.pdf, .jpg,...
  - Based on type, OS decides corresponding operation
    - Run an execution file that source is modified => Recompile
    - Double click on a text file (\*.doc) => Call word processor
- **Position:** Point to device and location of file on that device
- **Size:** Current/maximum size of file
- **Protection:** controls who can do reading, writing, executing...
- **Time:** Creation time, modified time, last used time ...

## Introduction

- Structure of file is defined by type of file

- Text file: Sequence of character organized into lines
  - Object file: Bytes organized into block to be read and edited by the linker
  - Executive File: Sequence of codes can be run in memory
- ...

## Chapter 4: File system management

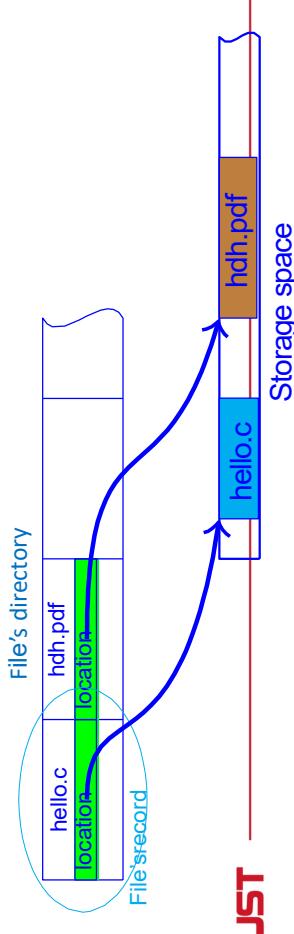
### 1. File system

## Chapter 4: File system management

### 1. File system

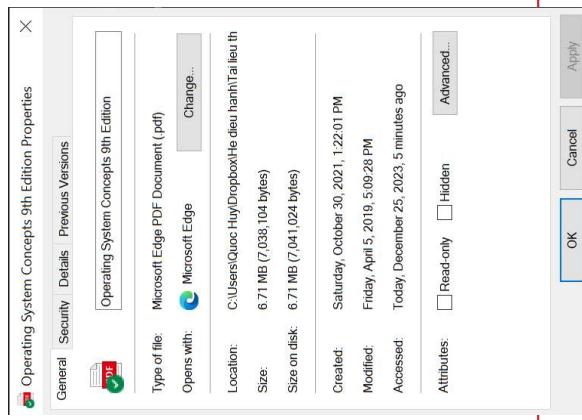
#### File's attributes (cont.)

- File's record are stored in **File's directory**
  - Size may be up to **Megabytes**
  - Often stored on external memory
  - Directory parts are loaded into memory when necessary



#### File's attributes

- File's record are stored in **File's directory**
  - Size may be up to **Megabytes**
  - Often stored on external memory
  - Directory parts are loaded into memory when necessary



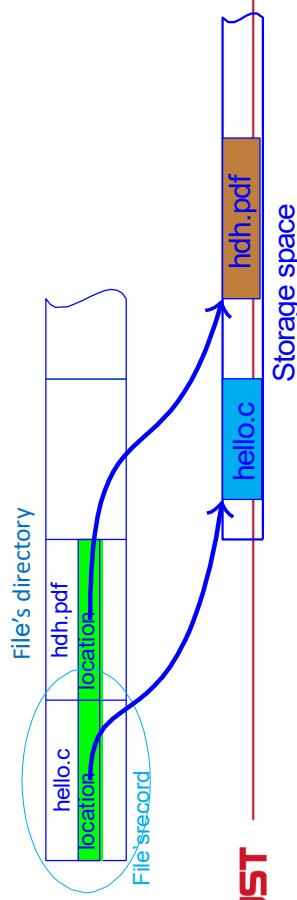
## Chapter 4: File system management

### 1. File system

#### File's concept

#### ● Directory structure

- File's attributes are stored in data structure: **File's record**
  - May contain only file's name and file's identifier; file's identifier define other information
  - Size from several bytes to kilobytes



## Chapter 4: File system management

### 1. File system

#### File's attributes (cont.)

- File's attributes are stored in data structure: **File's record**
  - May contain only file's name and file's identifier; file's identifier define other information
  - Size from several bytes to kilobytes

## 1. File system 1.2. Directory structure

## 1. File system 1.2. Directory structure

### Operations with directory

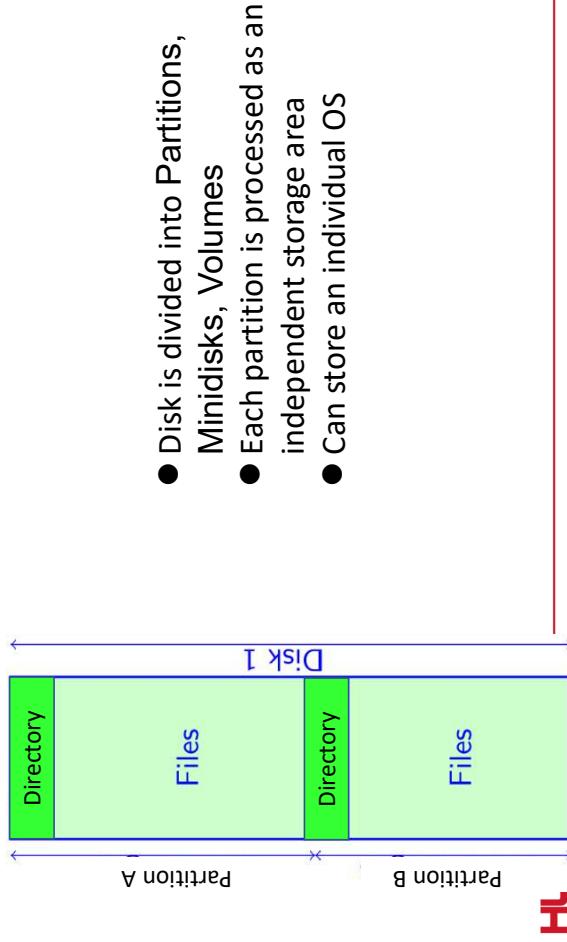
- Each partition contain information about contained files
  - File's information is stored in device's directory
  - Directory is a translation table allow mapping from one name (file) to a member inside directories
    - Directory can be implemented by different methods
  - Require operations for insert, create, delete or show the list

HUST

### Operations with directory

- Each partition contain information about contained files
  - File's information is stored in device's directory
  - Directory is a translation table allow mapping from one name (file) to a member inside directories
    - Directory can be implemented by different methods
  - Require operations for insert, create, delete or show the list

HUST



## 1. File System 1.2. Directory structure

### Operations with directory

- Operations
  - **Seek file:** Search for an item corresponding to a file name
    - Create file: require to create new item in directory
    - Delete file: remove corresponding item in the directory
    - Listing file: Show the list of files and corresponding item in directory
  - **Rename file:** Change file's name, position in the directory's structure
    - Traverse the file system: Access all directory and content of all files in the directory (**backup data to tape**)

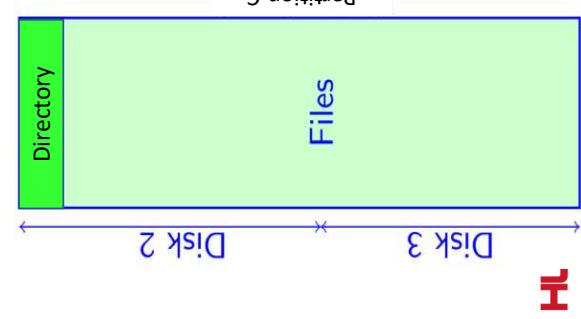
## 1. File system 1.2. Directory structure

### Operations with directory

- Disk is divided into Partitions, Minidisks, Volumes
  - Each partition is processed as an independent storage area
  - Can store an individual OS

Merge several disks into a large logic structure

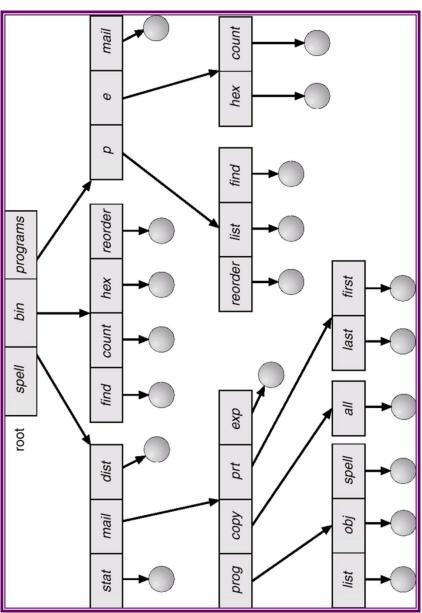
- User only care about file and directory's structure
- Do not care about how physical memory space is allocated to files



## 1. File system 1.2. Directory structure

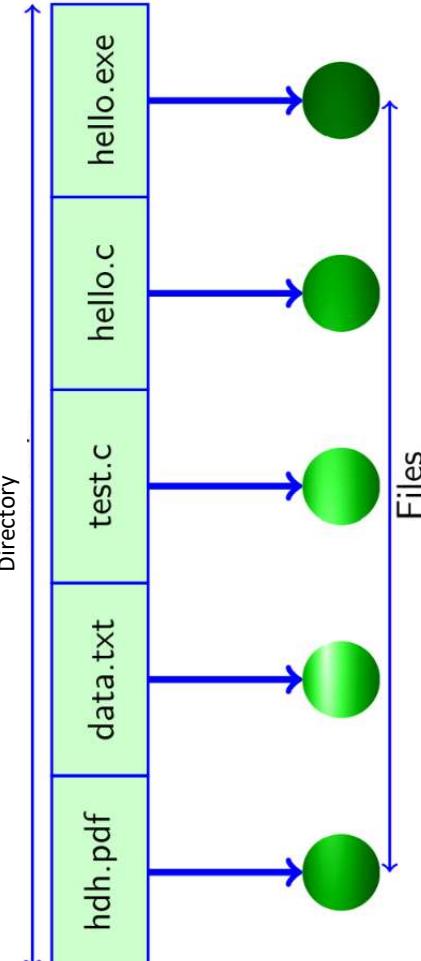
## 1. File system 1.2. Directory structure

### Tree structure directory



- A (relative/absolute) path to file exist
  - Sub-directory is a file is treated specially (bit for marking)
  - Operations create/delete/add... performed on current directory
- HUST** • Delete a sub-directory  $\Rightarrow$  delete its sub-tree

### One level directory

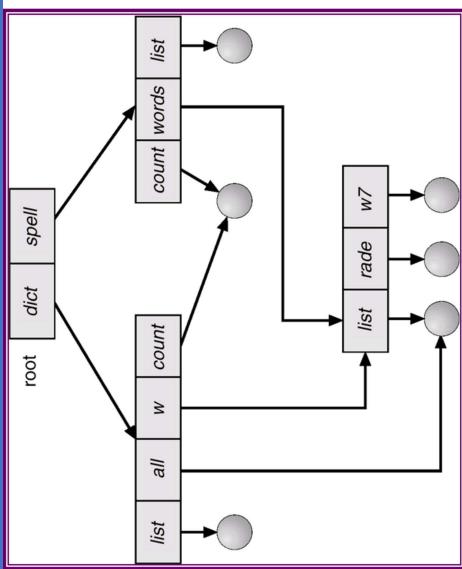


- Simplest structure, files are stored in the same directory
  - If number of user and file is large, it's possible for the filename to be duplicated
- HUST** • Each user has his own directory

## 1. File system 1.2. Directory structure

## 1. File system 1.2. Directory structure

### Acyclic-Graph structure



- Each user has his own private folder, when working, only work with private folder
  - When log in, system check and allow user to work with private folder
  - When a user is added
    - System create a new member in *Master file directory*
    - Create *User file directory*
    - Solve the duplicate file name problem; Effect when user are independent
- HUST** • When traversing the directory (backup), file can be traversed many times

- User can link to a file from other user
  - When traversing the directory (backup), file can be traversed many times
- HUST** • Delete file: link/ content (file-created-user /last link)

## 2. File system's implementation 2.1 Directory implementation

## Chapter 4 File system management

- Method
- ① Linear list with pointer to data blocks
  - ② Hash table – Hash table with linear list

### ① File system

### ② File system's implementation

### ③ Information organization on disk

### ④ FAT system

HUST

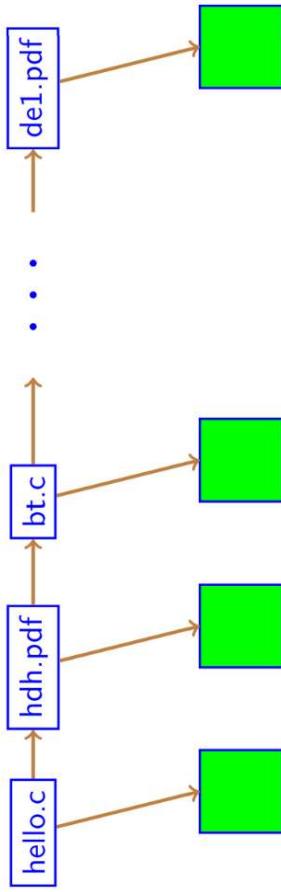
## 2. File system's implementation 2.1 Directory implementation

- Method

- ① Linear list with pointer to data blocks

- Simple for programming

- Time consuming when operate with directories
  - Must traverse all the list  $\Leftarrow$  Use binary tree?



