

# Writing OS

## • Writing an OS in Rust

<https://os.phil-opp.com/>

os.phil-opp.com

### Writing an OS in Rust Philipp Oppermann's blog

This blog series creates a small operating system in the [Rust programming language](#). Each post is a small tutorial and includes all needed code, so you can follow along if you like. The source code is also available in the corresponding [Github repository](#).

Latest post: [Async/Await](#)

BARE BONES

#### A Freestanding Rust Binary

The first step in creating our own operating system kernel is to create a Rust executable that does not link the standard library. This makes it possible to run Rust code on the [bare metal](#) without an underlying operating system. [read more »](#)

#### A Minimal Rust Kernel

In this post we create a minimal 64-bit Rust kernel for the x86 architecture. We build upon the [freestanding Rust binary](#) from the previous post to create a bootable disk image, that prints something to the screen. [read more »](#)

#### VGA Text Mode

The [VGA text mode](#) is a simple way to print text to the screen. In this post, we create an interface that makes its usage safe and simple, by encapsulating all unsafety in a separate module. We also implement support for Rust's [formatting macros](#). [read more »](#)

#### Testing

This post explores unit and integration testing in `no_std` executables. We will use Rust's support for custom test frameworks to execute test functions inside our kernel. To report the results out of QEMU, we will use different features of QEMU and the `bootimage` tool.

#### Other Languages

- [Persian](#)
- [Japanese](#)
- [Russian](#)
- [Chinese \(simplified\)](#)
- [Chinese \(traditional\)](#)

#### Recent Updates

No notable updates recently.

#### Repository

 [phil-opp/blog\\_os](#)  
Writing an OS in Rust

★ 7.7k    607    Sponsor

# Writing OS

## • Write operating system from scratch

[https://learningos.github.io/rcore\\_step\\_by\\_step\\_webdoc/](https://learningos.github.io/rcore_step_by_step_webdoc/)



The screenshot shows a web browser displaying the Learning OS documentation. The address bar shows the URL `learningos.github.io/rcore_step_by_step_webdoc/`. On the left, there is a search bar and a table of contents with items like "0. 从零开始写 OS", "1. 独立式可执行程序", "2. 最小化内核", "3. 格式化输出", "4. 中断异常", "5. 内存分配", "6. 页表简介", "7. 内核线程", "8. 线程调度", "9. 创建页表", "10. 用户进程", and "11. 命令行". The main content area is titled "从零开始写 OS" and includes a "前言" (Preface) section, a "前置知识" (Prerequisites) section, and a "如何使用" (How to Use) section. The "前言" section mentions that the series records the process of writing a small operating system using Rust. The "前置知识" section lists Rust syntax and RISC-V assembly. The "如何使用" section suggests using Docker for a better experience.