HUST

## 2. File system's implementation 2.1 Directory implementation

### ● Directory implementation

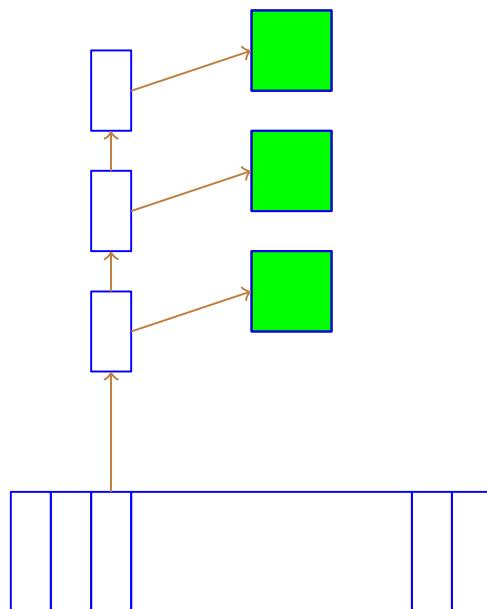
### ● Storage area allocation methods

### ● Free storage area management

HUST

## 2. File system's implementation 2.1 Directory implementation

## 2. File system's implementation 2.1 Directory implementation



Hash table

- ② Hash table with linear list
- Reduce directory traversing time
  - Require an effective hash function

HUST

## 2. File system's implementation 2.2 Storage area allocation methods

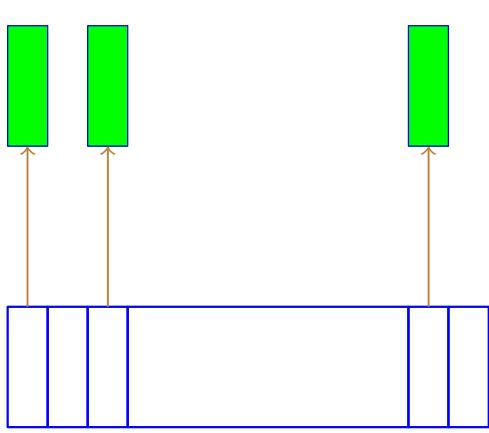
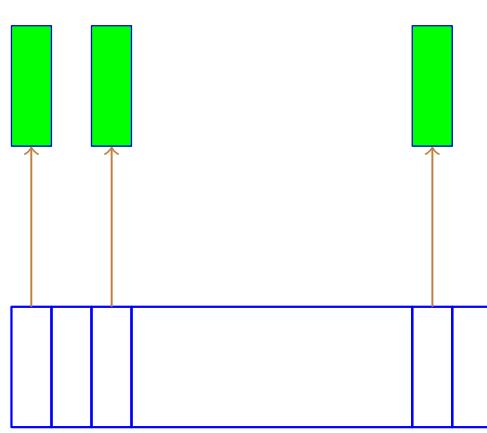
### ● Directory implementation

#### ● Storage area allocation methods

#### ● Free storage area management

HUST

Hash table



Hash table

$$h(\text{Name}) = \frac{\sum_{i=1}^{\text{Len}(\text{Name})} \text{ASCII}[\text{Name}[i]]}{\text{Table\_Size}}$$

- **Collision problem:** hash function return same result with 2 different file names

- Fixed table size problem:  
Increase size → need to recalculate existed numbers

HUST

## 2. File system's implementation

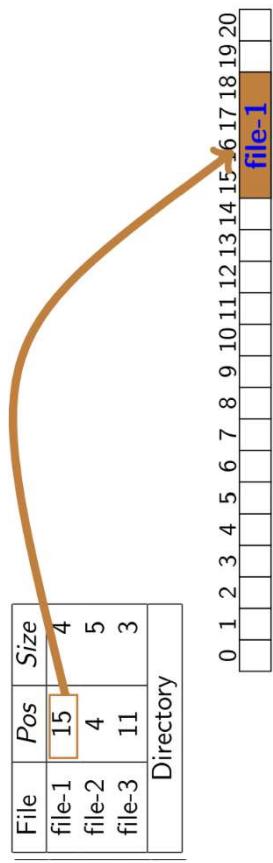
### 2.2. Storage area allocation methods

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Continuous Allocation

Rule: File is allocated with continuous memory block



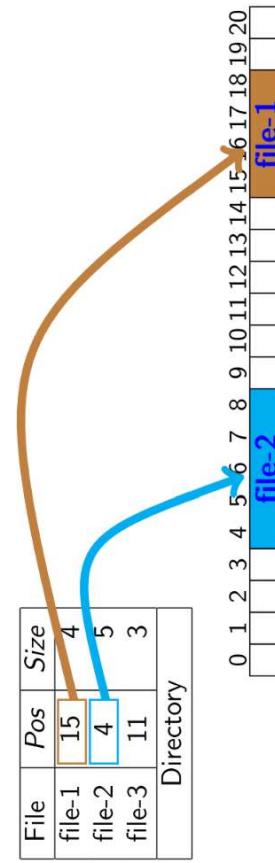
HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Continuous Allocation

Rule: File is allocated with continuous memory block



HUST

#### Methods

- Increase performance of sequence access
- Easy to randomly access file
- Easy to manage file

#### Methods

- Continuous Allocation
- Linked List Allocation
- Indexed Allocation

HUST

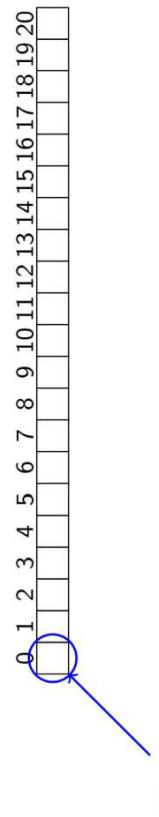
## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Continuous Allocation

Rule: File is allocated with continuous memory block

File	Pos	Size
file-1	15	4
file-2	4	5
file-3	11	3



HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

## 2. File system's implementation

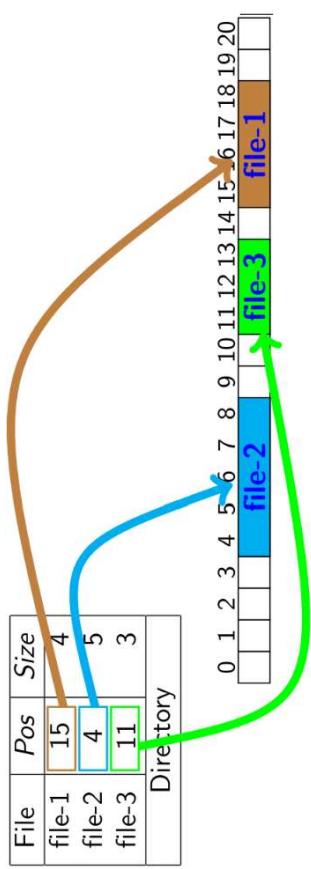
### 2.2. Storage area allocation methods

#### Linked List Allocation

Rule: File is allocated with non continuous memory blocks.  
End of each block is a pointer, point to next block

File	Pos	End
abc	12	3
def	5	11

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	3	-1	0	6	8	14	9	11	7	-1	10	0	15	2	0	0



HUST

HUST



#### Linked List Allocation

Rule: File is allocated with non continuous memory blocks.  
End of each block is a pointer, point to next block

File	Pos	End
abc	12	3
def	5	11

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	3	-1	0	6	8	14	9	11	7	-1	10	0	15	2	0	0

#### Continuous Allocation

Rule: File is allocated with continuous memory block,  
End of each block is a pointer, point to next block

File	Pos	Size
file-1	1	19
file-2	2	18

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	3	-1	0	6	8	14	9	11	7	-1	10	0	15	2	0	0

#### Continuous Allocation

Rule: File has the length  $n$  and begin at block  $b$  will occupy blocks  $b, b+1, \dots, b+n-1$

• Two block  $b$  and  $b+1$  are continuous

⇒ No need to move the header (except the last sector)

⇒ High speed access

• Allow directly access block  $i$  of file

⇒ access block  $b+i-1$  on storage device

• Select empty space when there are request?

• Strategies First-Fit /Worst Fit /Best Fit

• External fragmentation phenomenon

• Difficult when the size of file increase

HUST

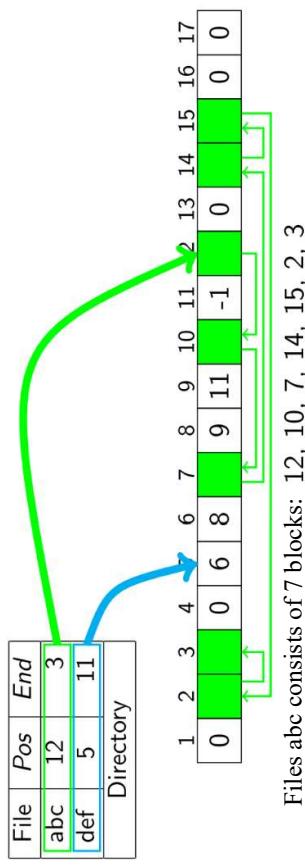
HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Linked List Allocation

Rule: File is allocated with non continuous memory blocks.  
End of each block is a pointer, point to next block



Files abc consists of 7 blocks: 12, 10, 7, 14, 15, 2, 3

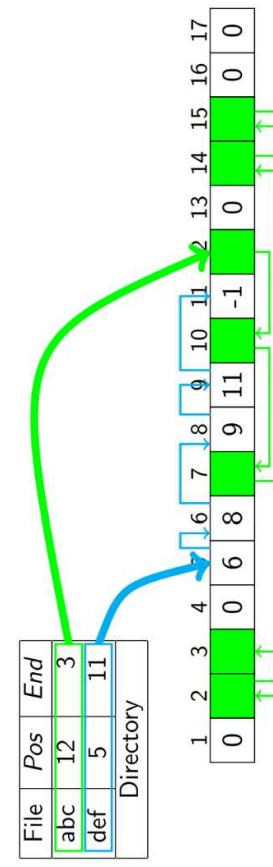
HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Linked List Allocation

Rule: File is allocated with non continuous memory blocks.  
End of each block is a pointer, point to next block



Files abc consists of 7 blocks: 12, 10, 7, 14, 15, 2, 3

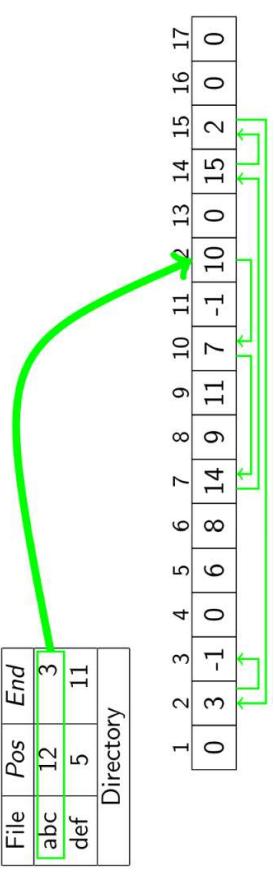
HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Linked List Allocation

Rule: File is allocated with non continuous memory blocks.  
End of each block is a pointer, point to next block



Files abc consists of 7 blocks: 12, 6, 8, 9, 11, 13, 15, 17, 0

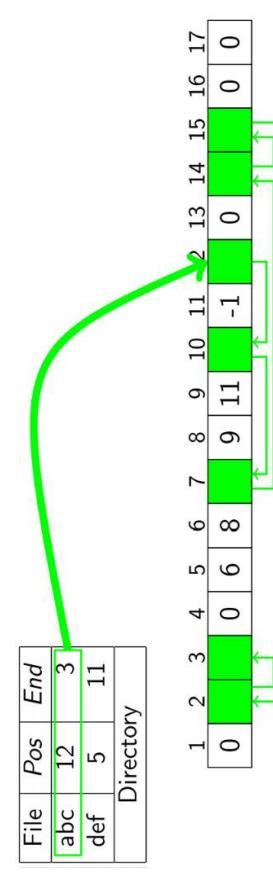
HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Linked List Allocation

Rule: File is allocated with non continuous memory blocks.  
End of each block is a pointer, point to next block



Files abc consists of 7 blocks: 12, 10, 7, 14, 15, 2, 3

HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Linked List Allocation (cont.)

- Apply: FAT
  - Utilized as a linked list
  - Combined of many members, each member corresponding to a block
  - Each member in FAT, contain next block of file
  - Last block has special value (FFFF)
  - Error block has value (FFF7)
  - Unused block has value (0)
  - Location field in file record contains the first block of file

HUST

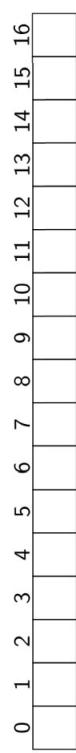
## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Index allocation

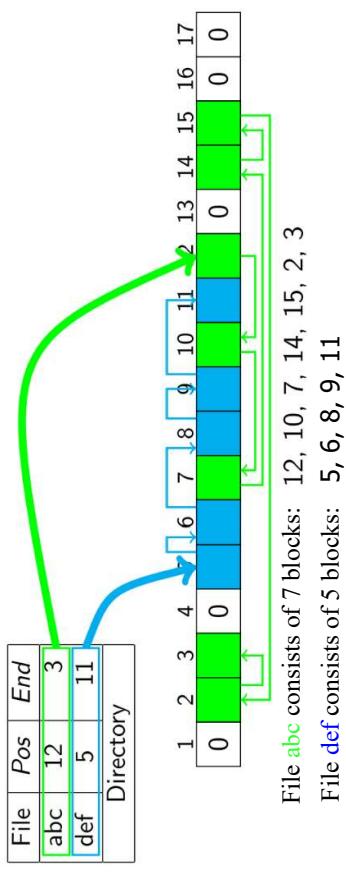
- Rule: Each file has a main index block which contains list of file block

File	Index block
abc	5
def	12



#### Linked List Allocation

Rule: File is allocated with non continuous memory blocks.  
End of each block is a pointer, point to next block



HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Linked List Allocation

- Only effective for sequent accessing file
- To access block n, must traverse n - 1 blocks before it
  - Blocks are noncontiguous, has to relocation from start
  - Slowly access speed
  - Blocks in file are linked by pointers -> if pointers broke?
    - Lost data due to link to block is lost
    - Link to block without data or block belongs to other file
- Solution: Apply many pointers in each block ⇒ Consume memory

HUST

## 2. File system's implementation

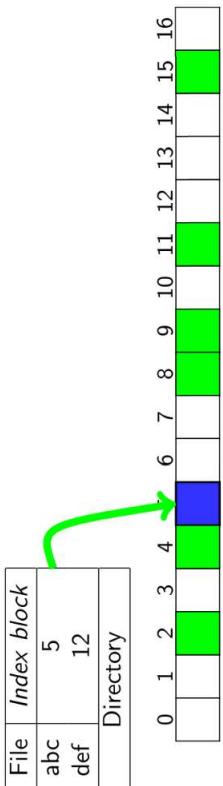
### 2.2. Storage area allocation methods

## 2. File system's implementation

### 2.2. Storage area allocation methods

Index allocation

Rule: Each file has a main index block which contains list of file block

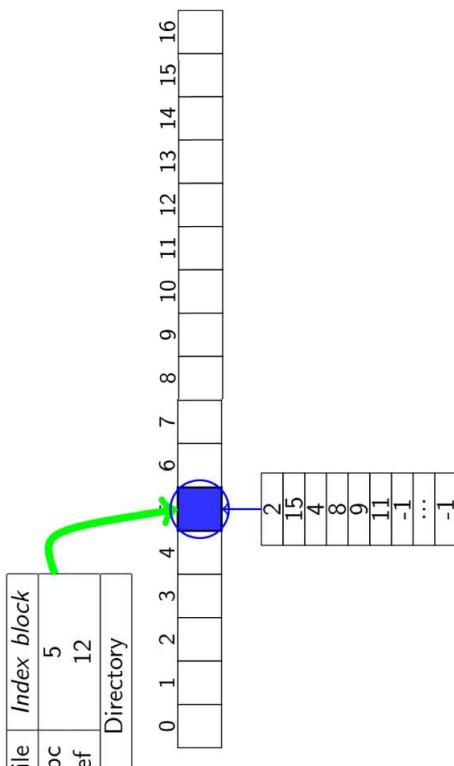


Files abc consists of 6 blocks: 2, 15, 4, 8, 9, 11

HUST

Index allocation

Rule: Each file has a main index block which contains list of file block



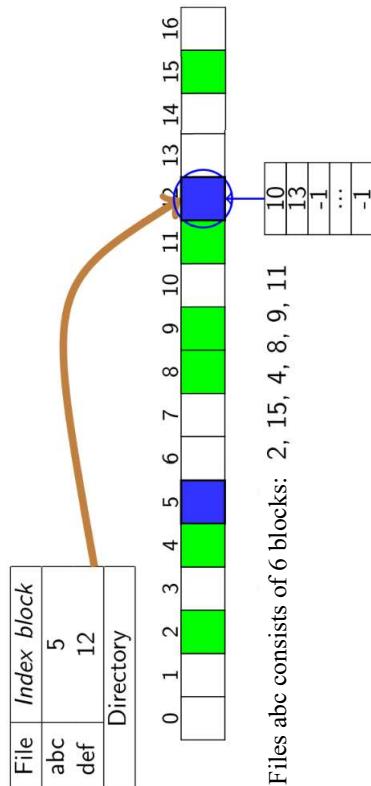
HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

Index allocation

Rule: Each file has a main index block which contains list of file block



Files abc consists of 6 blocks: 2, 15, 4, 8, 9, 11

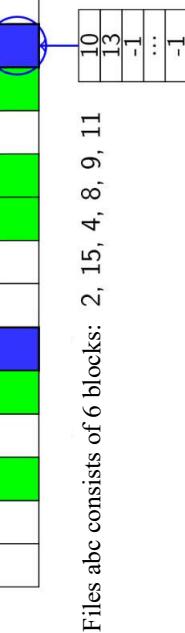
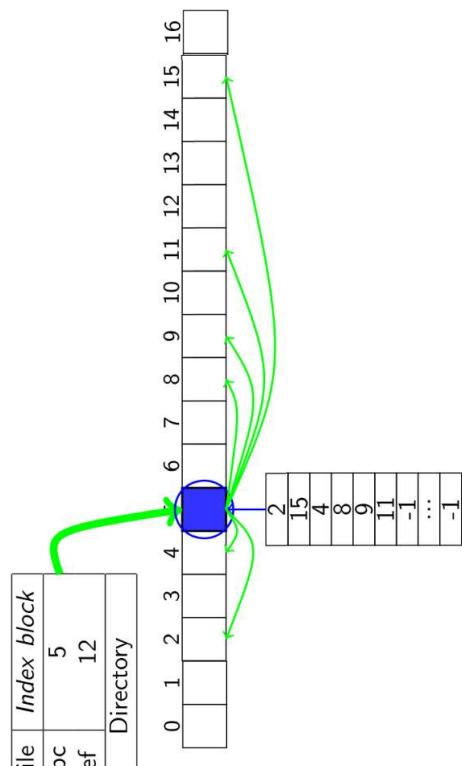
HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

Index allocation

Rule: Each file has a main index block which contains list of file block



Files abc consists of 6 blocks: 2, 15, 4, 8, 9, 11

HUST

## 2. File system's implementation

### 2.2. Storage area allocation methods

## 2. File system's implementation

### 2.2. Storage area allocation methods

#### Index allocation

file's large size storing problem.

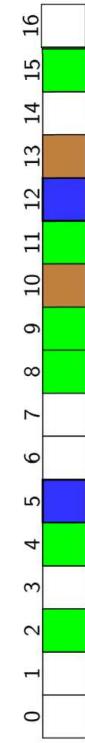
- Linked map
  - Connects index block
  - Last member of index block point to other index block if it's necessary
- Multi level Index
  - Use an index block point to other index block

HUST

#### Index allocation

Rule: Each file has a main index block which contains list of file block

File	Index block
abc	5
def	12



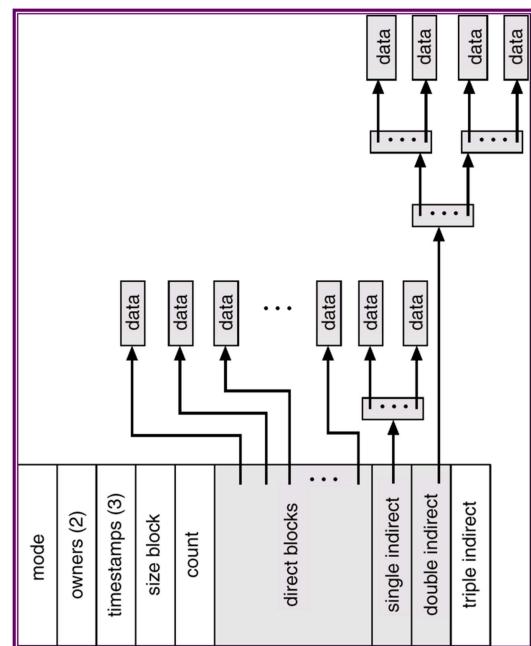
Files abc consists of 6 blocks: 2, 15, 4, 8, 9, 11  
Files def consists of 2 blocks: 10, 13

HUST

#### Link map (UNIX)

## 2. File system's implementation

### 2.2. Storage area allocation methods



HUST

#### Index allocation

## 2. File system's implementation

### 2.2. Storage area allocation methods

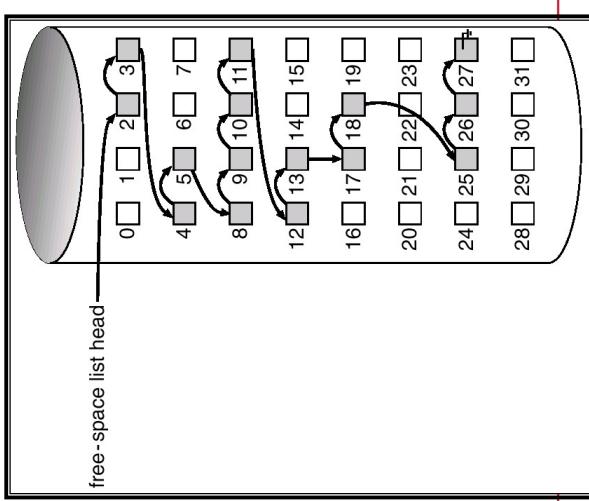
- Member **i** of index block point to block **i** of file
  - Read block **i** use pointer declared at member **i** of index block
  - Create file, members of index block has null value (-1)
  - Require block **i**, block address is allocated, putted into member **i**
- Remark
  - No external fragmentation
  - Allow direct access
- Require index block: file has small size require 2 blocks
  - Block for data
  - Block for index (use 1 member)
- Solution: Reduce block size ⇒ Reduce cost of memory ⇒ file's large size storing problem.

HUST

## 2. File system's implementation

### 2.3. Free storage area management

- Methods
- Link list
    - Hold pointer to first free disk block
    - This memory block contain pointer to next free disk block
    - Not effective when traversing the list



## 2. File system's implementation

### 2.3. Free storage area management

- Directory implementation
- Storage area allocation methods
- Free storage area management

## 2. File system's implementation

### 2.3. Free storage area management

- Methods
- Bit vector
    - Each block represented by 1 bit (1: free; 0: allocated)
    - Easy to find n contiguous memory block
    - Need instruction to work with bit
  - Counting
    - Hold address of n free block
    - n - 1 first free block, block n contain address of next n free block
    - Pros: Fast to find free memory area
  - Grouping
    - Rule: Store address of the first free block and size of contiguous memory area in free-memory-area management list
    - Effective when the counter larger than 1

\* Bit vector ( $n$  blocks)

$$\begin{matrix} & 0 & 1 & 2 & \dots & n-1 \\ \hline \square & \square & \square & \square & \dots & \end{matrix}$$

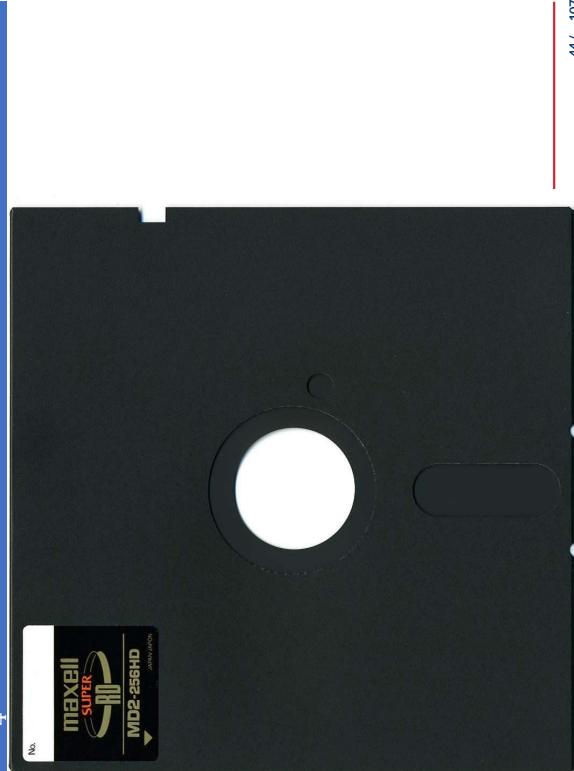
$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$

### 3. Information organization on disk

#### 3.1 Disk's physical structure

### Chapter 4 File system management

Floppy disk  $5\frac{1}{4}$



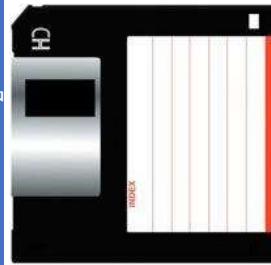
HUST

44 / 107

### 3. Information organization on disk

#### 3.1 Disk's physical structure

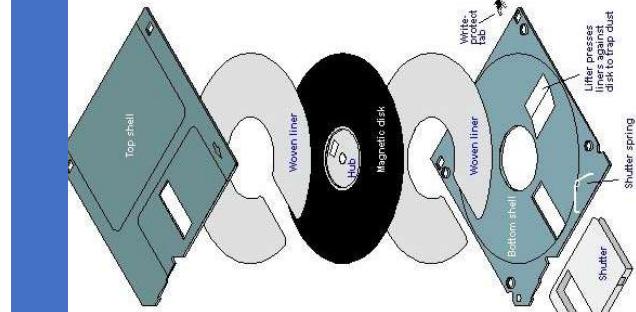
Floppy disk  $3\frac{1}{2}$



HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure



HUST

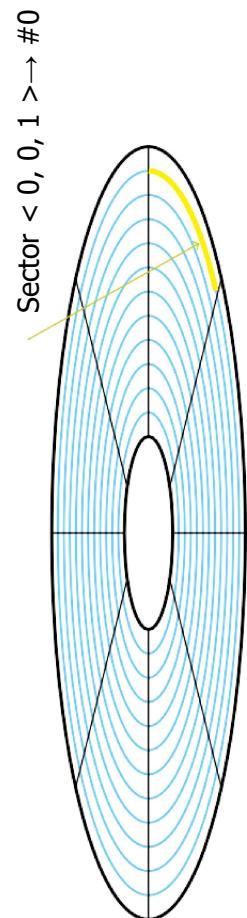
45 / 107

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



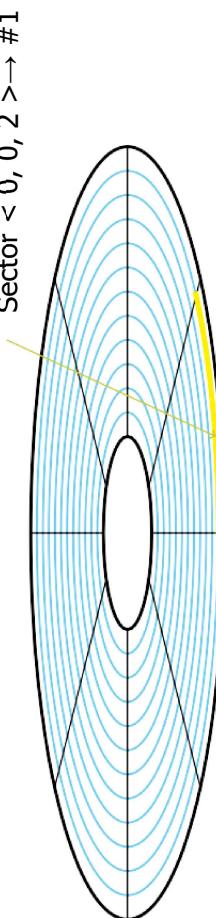
HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

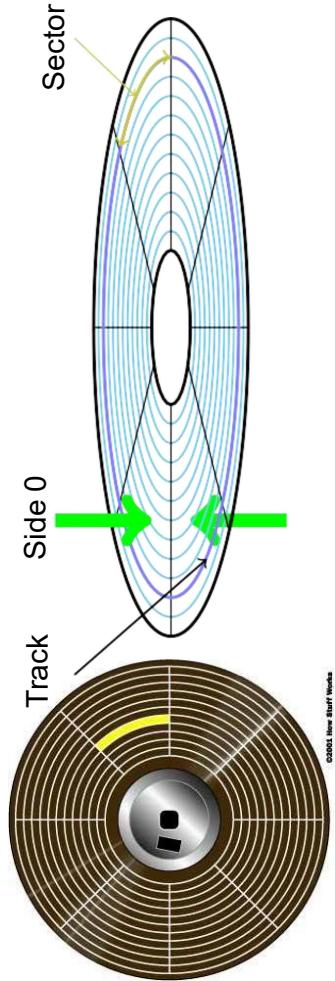


HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Floppy disk's physical structure



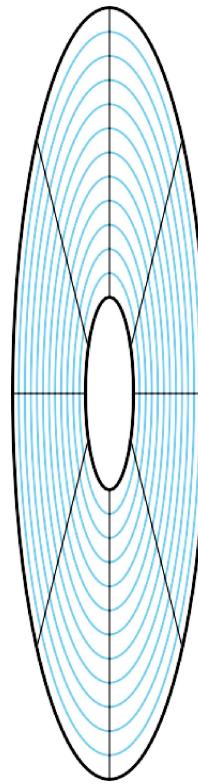
- Track: concentric circles around the disk
  - Numbered 0, 1, ... From outer to inner
- Side: Each side is read by a Header
  - Headers are numbered 0, 1
- Sector
  - Numbered 1, 2, ...