← → ↻ `learningos.github.io/rcore_step_by_step_webdoc/`

Type to search

0. 从零开始写 OS

1. 独立式可执行程序

2. 最小化内核

3. 格式化输出

> 4. 中断异常

5. 内存分配

6. 页表简介

7. 内核线程

8. 线程调度

9. 创建页表

10. 用户进程

> 11. 命令行

Published with GitBook

### 从零开始写 OS

#### 前言

本系列文章记录了使用 Rust 编程语言编写一个小型操作系统的详细过程。每篇文章包含所需所有所需代码和相关知识点讲解。

- 源代码
- 源文档
- web 文档

#### 前置知识

- 简单的 Rust 语法（这个语言的基础使用方式和 C 类似，文章会介绍其新奇的语法）
- RISC-V 汇编（用的不多）

#### 如何使用

在阅读文章以及复现的过程中了解 OS。

为了方便起见，建议使用 `docker`，可以省去配置环境的功夫。

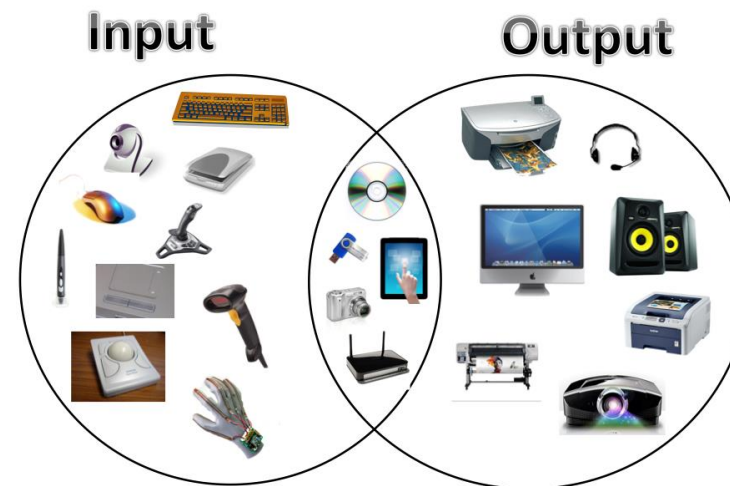
# Course Review

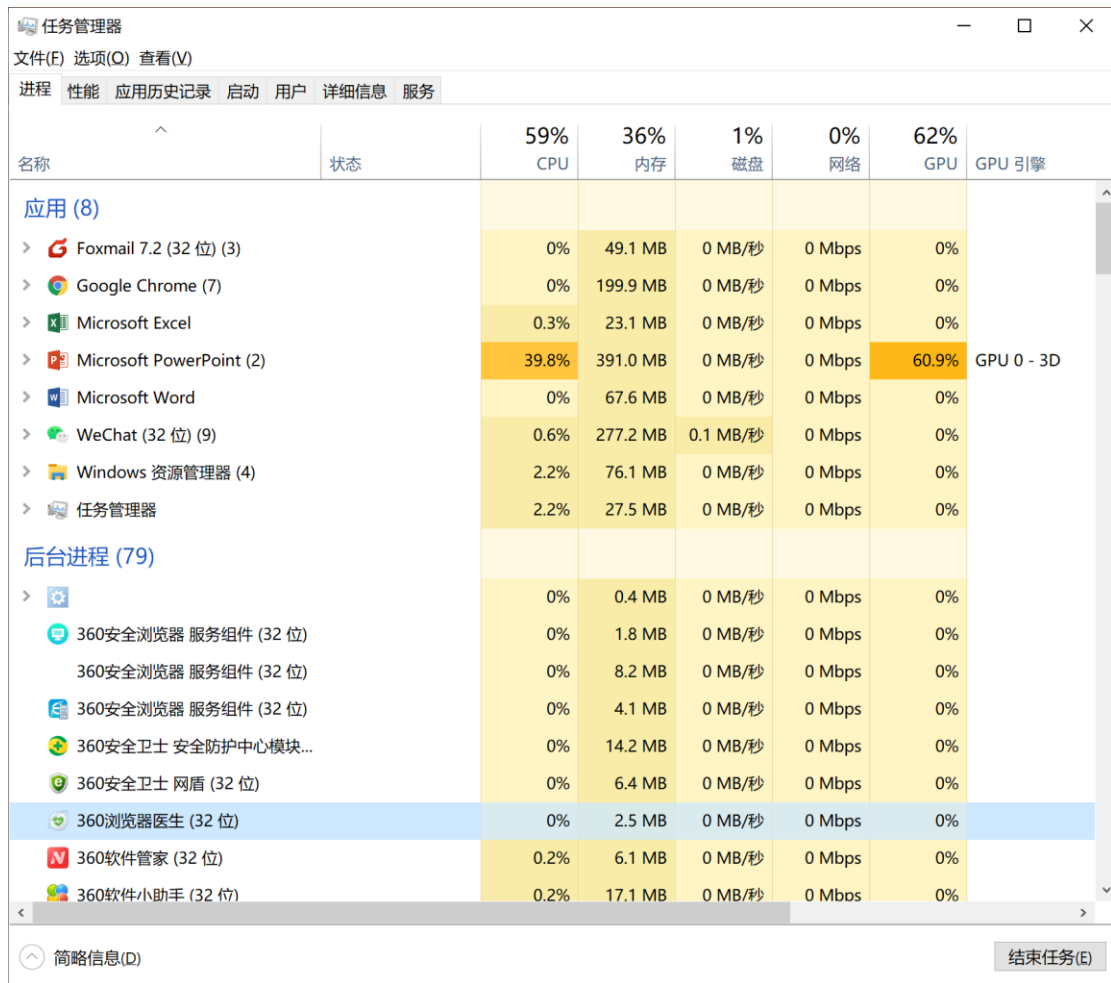
---

- What is OS ?
- OS components
- OS services
- OS design goals
- OS characteristics
- Early OSs

# Manage Execution of Applications

- **Resources** are made available to multiple applications
- **Processor is switched** among multiple applications
- The processor and I/O devices can be used efficiently







Windows 任务管理器

文件(F) 选项(O) 查看(V) 关机(U) 帮助(H)