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Information positioning on floppy disk

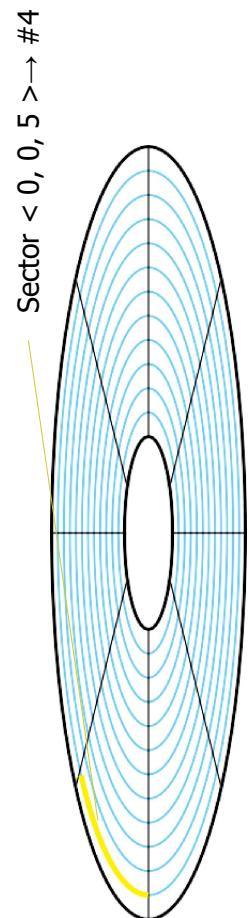
- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



HUST

## Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

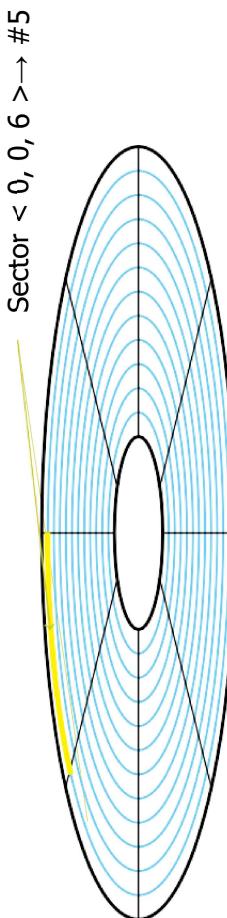


Sector < 0, 0, 5 >→ #5

HUST

## Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

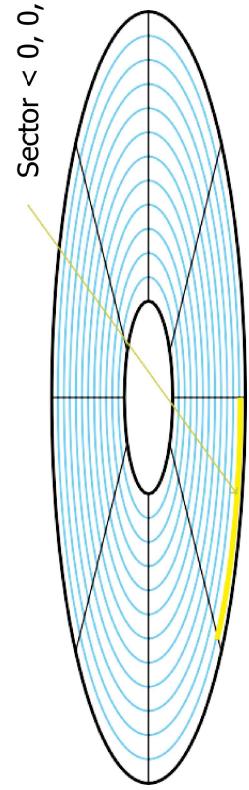


Sector < 0, 0, 6 >→ #5

HUST

## Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

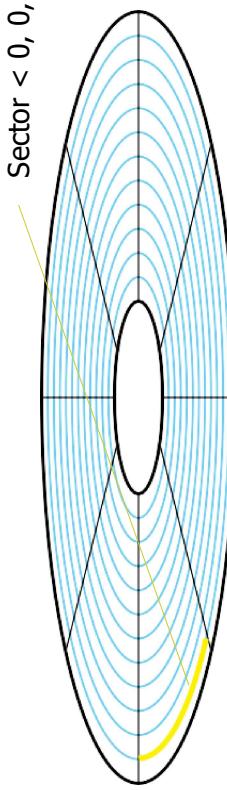


Sector < 0, 0, 2 >→ #2

HUST

## Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Sector < 0, 0, 3 >→ #3

HUST

### 3. Information organization on disk

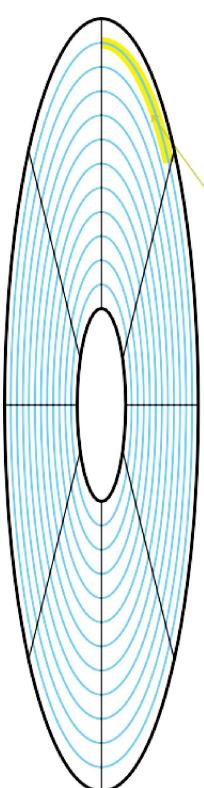
#### 3.1 Disk's physical structure

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Information positioning on floppy disk

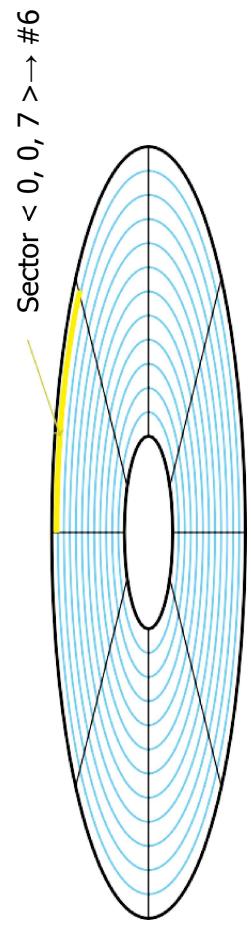
- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



HUST

#### Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



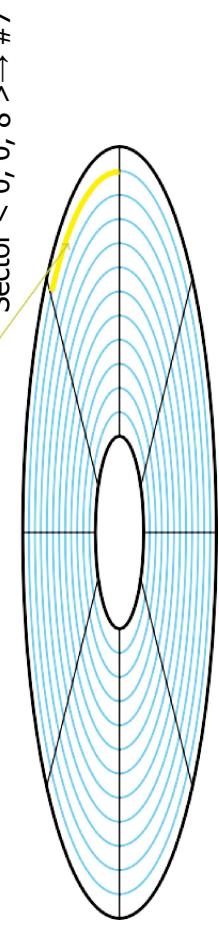
HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



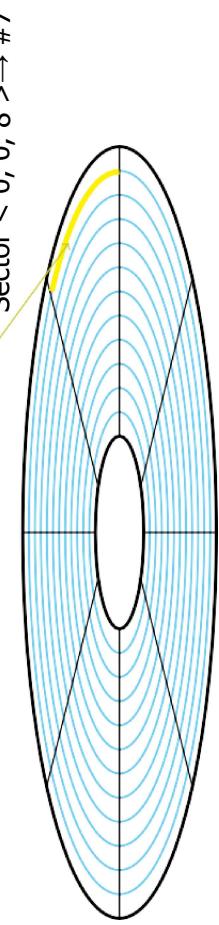
HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure

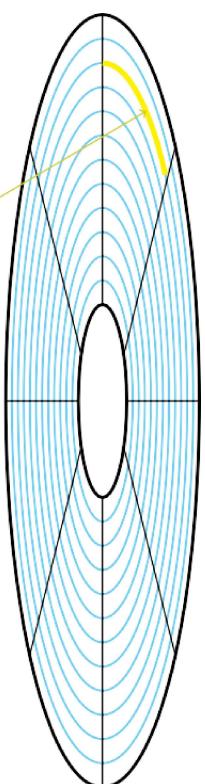
### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

Sector < 0, 1, 1> → #16

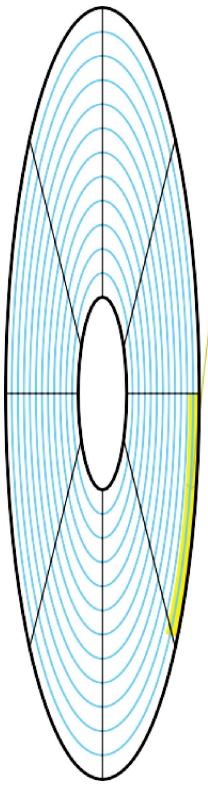


HUST

#### Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

Sector < 1, 0, 3> → #10



HUST

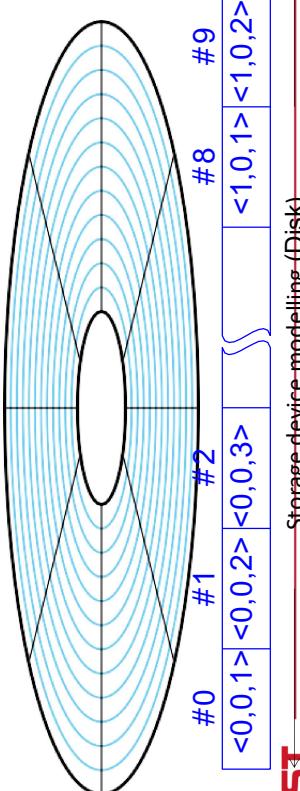
### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3-dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
  - Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

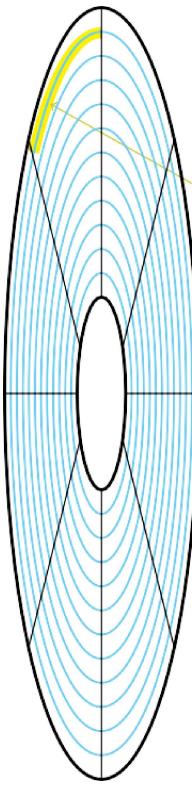
Sector < 1, 0, 3> → #10



Storage-device-modelling (Disk)

HUST

Sector < 1, 0, 8> → #15



HUST

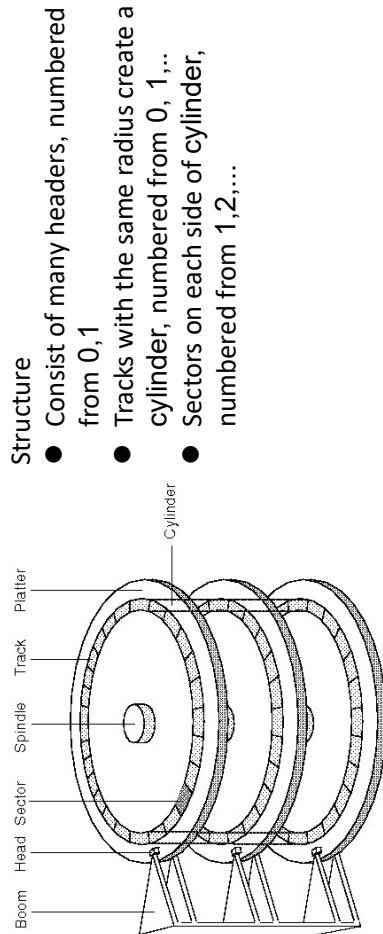
### 3. Information organization on disk

#### 3.1 Disk's physical structure

## Hard disk's physical structure



## Hard disk



Information positioning  
• 3 dimensions (H, C, S)  
• 1 dimension: sector's number

HUST

- Rule similar to floppy disk: Sector→Header→Cylinder

### 3. Information organization on disk

#### 3.1 Disk's physical structure

## Accessing sector on disk

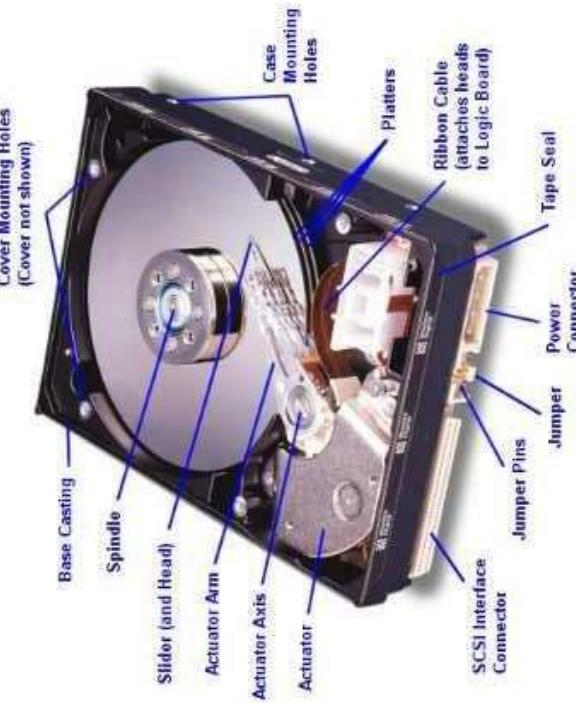
- Sector: information unit to work with disks
- Can access (read/write/format/...) to each sector
- Access via BIOS 13h interrupt (function 2, 3, 5,...)
  - Not depend on operating system
  - Sector is determined by address <H,C,S>
- Access via system call
  - Operating system's interrupt
  - Example: MSDOS provide 25h/26h interrupt allow read/write sectors at linear address
- Use WIN32 API functions
  - CreateFile()/ReadFile()/WriteFile()

HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure

## Hard disk



Cover Mounting Holes  
(Cover not shown)

HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Use WIN32 API

- **HANDLE CreateFile(...)**: Open file/I/O device
  - LPCTSTR lpFileName,  $\Rightarrow$  file/I/O device's name
    - " "?\?\.\.\C :" Partition / Drive C
    - "\\\.\PhysicalDrive0" First hard drive
  - DWORD dwDesiredAccess, $\Rightarrow$  Operation with device
  - DWORD dwShareMode, $\Rightarrow$  Allow sharing
  - LPSECURITY\_ATTRIBUTES lpSecurityAttributes (NULL),
  - DWORD dwCreationDisposition, $\Rightarrow$  Operation to perform
  - DWORD dwFlagsAndAttributes,  $\Rightarrow$  Attribute
  - HANDLE hTemplateFile (NULL)
- **BOOL ReadFile(...)**
  - HANDLE hFile,  $\Rightarrow$  File to read
  - LPVOID lpBuffer,  $\Rightarrow$  Buffer to store data
  - DWORD nNumberOfBytesToRead, $\Rightarrow$ , number of byte to read
  - LPDWORD lpNumberOfBytesRead, $\Rightarrow$  number of read bytes
  - LPOVERLAPPED lpOverlapped (NULL)
- **BOOL WriteFile(...)**  $\Rightarrow$  Parameters similar to ReadFile()

HUST

#### Use Interrupt 13h

Register	Meaning
AH	2h:Read sector; 3h: write Sector
AL	Number of sector to read Sectors must be on the same side, track
DH	Header number
DL	Disk's number. 0h:A; 80h: First hard disk; 81h Second hard disk 2
CH	Track/Cylinder's number (Use 10 bits, borrow 2 high bits from CL)
CL	Sector's number (only use 6 lower bits)
ES:BX	Point to buffer area where data will be read (when AH=2h) or write to disk (when AH=3h)
CarryFlag	CF=0 no error; CL contains number of sector read CF=1 Had error, AH contains error's code