应用程序 进程 性能 联网 用户

映像名称	用户名	CPU	内存使用
alg.exe	LOCAL SERVICE	00	3,928 K
ati2evxx.exe	SYSTEM	00	4,196 K
ati2evxx.exe	SYSTEM	00	4,692 K
AztecTribe.exe	Formal	00	82,680 K
calc.exe	Formal	00	4,196 K
CCC.exe	Formal	00	7,196 K
Com4QLBEx.exe	SYSTEM	00	3,196 K
csrss.exe	SYSTEM	00	10,196 K
ctfmon.exe	Formal	00	3,196 K
explorer.exe	Formal	00	16,196 K
hpqwmie.exe	SYSTEM	00	4,196 K
lsass.exe	SYSTEM	00	1,196 K
MDM.EXE	SYSTEM	00	3,248 K
MOM.exe	Formal	00	6,428 K
POWERPNT.EXE	Formal	00	52,824 K
QLBCTRL.exe	Formal	00	5,928 K
RavMonD.exe	SYSTEM	00	8,192 K
rftmain.exe	Formal	00	12,796 K

☐ 显示所有用户的进程(S)

进程数: 40 CPU 使用: 0% 提交更改: 744M / 3935M

结束进程(E)  
结束进程树(T)  
调试(D)  
设置优先级(P) >  
关系设置(A)...

实时(R)  
高(H)  
高于标准(A)  
● 标准(N)  
低于标准(B)  
低(L)

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	PID	状态	用户名	CPU	内存(专用...)	描述
acrotray.exe	13020	正在运行	dizhang	00	1,140 K	AcroTray
aesm_servi...	...	正在运行	SYSTEM	00	1,716 K	Intel® SGX Application Encla...
AGMServic...	...	正在运行	SYSTEM	00	2,232 K	Adobe Genuine Software Ser...
AGSServic...	...	正在运行	SYSTEM	00	2,864 K	Adobe Genuine Software Inte...
ApplePhot...	...	正在运行	dizhang	00	7,948 K	iCloud Photo Stream
Application...	...	正在运行	dizhang	00	7,784 K	Application Frame Host
APSDaemo...	...	正在运行	dizhang	00	4,668 K	Apple Push
armsvc.exe	...	正在运行	dizhang	00	1,032 K	Adobe Acrobat Update Servi...
audiodg.ex...	...	正在运行	dizhang	00	7,340 K	Windows 音频设备图形隔离
chrome.exe	...	正在运行	dizhang	00	2,652 K	Google Chrome
chrome.exe	...	正在运行	dizhang	00	7,200 K	Google Chrome
chrome.exe	...	正在运行	dizhang	00	1,428 K	Google Chrome
chrome.exe	...	正在运行	dizhang	00	99,160 K	Google Chrome
chrome.exe	...	正在运行	dizhang	00	23,080 K	Google Chrome
chrome.exe	4644	正在运行	dizhang	00	122,108 K	Google Chrome
chrome.exe	2520	正在运行	dizhang	00	16,860 K	Google Chrome
chrome.exe	12928	正在运行	dizhang	00	22,296 K	Google Chrome
chrome.exe	11144	正在运行	dizhang	00	7,948 K	Google Chrome
chrome.exe	17052	正在运行	dizhang	00	26,104 K	Google Chrome
chrome.exe	8500	正在运行	dizhang	00	50,924 K	Google Chrome

结束任务(E)  
结束进程树(T)  
设置优先级(P) >  
设置相关性(F)  
分析等待链(A)  
UAC 虚拟化(V)  
创建转储文件(C)  
打开文件所在的位置(O)  
在线搜索(N)  
属性(R)  
转到服务(S)

实时(R)  
高(H)  
高于正常(A)  
● 正常(N)  
低于正常(B)  
低(L)

简略信息(D) 结束任务(E)



# Processes

---

# Outline

---

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Communication in Client-Server Systems



?



What is a Process?  
How to describe a  
Process?



# Process Concept

---

# What is a process?

- One program which has an independent function works on certain data set dynamically and allocates resources dynamically
- **Process** is a program in execution
- **An instance** of a **program** running on a computer



# Difference Between Process & Program

- A program is a **passive** entity but a process is an **active** entity
- Process is **dynamic**, and the program is **static**
- Process is **temporary**, the program is **permanent**
- The elements of process and program is different
  - Process: **Process Control Block**
  - Program: **Code** and **Data**
- Two **processes** may be associated with the same **program**, they are considered as separate execution sequences

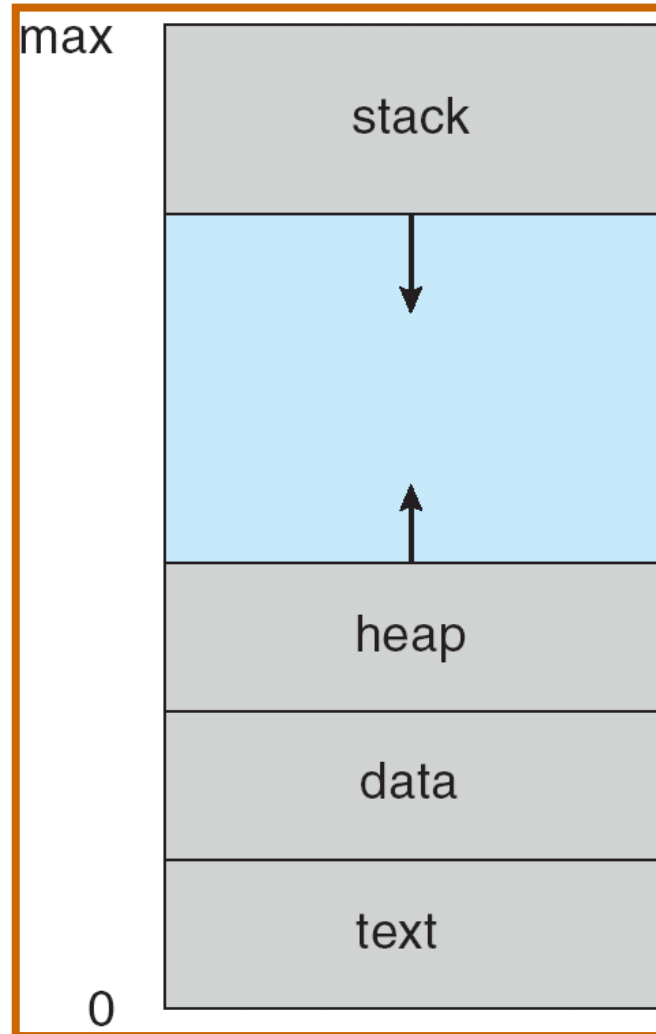
# Processes

---

- A system therefore consists of a collection of processes:
  - Operating system processes execute system code, *and*
  - User processes execute user code.

***System Process  
& User Process***

# Process Structure in Memory



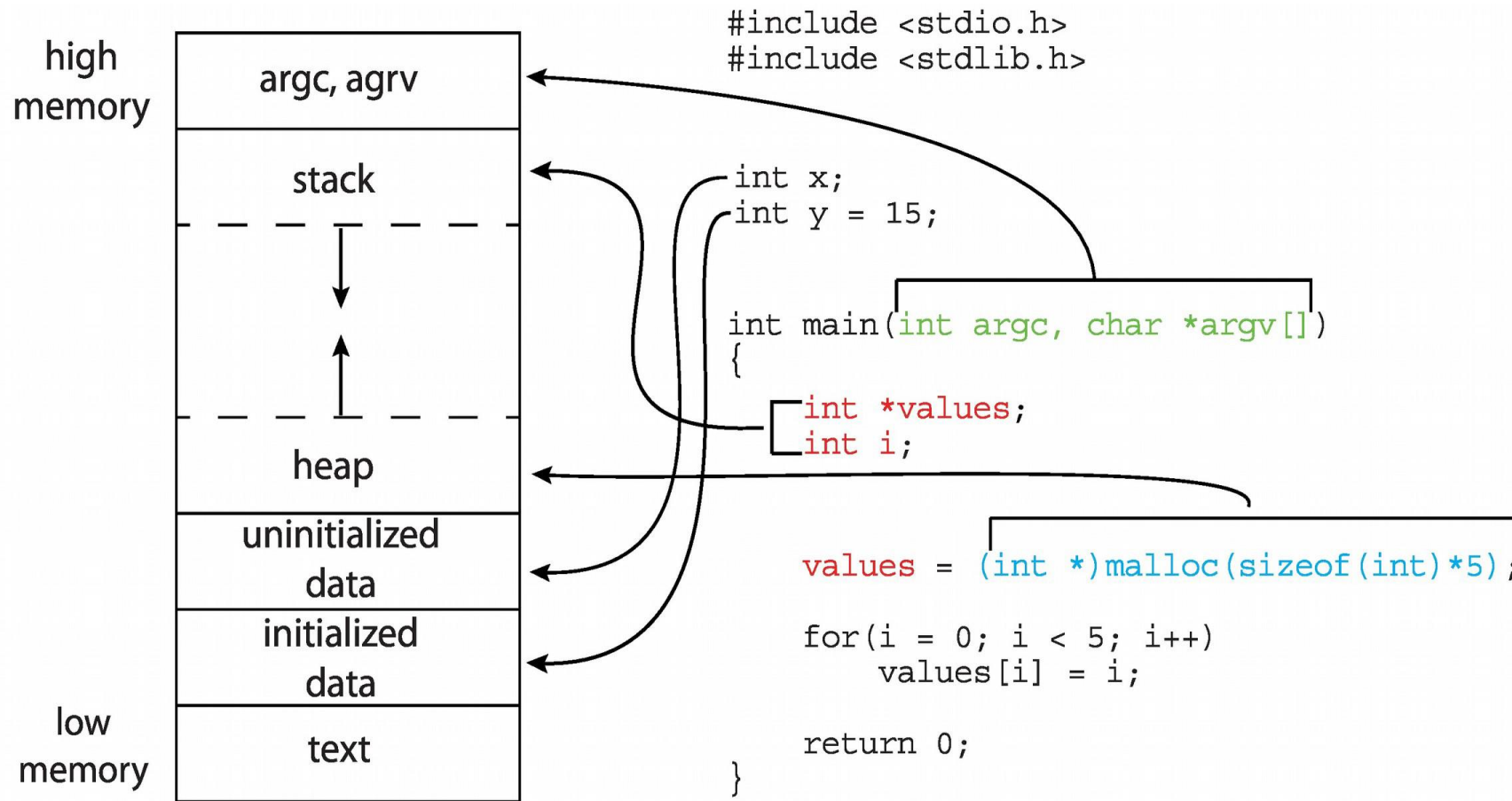
**Temporary data**  
(function parameters, return addresses, local variables)