WinXP limit the use on interrupt 13h to direct access  
**HUST**

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Use WIN32 API (Example)

```
#include <windows.h>
#include <stdio.h>
#include <dos.h>
int main(int argc, char *argv[]){
    union REGS regs;
    struct SREGS sregs;
    int Buff[512];
    int i;
    regs.h.ah = 0x02;
    regs.h.al = 0x01;
    regs.h.dh = 0x00;
    regs.h.dl = 0x80;
    regs.h.ch = 0x00;
    regs.h.cl = 0x01;
    regs.x.bx = FP_OFF(Buf);
    sregs.es = FP_SEG(Buf);
    int86x(0x13,&sregs,&sregs);
    for(i=0;i<512;i++) printf("%4X",Buf[i]);
}
return 0;
```

HUST

### 3. Information organization on disk

#### 3.1 Disk's physical structure

#### Use interrupt 13h (Example)

```
#include <stdio.h>
#include <dos.h>
int main(int argc, char *argv[]){
    union REGS regs;
    struct SREGS sregs;
    int Buff[512];
    int i;
    regs.h.ah = 0x02;
    regs.h.al = 0x01;
    regs.h.dh = 0x00;
    regs.h.dl = 0x80;
    regs.h.ch = 0x00;
    regs.h.cl = 0x01;
    regs.x.bx = FP_OFF(Buf);
    sregs.es = FP_SEG(Buf);
    int86x(0x13,&sregs,&sregs);
    for(i=0;i<512;i++) printf("%4X",Buff[i]);
}
return 0;
```

HUST

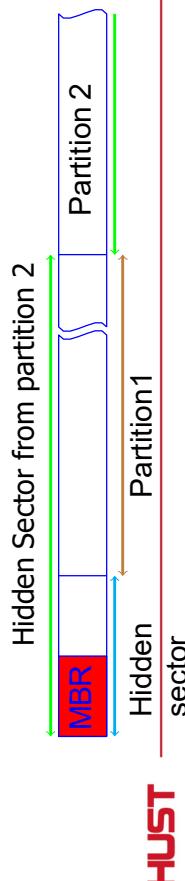
### 3. Information organization on disk

#### 3.2 Disk's logical structure

### 3. Information organization 3.1 Disk's physical structure

## logical structure

- Floppy disk: Each OS has their own management strategy
  - Hard disk (mass storage)
    - Divided into sections (Partitions, Volumes,...)
      - Each section is a set of contiguous Cylinders
      - User can set the size (Example: use fdisk command)
      - Each partition can be managed by an independent OS
      - OS format partition in an usable format
      - Different systems exist: FAT, NTFS, EXT3,...
    - In front of partitions are masked sector
    - Master Boot Record (MBR): Disk's first Sector



3. Information organization on disk  
    3.2 Disk's logical structure

- Disk's most important Sector
  - Disk's first sector (Number 0 or address <0, 0, 1>)
  - Structure consist of 3 parts

Boot strap code	Partition table
	55AA

- Disk's physical structure
- Disk's logical structure

1. Bootstrap code
    - Read partition table to know
      - Position of partitions
      - Active partition (contains OS)
    - Read and execute first sector of active partition
  2. Partition table (64bytes)
    - Consist of 4 elements, each element 16 bytes
      - Element contains information of one partition: Position, size, owned system
  3. OS's signature (always 55AA)

Result

3. Information organization on disk
  - 3.2 Disk's logical structure

- 3. Information organization on disk
  - 3.2 Disk's logical structure

Master Book Record

- Disk's most important Sector
  - Disk's first sector (Number 0 or address <0, 0, 1>)
  - Structure consist of 3 parts

Master Boor Record

- 3. Information organization on disk
  - 3.2 Disk's logical structure

Master Boor Record

הט

### 3. Information organization on disk

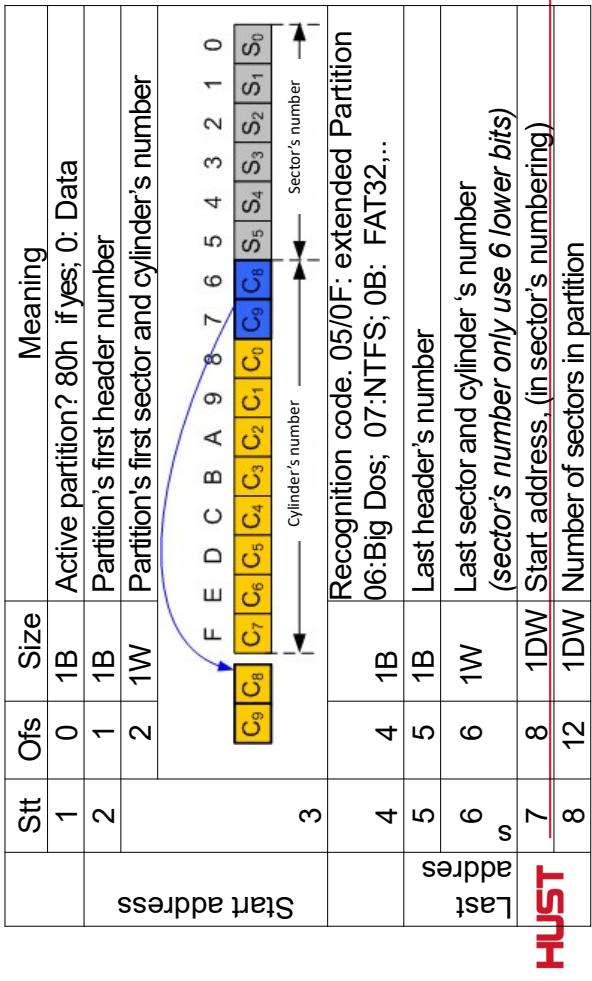
#### 3.2 Disk's logical structure

### 3. Information organization on disk

#### 3.2 Disk's logical structure

### Example 2

Structure of one partition table's element



### 3. Information organization on disk

#### 3.2 Disk's logical structure

## Extended partition

- When recognition code is 05 or 0F -> extended partition
  - Organized as a physical hard disk
    - First Sector is MBR, contain information of partitions inside extended partition
    - Element in extended partition can be an extended partition
    - Allow to create more than 4 logic drive

### 3. Information organization on disk

#### 3.2 Disk's logical structure

## Example 1

Act	Size	Sys#Sector
80	07	
00	0B	
00	0F	
00	0F	

Act	Size	Sys#Sector
00	0B	
00	05	

Partition table



Decode	Start position	End position			#sector	Number of sector
		Hdr	Cyl	Sec		
Boot						
No	1	0	1	254	248	63
Yes	0	249	1	254	560	4000185
No	0	747	1	254	1023	12000555
-	0	0	0	0	0	8000370
					0	0

### 3. Information organization on disk

#### 3.2 Disk's logical structure

### 3. Information organization on disk

#### 3.2 Disk's logical structure

### Example of extended partition 3

Extended Partition (Sector number 57546720)						
	Begin	End	Sys	Relative	Number Of Sector	
Active	Hdr	Cyl	Sct	Hdr	Cyl	Sct
NO	1	1023	1	0B	239	1023
NO	0	0	0	00	0	0
NO	0	0	0	00	0	0
NO	0	0	0	00	0	0
NO	0	0	0	00	0	0
55	AA					

## Example of extended partition 1

80	01	01	00	07	EF	FF	3F	00	00	11	2F	F7
00	00	C1	FF	0F	EF	FF	5F	00	00	B0	23	B1
00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00
55	AA											
				Begin			End		Relative		Number	
Active	Hdr	Cyl	Sct	Sys	Hdr	Cyl	Sct	Sect	Sector		Of	Sector
YES	1	0	1	07	239	1023	63	63	63	32976657		
NO	0	1023	1	0F	239	1023	63	63	63	32976720		
NO	0	0	0	0	00	0	0	0	0	0	45163440	
NO	0	0	0	0	00	0	0	0	0	0	0	0
NO	0	0	0	0	00	0	0	0	0	0	0	0

Chapter 4 File system management

## ① File system

- ② File system's implementation
  - ③ Information organization on disk
  - ④ **FAT system**

### **3. Information organization on disk**

Extended Partition (Sector number 329976720)...

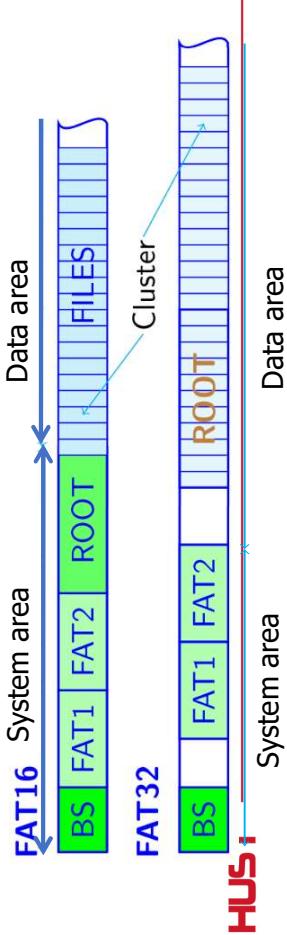
## Example of extended partition 2

Partitioning structure for FAT

FAT12/16

- Maximum number of cluster FAT12:  $2^{12} - 18$ ; FAT16 :  $2^{16} - 18$
  - Max size: FAT12: 32MB; FAT16: 2GB/4GB (32K/64K Cluster)
  - Only use 28 bits  $\Rightarrow$  Maximum Number of cluster:  $2^{28} - 18$
  - Max size: 2TB/8GB/16TB (8KB/32KB/64KB Cluster)

FAT's logical structure



Chapter 4: File system management  
4. FAT system

Structures

Jmp xx	Parameter Table	Boot strap loader	55AA
--------	-----------------	-------------------	------

- Partition's First Sector
  - Structure consists of 3 parts
    - BPB: Bios Parameter Block
    - Boot strap loader
  - System's signature (always 55AA)





Number of ROOT's elements

There are maximum 512 elements in Root's directory

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00
02	00	02	00	00	FF	FF	00	3F	00	00	00	00	00	00
C1	EB	01	00	00	00	29	A6	E4	70	4F	4F	00	00	00
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33

HUST

Number of sector before FAT

6 sectors before the first FAT

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00
02	00	02	00	00	FF	FF	00	3F	00	00	00	00	00	00
C1	EB	01	00	00	00	29	A6	E4	70	4F	4F	00	00	00
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33

HUST

Total sector on disk

Disk larger than 32MB

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00
02	00	02	00	00	FF	FF	00	3F	00	00	00	00	00	00
C1	EB	01	00	00	00	00	29	A6	E4	70	4F	4F	00	00
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33

HUST

There are 2 FAT

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00
02	00	02	00	00	FF	FF	00	3F	00	00	00	00	00	00
C1	EB	01	00	00	00	00	29	A6	E4	70	4F	4F	00	00
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33

HUST

Number of sector for 1 track: 63

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00
02	00	02	00	00	FF	00	FF	00	3F	00	FF	00	3F	00
C1	EB	01	00	00	00	00	29	A6	D4	70	4E	4F	20	4E
4D	45	20	20	20	20	46	41	54	31	36	20	20	33	09

HUST

Disk format code: F8

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00	
02	00	02	00	00	00	FF	00	FF	00	3F	00	FF	00	3F	00
C1	EB	01	00	00	00	00	00	00	00	29	A6	D4	70	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	33	09	

HUST

Number of headers: 255

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00
02	00	02	00	00	FF	00	FF	00	3F	00	FF	00	3F	00
C1	EB	01	00	00	00	00	29	A6	D4	70	4E	4F	20	4E
4D	45	20	20	20	20	46	41	54	31	36	20	20	33	09

HUST

Number of sector for 1 FAT:245

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00	
02	00	02	00	00	00	FF	00	FF	00	3F	00	FF	00	3F	00
C1	EB	01	00	00	00	00	00	00	00	29	A6	D4	70	4E	
4D	45	20	20	20	20	46	41	54	31	36	20	20	33	09	

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00	
02	00	02	00	00	00	FF	00	FF	00	3F	00	FF	00	3F	00
C1	EB	01	00	00	00	00	00	00	00	29	A6	D4	70	4E	
4D	45	20	20	20	20	46	41	54	31	36	20	20	33	09	

HUST

HUST



## FAT16's disk parameter decoding example

FAT Type: FAT16

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00	00
02	00	02	00	00	FF	55	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	E4	D4	70	4E	4F	70	4E	41
4D	45	20	20	20	46	41	54	31	36	20	20	46	41	31	09

HUST

Volume serial number: 70D4-EAA6

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00	00
02	00	02	00	00	FF	55	00	00	F8	F5	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	E4	D4	70	4E	4F	70	4E	41
4D	45	20	20	20	46	41	54	31	36	20	20	46	41	31	09

HUST

FAT16's disk parameter decoding example

Chapter 4: File system management

4. FAT system

4.1 Boot sector

FAT16's disk parameter decoding example

Start of boot strap loader

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00	00
02	00	02	00	00	FF	55	00	00	F8	F5	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	E4	D4	70	4E	4F	70	4E	41
4D	45	20	20	20	46	41	54	31	36	20	20	46	41	31	09

Disk label: NO NAME

EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	06	00	00
02	00	02	00	00	FF	55	00	00	F8	F5	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	E4	D4	70	4E	4F	70	4E	41
4D	45	20	20	20	46	41	54	31	36	20	20	46	41	31	09

HUST



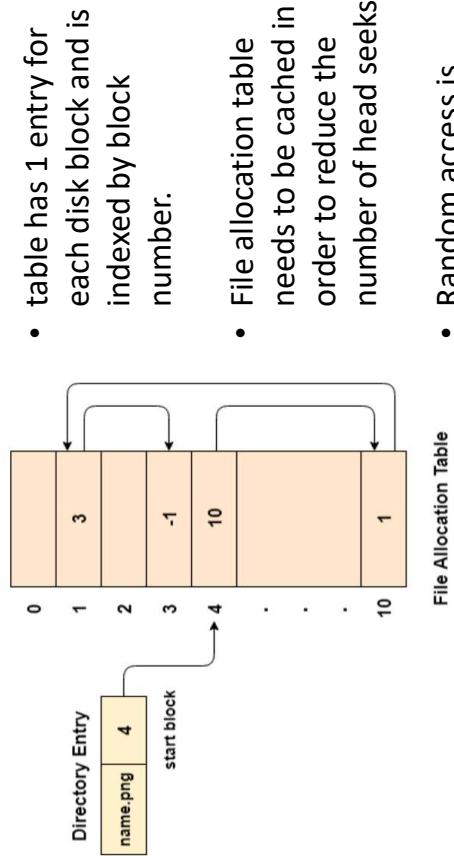


## Implementation

Each element of FAT contain a value represents the property of corresponding cluster

FAT[32]16 12	Meaning
[0000]0 000h	Cluster is free
[0000]0 001h	Unused value
[0000]0 002h →[(0FFF)F]FFEfh	Cluster being used. Value is a pointer, point to next cluster of file
[0FFF]F FF0h →[(0FFF)F]FF6h	Reserved value, unused
[0FFF]F FF7h →[(0FFF)F]FF8h→	Mark corresponding cluster is broken
[0FFF]F FFFh →[(0FFF)F]FFFh	Cluster is being used and it's the last cluster of file (EOC:End Of Cluster chain). Usually value: [0FFF]FFFh

HUST



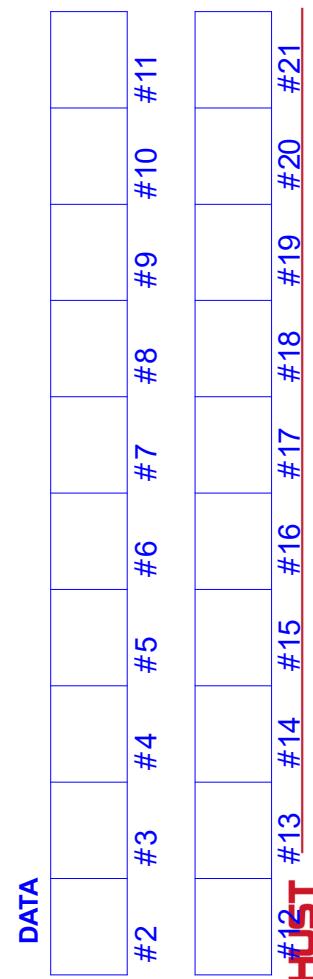
- table has 1 entry for each disk block and is indexed by block number.
- File allocation table needs to be cached in order to reduce the number of head seeks.
- Random access is accomplished.