**Dynamically allocated memory at runtime**

**Global variables**

**Value of program counter;  
Contents of processor's registers**

# Memory Layout of a C Program

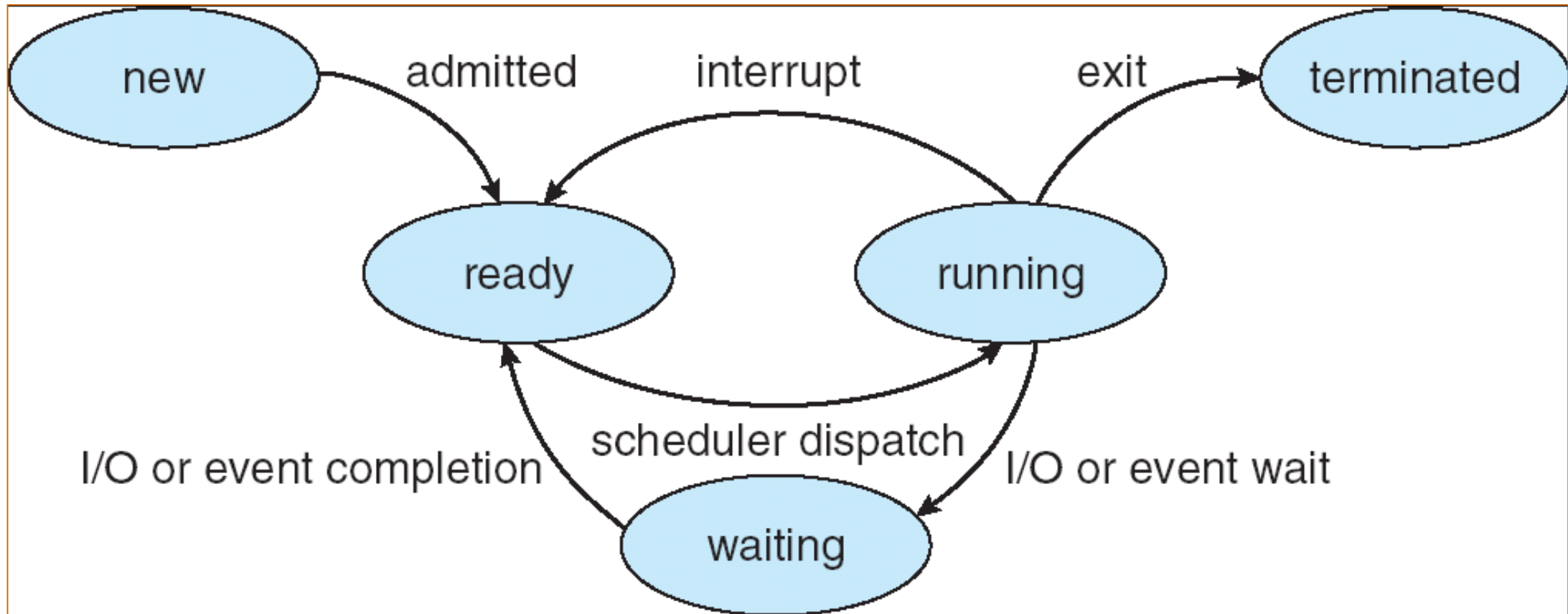


# Process State

- Process state
  - **new**: the process is being created
  - **running**: instructions are being executed
  - **waiting**: the process is waiting for some event to occur (such as an I/O completion or reception of a signal)
  - **ready**: the process is waiting to be assigned to a processor
  - **terminated**: the process has finished execution
- As a process executes, it changes state
- **Only one** process is **running** on one processor at any instant. However, **many** processes may **be ready** or **waiting**



# Diagram of Process State

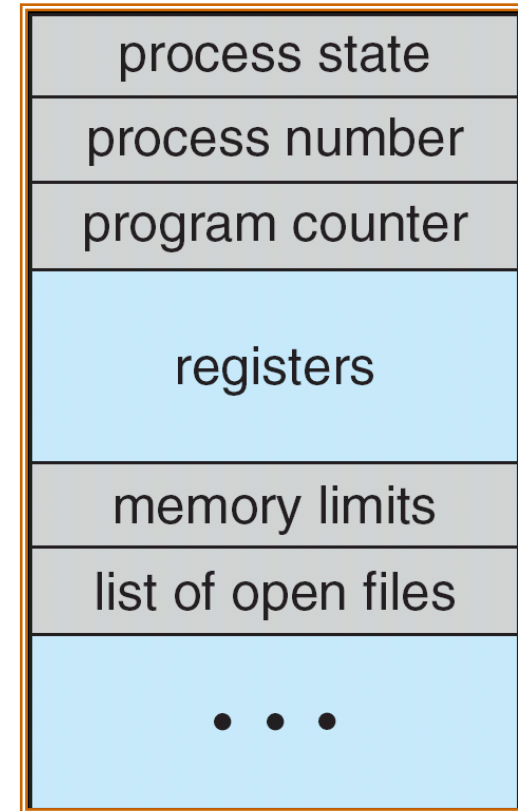


# Process Control Block (PCB)

- Information associated with **each** process
  - Identifier
  - Process state
  - CPU scheduling information (e.g., priority)
  - Program counter: **next** instruction to be executed
  - CPU registers
  - Memory-management information (e.g., base/limit register, page or segment tables)
  - Accounting information
  - I/O status information

# Process Control Block (PCB)

- The current state of process is held in a **process control block (PCB)**:
  - This is a “snapshot” of the execution and protection environment