HUST

## Cluster linking

### Root entry

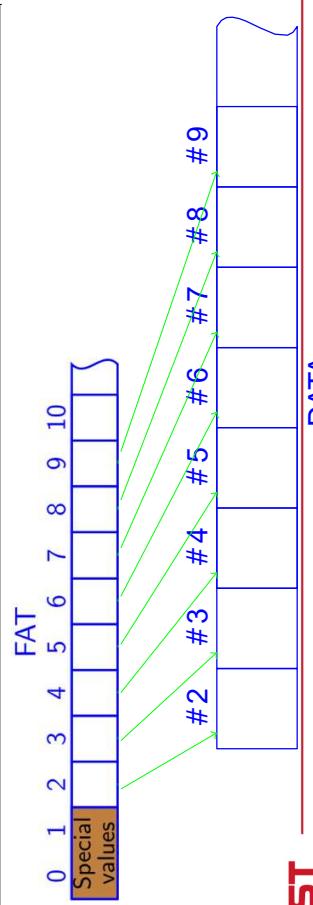
ABC	TXT	A	Time	Date	008	Size
FAT	2	3	4	5	6	7
Special Values	03	04	05	FFF	00	09



HUST

## Method

- FAT include many elements
- Each element can be 12bit, 16bit, 32bit
- Each element corresponds to 1 block (cluster) on the data area
- First 2 elements (0,1) has special meaning
  - Disk format, Bit shutdown, Bit diskerror
  - Element # 2 corresponding to first cluster of the Data



HUST

## Chapter 4: File system management

### 4.1 FAT system

## Chapter 4: File system management

### 4.2 FAT

#### Cluster linking

#### Root entry

ABC	TXT	A	Time	Date	008	Size
#2	#3	#4	#5	#6	#7	#8
HUST	#13	#14	#15	#16	#17	#18

FAT	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Special Values	03	04	05	FFF	00	00	09	10	12	FFF	15	00	00	00	00	00	00	00

#### DATA

#2	#3	#4	#5	#6	#7	#8	#9	#10	#11									
HUST	#13	#14	#15	#16	#17	#18	#19	#20	#21									

#### DATA

FAT	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Special Values	03	04	05	FFF	00	00	09	10	12	FFF	15	00	00	00	00	00	00	00

## Chapter 4: File system management

### 4.2 FAT

#### Cluster linking

#### Root entry

ABC	TXT	A	Time	Date	008	Size
#2	#3	#4	#5	#6	#7	#8
HUST	#13	#14	#15	#16	#17	#18

FAT	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Special Values	03	04	05	FFF	00	00	09	10	12	FFF	15	00	00	00	00	00	00	00

#### DATA

#2	#3	#4	#5	#6	#7	#8	#9	#10	#11									
HUST	#13	#14	#15	#16	#17	#18	#19	#20	#21									

FAT	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Special Values	03	04	05	FFF	00	00	09	10	12	FFF	15	00	00	00	00	00	00	00

#### DATA

#2	#3	#4	#5	#6	#7	#8	#9	#10	#11									
HUST	#13	#14	#15	#16	#17	#18	#19	#20	#21									

## Chapter 4: File system management

### 4.2 FAT

## Chapter 4: File system management

### 4.2 FAT

#### Root entry

ABC	TXT	A	Time	Date	008	Size
#2	#3	#4	#5	#6	#7	#8
HUST	#13	#14	#15	#16	#17	#18

FAT	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Special Values	03	04	05	FFF	00	00	09	10	12	FFF	15	00	00	00	00	00	00	00

#### DATA

#2	#3	#4	#5	#6	#7	#8	#9	#10	#11									
HUST	#13	#14	#15	#16	#17	#18	#19	#20	#21									

FAT	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Special Values	03	04	05	FFF	00	00	09	10	12	FFF	15	00	00	00	00	00	00	00

#### DATA

**Example:** Read one sector of FA132

- Boot sector
  - FAT (File Allocation Table)
  - Root directory

HUST

Chapter 4: File system management  
4. FAT system

- Table consists of **file record**
    - Each record size is 32 bytes
      - Contain information involve one file/directory/ volume label
    - FAT12/FAT16
      - Root directory lie right behind FAT
      - Size = Number of maximum elements in root directory \* 32
    - FAT32
      - Location is defined based on BPB
      - Filed #18: Number of ROOT's first cluster
      - Undefined size
      - Support long file name (LFN: Long File Name)
        - One file may use more than 1 element

四五

Ex

**Example:** Read one sector of FA132

- ```

BYTE Buf[512]; DWORD hDisk;
WORD FATAddr; DWORD byteread, i;
FILE_SHARE_READ|FILE_SHARE_WRITE, NULL,
hDisk = CreateFile("\\\\.\\"F:", GENERIC_READ,
OPEN_EXISTING, 0,NULL);
ReadFile(hDisk,Buf,512,&byteread,NULL);
ReadFile(hDisk,FAT[14],2); //Offset 14 Sector truc FAT
memcp(&FATAddr,&Buf[14],2);
SetfilePointer(hDisk,FATAddr * 512, NULL,FILE_BEGIN);
ReadFile(hDisk,FAT,512,&byteread,NULL);
for(i=0;i<128;i++) printf("%08X ",FAT[i]);
CloseHandle(hDisk);
return 0;

```

הנִ  
בְּ

## Example: First Sector of a FAT32



四

## Structure of an element : file name

- First character may contain special meaning (**continues**)
  - 2Eh2Eh (character ".") : Parent directory of current directory
  - Cluster number field point to parent directory
  - If parent is root, #cluster start by zero (FAT12/16)
  - Sub directory lie in **Data area**, manage similar to a file => **File of file record**
  - FAT12/16: Root directory is at a defined position; FAT32: Root directory lies in **Data area**

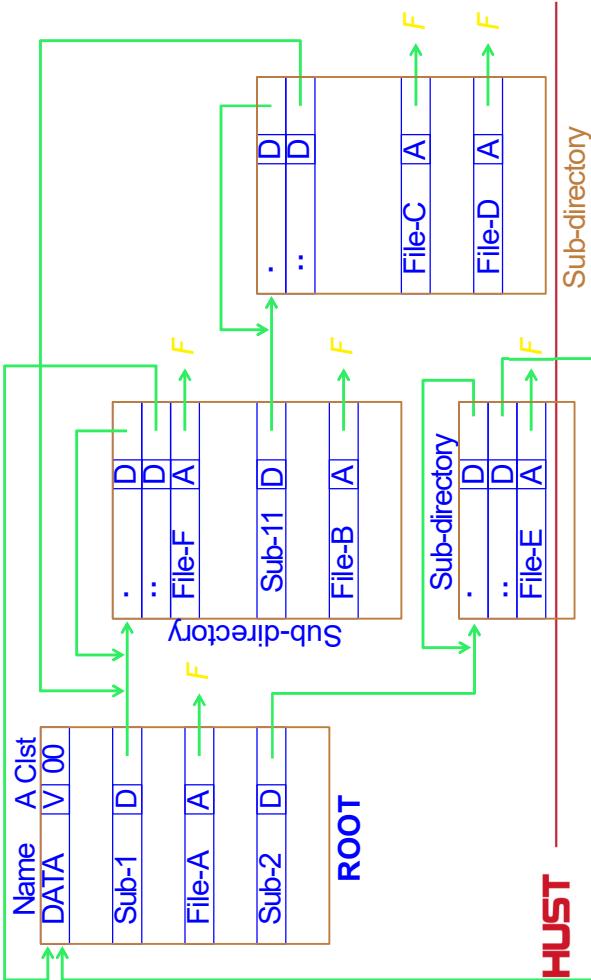
**HUST**

| Stt | Ofs | Size | Meaning                                              |
|-----|-----|------|------------------------------------------------------|
| 1   | 0   | 8B   | File name                                            |
| 2   | 8   | 3B   | Extension                                            |
| 3   | 11  | 1B   | File's attribute                                     |
| 4   | 12  | 10B  | Not used in FAT12/FAT16. Used in FAT32 Reserved      |
| 4.1 | 12  | 1B   | File created time, unit 10ms                         |
| 4.2 | 13  | 1B   | File created time (hour - minute - second )          |
| 4.3 | 14  | 1W   | File created date                                    |
| 4.4 | 16  | 1W   | Last access date                                     |
| 4.5 | 18  | 1W   | Number of starting cluster of file(FAT32: High part) |
| 4.6 | 20  | 1W   | Last update time                                     |
| 5   | 22  | 1W   | Last update date                                     |
| 6   | 24  | 1W   | Number of starting cluster of file (FAT32: Low part) |
| 7   | 26  | 1W   | Size in byte                                         |
| 8   | 28  | 1DW  |                                                      |

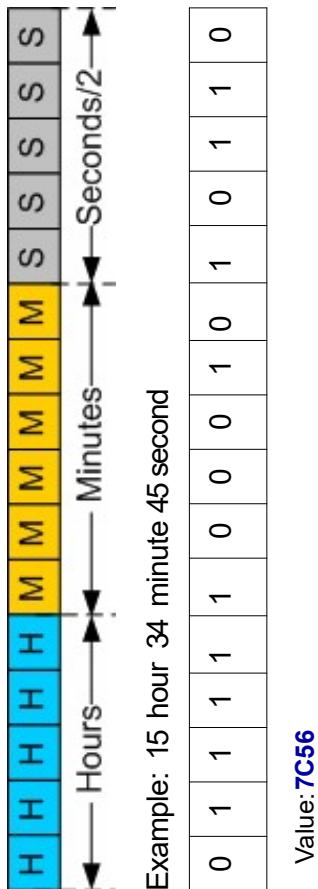
## Sub directory

## Structure of an element : file name

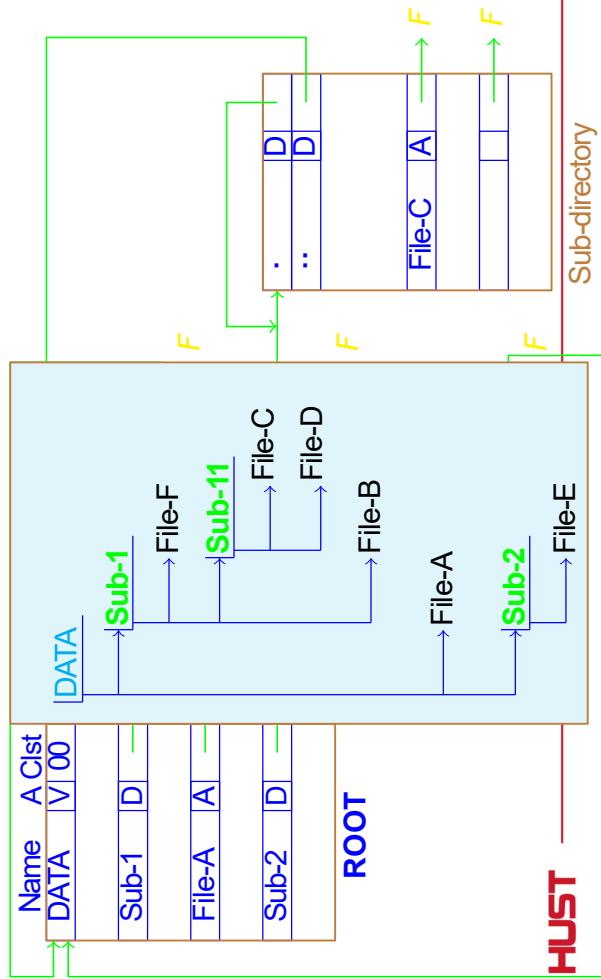
- Sequence of ASCII contain file name
- Not accept space in file name
  - Command copy, del,... do not recognize name contains space
- If smaller than 8 character, insert space character for sufficient 8 character
  - First character may contain special meaning
  - 00h: First character of not used part
  - E5h (character "δ"): File corresponding to this element was deleted
  - 2Eh (character ".") : Sub directory
    - cluster number field point to itself
    - Structure of sub directory is similar to root directory:

**HUST**

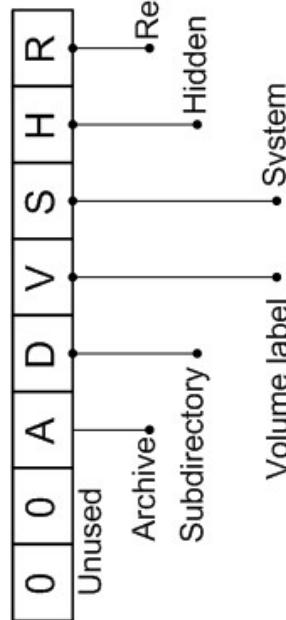
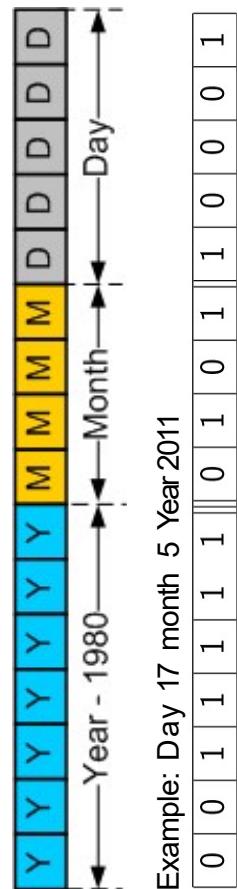
## Structure of an element



## Sub directory



## Structure of an element : Date



Example: Byte attribute **0Fh**: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

  
⇒ File has attribute **Volume label+System+Hidden+ReadOnly**

**Note:** Attribute byte's value **0xF** is not used in MS-DOS ⇒ For marking the **Long File Name** element

## Example: Content of ROOT

| Example: A sector of ROOT |    |    |    |    |    |    |    |    |    |    |
|---------------------------|----|----|----|----|----|----|----|----|----|----|
| 44                        | 41 | 54 | 41 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 45                        | 00 | 00 | 00 | 00 | 64 | 25 | 45 | 00 | 00 | 00 |
| E5                        | 44 | 48 | 20 | 20 | 20 | 20 | 44 | 46 | 20 | 20 |
| 05                        | 3E | A5 | 3E | 00 | 00 | 00 | 05 | 3E | 03 | 00 |
| 45                        | 41 | 45 | 00 | 78 | 00 | 65 | 6D | 00 | 00 | 38 |
| 65                        | 00 | 73 | 00 | 00 | FF | FF | FF | FF | FF | 25 |
| 45                        | 58 | 45 | 4D | 50 | 4C | 45 | 53 | 20 | 20 | 29 |
| 45                        | 3E | A5 | 3E | 00 | 9C | 34 | 3E | 96 | 14 | 00 |
| 42                        | 72 | 00 | 2E | 00 | 63 | 00 | 00 | FF | FF | 00 |
| FF                        | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF |
| 01                        | 52 | 00 | 65 | 00 | 61 | 00 | 64 | 00 | 42 | 43 |
| 6F                        | 00 | 73 | 00 | 53 | 00 | 65 | 00 | 63 | 00 | 74 |
| 52                        | 45 | 41 | 44 | 42 | 49 | 7E | 31 | 43 | 20 | 20 |
| 05                        | 3E | A5 | 3E | 00 | 00 | 76 | 5C | 9A | 3E | 20 |
| 52                        | 45 | 41 | 44 | 4D | 42 | 52 | 20 | 43 | 20 | 20 |
| 41                        | 3E | A5 | 3E | 00 | 00 | 79 | 04 | 40 | 20 | 20 |
| 41                        | 54 | 00 | 65 | 00 | 6D | 00 | 79 | 04 | 2E | BD |
| FF                        | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF |
| 54                        | 45 | 4D | 50 | 53 | 20 | 20 | 20 | 20 | 19 | 00 |
| 95                        | 3E | A5 | 3E | 00 | 00 | 9B | 96 | 45 | 3E | 46 |
| 42                        | 41 | 01 | 6E | 00 | 67 | 00 | 20 | 00 | 34 | 00 |
| 70                        | 00 | 64 | 00 | 66 | 00 | 00 | 00 | 00 | 00 | 00 |
| 01                        | 42 | 00 | E0 | 00 | 62 | 00 | 00 | 00 | 67 | 00 |
| 43                        | 1E | 6E | 00 | 67 | 00 | 20 | 00 | 63 | 00 | 00 |
| 42                        | 41 | 49 | 47 | 49 | 4E | 7E | 31 | 50 | 44 | 46 |
| 95                        | 3E | A5 | 3E | 00 | 00 | 6F | 34 | 51 | 3E | 00 |
| 00                        | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00                        | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00                        | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00                        | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

Chapter 4: File system management  
4. FAT system

## Example: Content of ROOT

```
F:\>DIR
Volume in drive F is DATA
Volume Serial Number is DC27-F353
          Directory of F:\

05/05/2011  06:36 AM <DIR>
04/26/2011  11:35 AM
05/04/2011  03:14 PM <DIR>
05/05/2011  06:52 PM
05/05/2011  06:35 AM <DIR>
```

Exemples  
465 ReadBiosSector.C  
2,749 READMBR.C  
Temps

```
2,696,504 Bài gi?ng chuong 4.pdf
2,699,718 bytes free
14,247,424 bytes free
```

F:\>

## Long File Name (LFN) system

| ofs | size | meaning                                                               |
|-----|------|-----------------------------------------------------------------------|
| 0   | 1B   | Element Order                                                         |
| 1   | 5W   | First 5 Unicode characters                                            |
| 11  | 1B   | Attribute. Mark LFN element<br>Always value 0Fh                       |
| ... |      |                                                                       |
| 12  | 1B   | Reserved (00)                                                         |
| 13  | 1B   | Checksum: Allow check if long file name is corresponding to 8.3 name? |
| 14  | 6W   | unicode character 6,7,8,9,10,11                                       |
| 26  | 1W   | Cluster number. Not used (0000)                                       |
| 28  | 1W   | unicode character 12                                                  |
| 30  | 1W   | unicode character 13                                                  |

Chapter 4: File system management  
4. FAT system

## Long File Name (LFN) system: Ordering field

- Show the order of LFN element
- Each LFN element contain 13 Unicode characters
- First element has value of ordering field: 1
- Last element use bit number 6 to mark
- Unused character has values 0xFF 0x00.
- Only use maximum 20 elements
- After last character is 0x00 0x00.
- Bit number 7 (0x80) show corresponding element is deleted
- Example: file "This is a very long file name.docx"

| entry    | ord  | attr | data          |
|----------|------|------|---------------|
| LFN 3    | 0x43 | 0x0F | ame.docx      |
| LFN 2    | 0x02 | 0x0F | y long file n |
| LFN 1    | 0x01 | 0x0F | This is a ver |
| 8.3 Name |      |      | THISIS~1.DOC  |

HUST



# Hệ Điều Hành

(Nguyên lý các hệ điều hành)

Đỗ Quốc Huy  
 huydq@soict.hust.edu.vn  
 Bộ môn Khoa Học Máy Tính  
 Viện Công Nghệ Thông Tin và Truyền Thông

File: **ReadBiosSector.c**

LFN# 1(ReadBiosSector)

LFN#2 – last LFN (r.c)

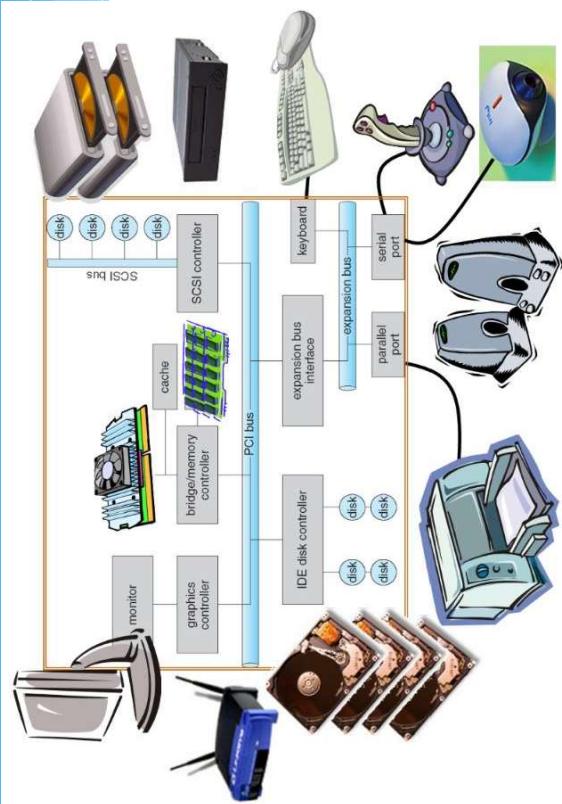
Character after last character(0x00 00)

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 42 | 72 | 00 | 2E | 00 | 63 | 00 | 00 | FF | FF | 0F | 00 | 43 | FF | FF |
| FF | 43 | FF | FF |
| 01 | 52 | 00 | 65 | 00 | 61 | 00 | 64 | 00 | 42 | 00 | 43 | 69 | 00 | 00 |
| 6F | 00 | 73 | 00 | 53 | 00 | 65 | 00 | 63 | 00 | 00 | 74 | 00 | 6F | 00 |
| 52 | 45 | 41 | 44 | 42 | 49 | 7E | 31 | 43 | 20 | 20 | 00 | A6 | B2 | 4B |
| A5 | 3E | A5 | 3E | 00 | 00 | 76 | 5C | 9A | 3E | 3F | 2E | D1 | 01 | 00 |

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 42 | 72 | 00 | 2E | 00 | 63 | 00 | 00 | FF | FF | 0F | 00 | 43 | FF | FF |
| FF | 43 | FF | FF |
| 01 | 52 | 00 | 65 | 00 | 61 | 00 | 64 | 00 | 42 | 00 | 00 | 74 | 00 | 6F |
| 6F | 00 | 73 | 00 | 53 | 00 | 65 | 00 | 63 | 00 | 00 | 00 | 00 | 6F | 00 |
| 52 | 45 | 41 | 44 | 42 | 49 | 7E | 31 | 43 | 20 | 20 | 00 | A6 | B2 | 4B |
| A5 | 3E | A5 | 3E | 00 | 00 | 76 | 5C | 9A | 3E | 3F | 2E | D1 | 01 | 00 |

# Chapter 5 I/O management

## Chapter 5 I/O Management



Chapter 5: IO Management  
1. General management principle  
1.1 Introduction  
IO device

- Diversity, many kinds, different types
- Engineering perspective: device with processor, motor, and other parts
- Programming perspective: Interface like software to receive, execute command and return result
- Categorize
  - **Block** device (disk, magnetic tape)
    - Info mation is stored with fixed size and private address
    - Possible to read/write a block independent from other
      - Operation to locate information exist (seek)
  - **Character** device (printer, keyboard, mouse,..)
    - Accept a stream of character, without block structure
    - No information localization operation
  - Other type: **Clock**

Chapter 5: IO Management  
1. General management principle

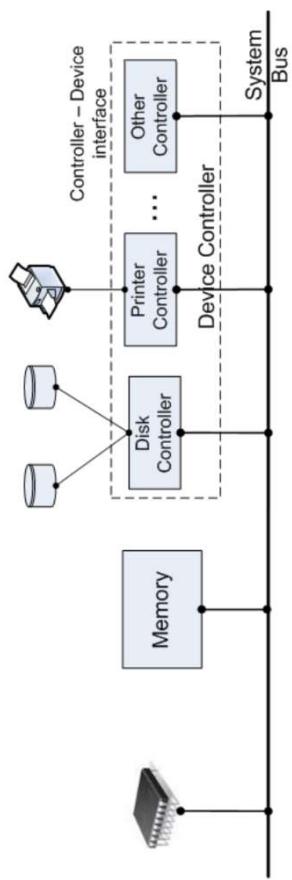
### ● Introduction

### ● Interrupt and Interrupt handle

- Each DC can control 1,2,4,.. peripheral devices
- Depend on the number of connector on the DC
- If the controller **interface is standard** (ANSI, IEEE, ISO,...) -> can connect to different devices



- Each DC has its own register to work with CPU
  - Use special address space for registers: **IO port**



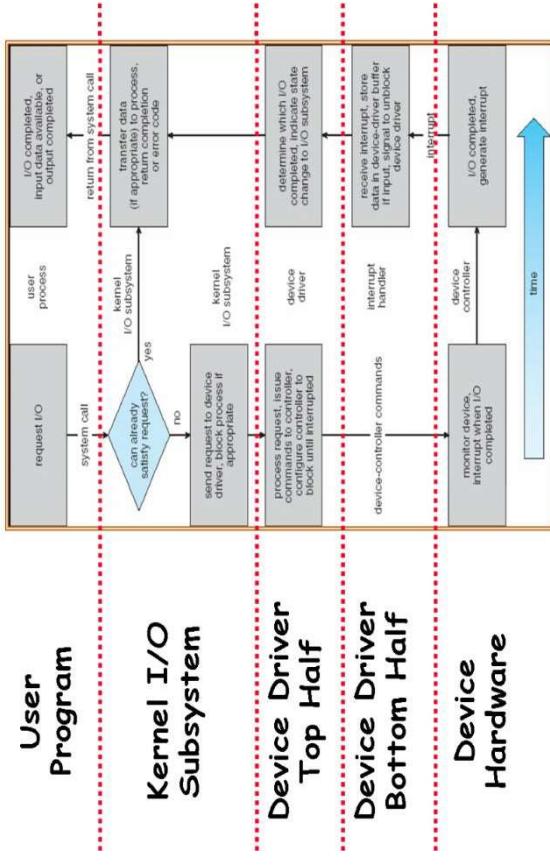
- Peripheral devices are diversity with many types
  - CPU do not know them all => No individual signal for each device
  - Processor do not control device directly
  - Peripheral device is connected to the system via **Device controller (DC)**



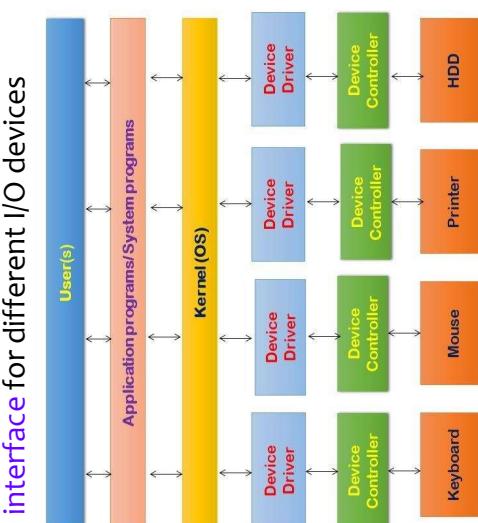
- Electrical circuit attached to the **mainboard's slot**

- Controller and device interface: Low level interface
  - Sector = 512bytes = 4096bits
  - Disk controller must **read/write bits** and **group** them into **sectors**
  - OS **only** work with **controller**
    - Via device's **registers**
    - **Commands and parameters** are putted into controller's **registers**
  - When a **command is accepted** by the controller, **CPU** let the controller work itself and **turn to other job**
  - When command is **finished**, controller **notify** **CPU** via **interrupt signal**
  - CPU take **result** and **device status** via controlling device's **register**

## IO request cycle



- Code segment in system's kernel allow **interactive** with hardware device
- Provide **standard interface** for different I/O devices



## Peripheral device – Operating system interact

- After sending request to device, OS need to acknowledge
  - When device finish request
  - If device has error
  - 2 methods to acknowledge
    - **I/O interrupts**
    - Device generate an interrupt signal to let CPU know IRQ: physical path to **interrupt manager**
  - Map IRQ signal to **interrupt vector**
  - Call to **interrupt handle routine**
- Polling
  - OS timely check device's status register
  - **Waste** checking period if the **I/O operation** is **not frequent**
- Nowadays device can combine 2 methods (E.g. high bandwidth network device)
  - Send interrupt when first packet arrive
  - Pooling next coming packet until the buffer is empty

- Based on Source
  - Internal interrupt
  - External interrupt
- Based on device
  - Hard
  - Soft
- Based on handling ability
  - maskable
  - unmask able
- Based on interrupt moment
  - Request
  - Report

- ① Write characteristic of event caused the interrupt into defined memory area
- ② Save interrupted process 'state'
- ③ Change address of interrupt handle routine to instruction pointer register
  - Utilize interrupt vector table (IBM-PC)
- ④ Run interrupt handle routine
- ⑤ Restore interrupted process
  - **Interrupt >< procedure !?**

● Introduction

● Interrupt and Interrupt handle

Mechanism to help device let the processor know its status

Phenomenon that a process is suddenly stopped, and the system executive other process correspond to an event

● **Interrupt >< procedure !?**

# Chapter 5 I/O Management

- Peripheral device's characteristic: operate slow
  - Active the device
  - Wait for device to get to proper working status
  - Wait for IO operation to be performed
- To Guarantee the **System's performance** -> need to
  - **Reduce** number of **IO operations**, work with **block of data**
  - Perform IO operations **parallelly** with other operations
  - Perform **accessing operation in advance**

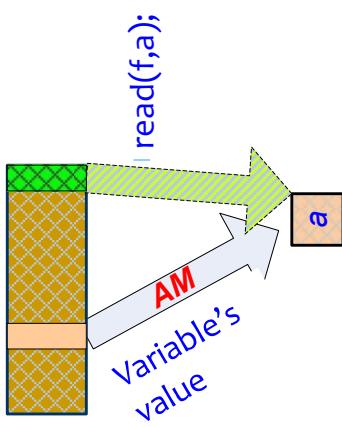
**Buffer:** Intermediate memory area, utilized for storing information during IO operation

- Input buffer
  - Can perform data access command
  - Example: read data from disk
- Output buffer
  - Information is putted into buffer, when buffer's full, buffer content is then written to device

- Buffer
- SPOOL mechanism

## Buffer organization

- Value buffer
- Input buffer
- Output buffer

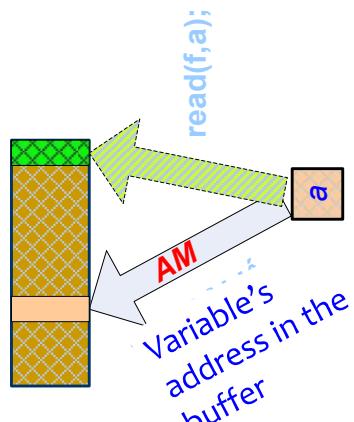


## Buffer classification

- Buffer attached to device
  - Constructed when open device/file
    - **Serve device only**, cleared when device is close
    - Good when devices have different physical record's structures
  - **Buffer attached to system**
    - Constructed when the system start, not attached to a specific device
    - Exist during system working process
      - Open file/device ⇒ attach to already available buffer
      - Close device/file ⇒ buffer returned to system
      - Good for devices have same physical record's structure

## Buffer organization

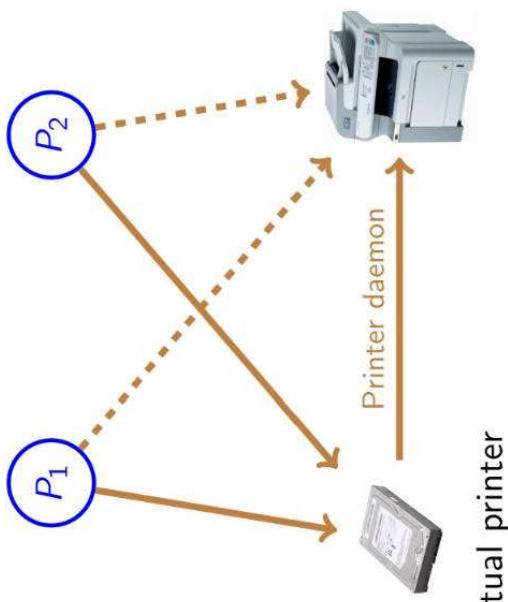
- Processing buffer
  - Input buffer
  - Output buffer
  - Processing buffer
  - Circular buffer
  - Input buffer
  - Output buffer
  - Processing buffer



## Buffer organization

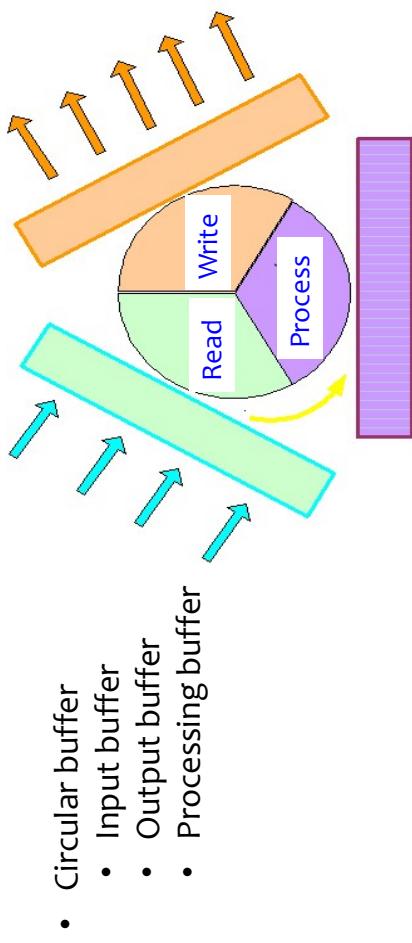
## SPOOL (Simultaneous Peripheral Operation On-line)

- From programming perspective, IO device is
  - Station to receive request from program and perform
  - Return status code to be analyzed by the system
- use software to simulate IO device
  - IO device can be treated as process
  - Synchronized like in process management
- Objective
  - Simulate process of controlling and managing peripheral device
  - Check creating device working status
  - Create parallel effect for sequential device

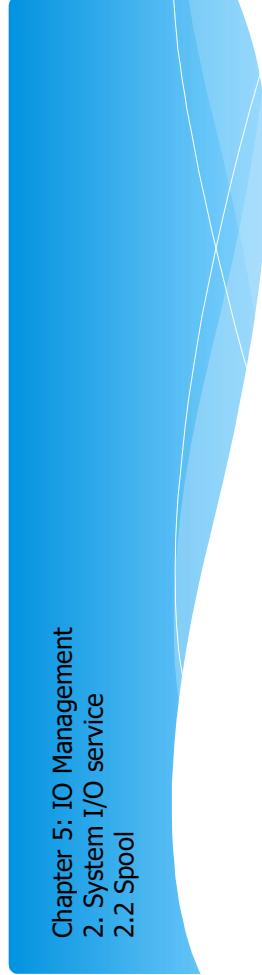


Virtual printer

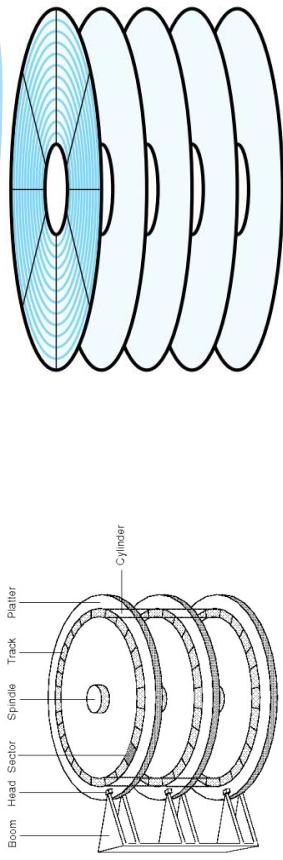
## Buffer organization



- Circular buffer
  - Input buffer
  - Output buffer
  - Processing buffer



# Chapter 5 I/O Management



- ① General management principle
- ② System I/O service
- ③ Disk I/O system

- Modelled as array of logic blocks
  - logic block is the smallest exchange unit
  - Map continuous logic block to disk's sector
  - Block 0 is first sector header 0 outer most track/Cylinder
    - Mapping follow an order: Sector → Header → Track/Cylinder
    - Reading header do not need to move much when read sector next to each other

## Chapter 5: IO Management 3. Disk I/O device 3.1 Disk structure Disk accessing problem

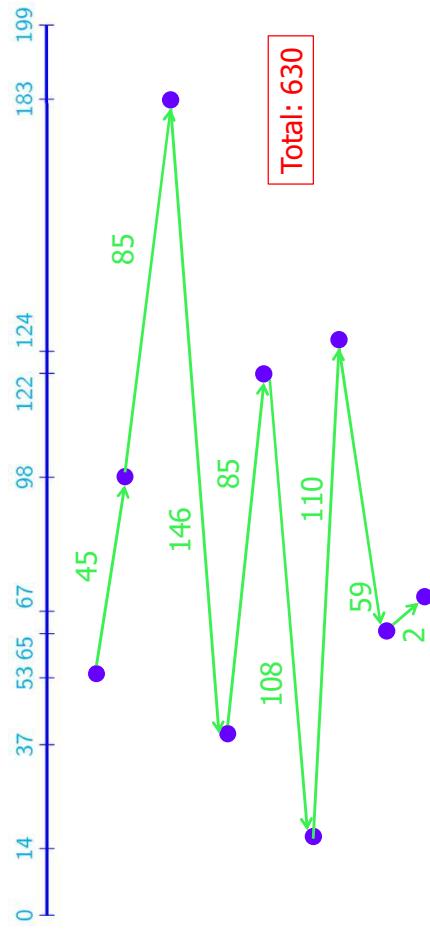
- OS is responded for effectively exploit the hardware
  - For disk: **Fast access time** and **high bandwidth**
  - Bandwidth is calculated based on
    - Total bytes exchanged
    - Time from the **first service request** until the request is **completed**
- Access time consist of 2 parts
  - **Seek time** : Time to move header to cylinders contain required sector
  - **Rotational latency**: Time to wait until disk rotate to required sector

## Chapter 5: IO Management 3. Disk I/O device

- Disk structure
- Disk accessing scheduling

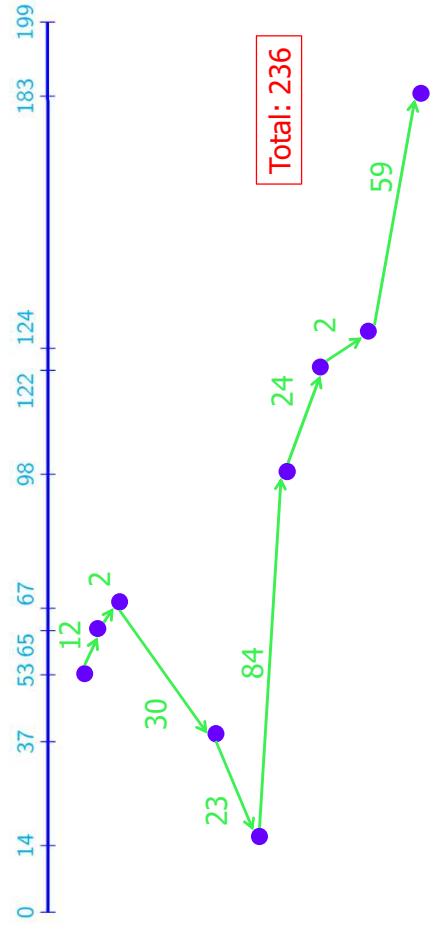
**Access follow the request order  $\Rightarrow$  Not effective**

Accessing requests 98, 183, 37, 122, 14, 124, 65, 67



Select access has smallest seek time from current position  $\Rightarrow$  A request may wait forever if new appearing requests closer to header (similar to SJF)

Accessing requests 98, 183, 37, 122, 14, 124, 65, 67



- Assumption
  - Accessing requests 98, 183, 37, 122, 14, 124, 65, 67
  - Header current position at cylinder 53
- Disk structure
- Disk accessing scheduling

### C-SCAN

Header move from outermost cylinder to innermost cylinder and return. Serve request met on the way



### SCAN

Header move from outermost cylinder to innermost cylinder and return. Serve request met on the way



### LOOK/ C-LOOK

SCAN/C-SCAN's version: Header does not move to outermost/innermost cylinders, only to farthest request at 2 sides and return



### C-SCAN

Principle: Treat cylinders like a circular linked list: Outer most Cylinder connect with innermost cylinder

- Header move from outermost cylinder to innermost cylinder
  - Serve request met on the way
  - When inner most Cylinder is reached, return to outermost Cylinder
    - Do not serve request met on the way
- Remark: Retrieve more equal waiting time than SCAN
  - When header reach to one side of disk (innermost/outermost cylinders), density of requests appear at other side will be higher than current place (reason: header just passing by). This request need to wait longer => Return to other side immediately

- SSTF: More popular, more efficient than FCFS
- SCAN/C-SCAN works better for systems with lots of disk access requests
  - No starvation problem: “Queue too long”
- The efficiency of the algorithms depends on the number and type of requests

- Disk access requests are affected by **disk allocation methods** for files
  - **Continuous Allocation:** making requests to access adjacent to each other
  - **Link/index Allocation:** can include widely split blocks on disk
- Disk access control algorithms can be written as **separate modules** of the OS allowing them to be replaced by other algorithms as needed.
- Both **SSTF** and **LOOK** can be reasonable choices for the **default algorithm**