  - Only one PCB is active at a time
- Created and managed by the **operating system**



**Process Control Block**

# Implementation of Processes

---

- OS creates process, allocates memory, I/O devices, files, and so on to user program.
- Create the PCB
  - the number is limited



# Process Scheduling

---

```

Untitled-1 *
In[4]= 2000!
Out[4]= 331 627 509 245 063 324 117 539 338 057 \
        632 403 828 111 720 810 578 039 457 193 \
        543 706 038 077 905 600 822 400 273 230 \
        859 732 592 255 402 352 941 225 8
        258 084 817 415 293 796 131 386 6
        343 688 905 634 058 556 163 940 6
        252 571 870 647 856 393 544 045 4
        957 467 037 674 108 722 970 434 6
        343 752 431 580 877 533 645 127 4
        436 859 247 408 032 408 946 561 5
        250 652 797 655 757 179 671 536 7
        359 056 112 815 871 601 717 232 6
        110 004 214 012 420 433 842 573 7
        175 883 547 796 899 921 283 528 9
        853 405 579 854 903 657 366 350 1
    
```


### 磁盘碎片整理程序

文件(F) 操作(A) 查看(V) 帮助(H)


← → [E] [F] [B]

卷	会话状态	文件系统	容量	可用空间	% 可用空间
(C:)		NTFS	20.00 GB	9.39 GB	46 %
SOFTWARE (D:)	经过分析的	NTFS	98.04 GB	66.39 GB	67 %
MEDIA (E:)	停止	NTFS	98.04 GB	55.17 GB	56 %
BACKUP (F:)	正在进行碎片整...	NTFS	82.01 GB	533 MB	0 %
紫光凯远 (H:)	停止	FAT	1.95 GB	382 MB	19 %

进行碎片整理前预计磁盘使用量:



进行碎片整理后预计磁盘使用量:



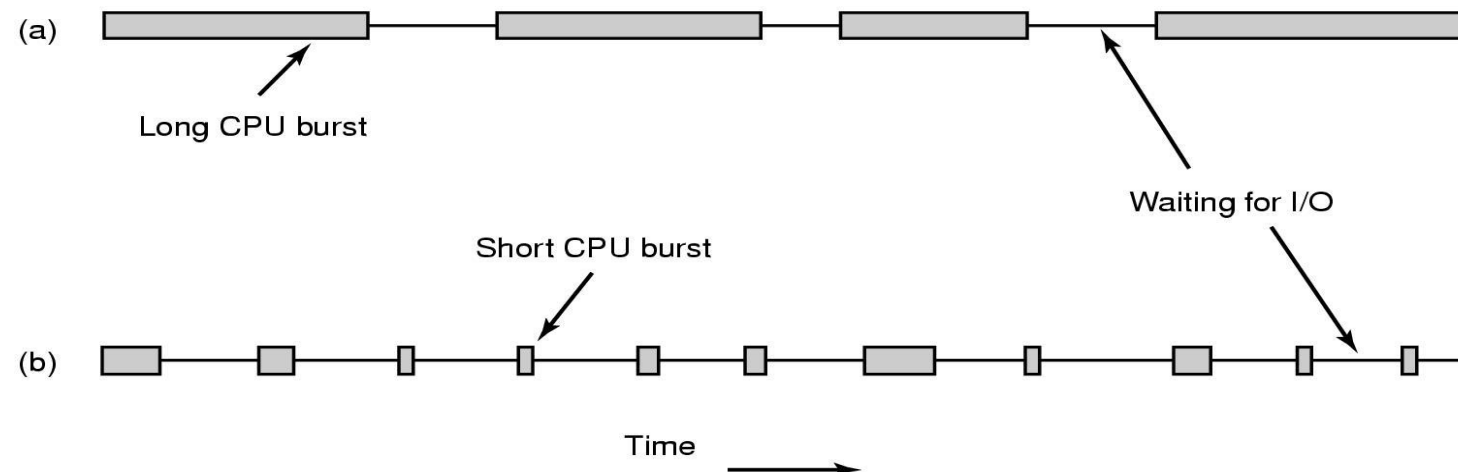
分析 碎片整理 暂停 停止 查看报告

☒ 零碎的文件
 ☒ 连续的文件
 ☐ 无法移动的文件
 ☐ 可用空间

BACKUP (F:) 正在进行碎片整理... 3% 正在移动文件unins000.dat

# Introduction to Scheduling

- Bursts of CPU usage alternate with periods of I/O wait
- Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; many long CPU bursts



# Introduction to Scheduling

- Why Scheduling?
- For a uni-processor system, there is **only one** running process. The rest should wait until CPU free and is rescheduled





# Process Scheduling Queues

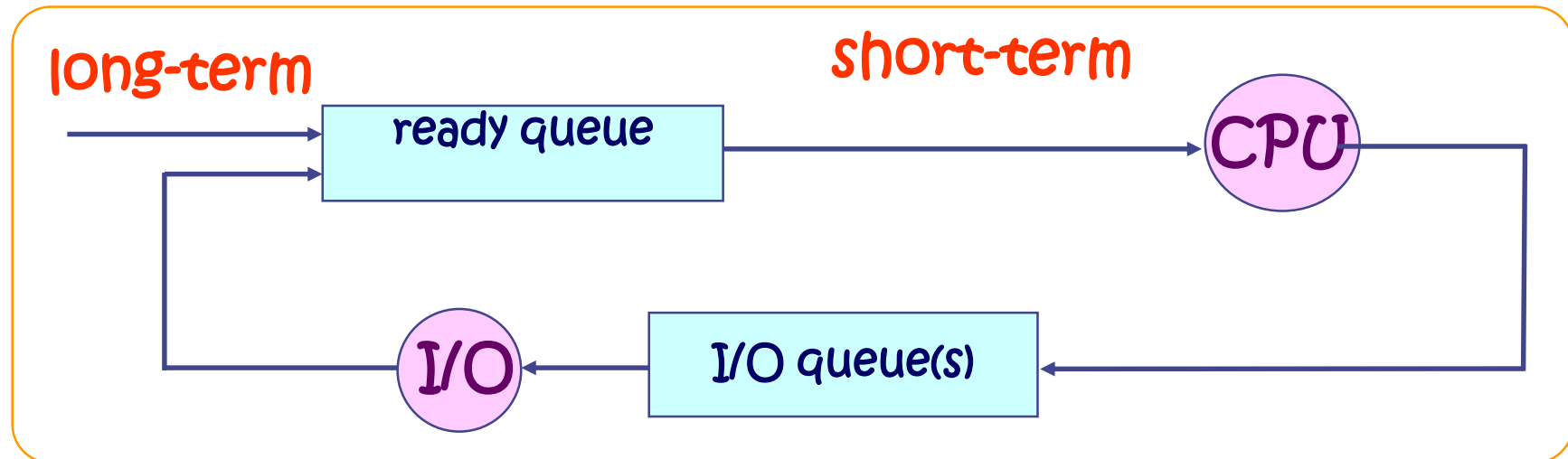
- Job queue
  - set of all processes in the system
- Ready queue
  - set of all processes residing in main memory, ready and waiting to execute
- Device queues
  - set of processes waiting for an I/O device
- Processes **migrate** among the various queues

# Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Execute **less frequently** (e.g., once several minutes)
  - Control the degree of multiprogramming
  - Select a good process mix of I/O-bound processes and CPU-bound processes

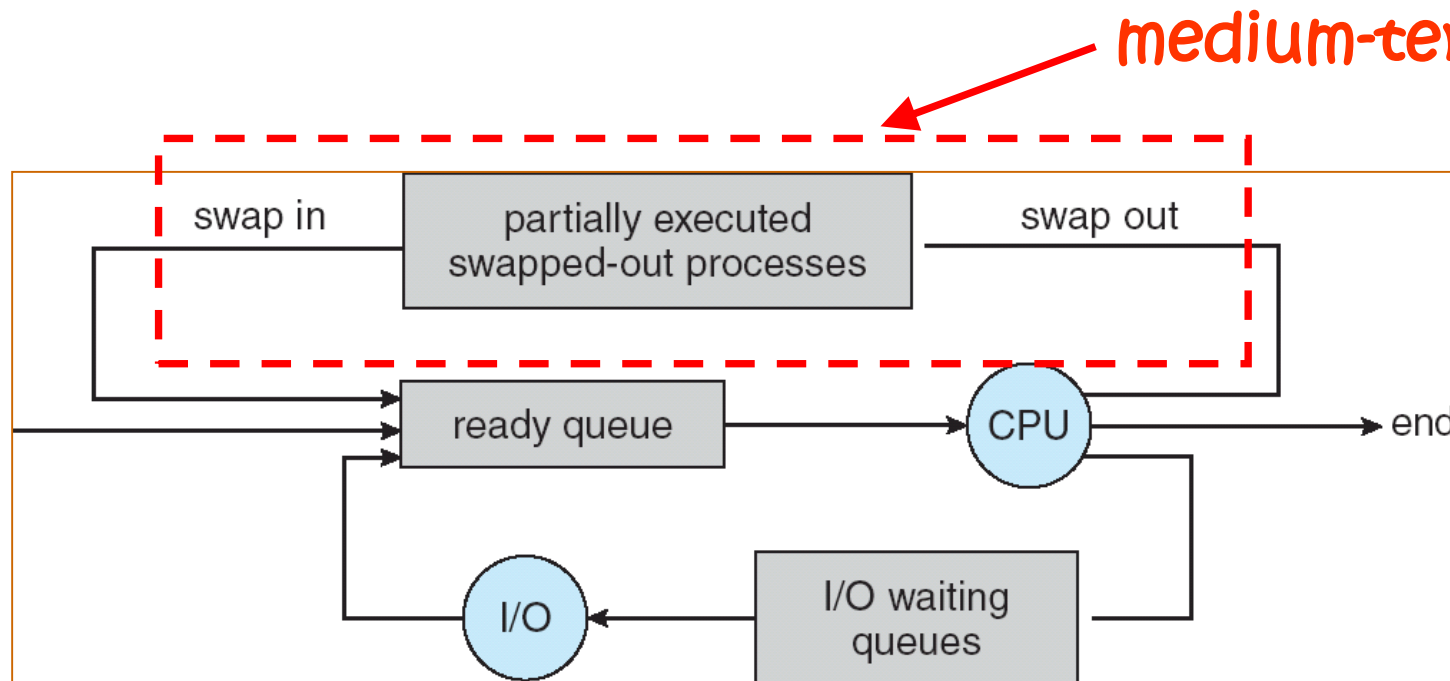
# Schedulers

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Execute **quite frequently** (e.g., once per 100ms)
  - Must be very fast
    - e.g., if it takes 10 ms, then  $10/(100+10)$  percent of CPU is wasted.



# Schedulers

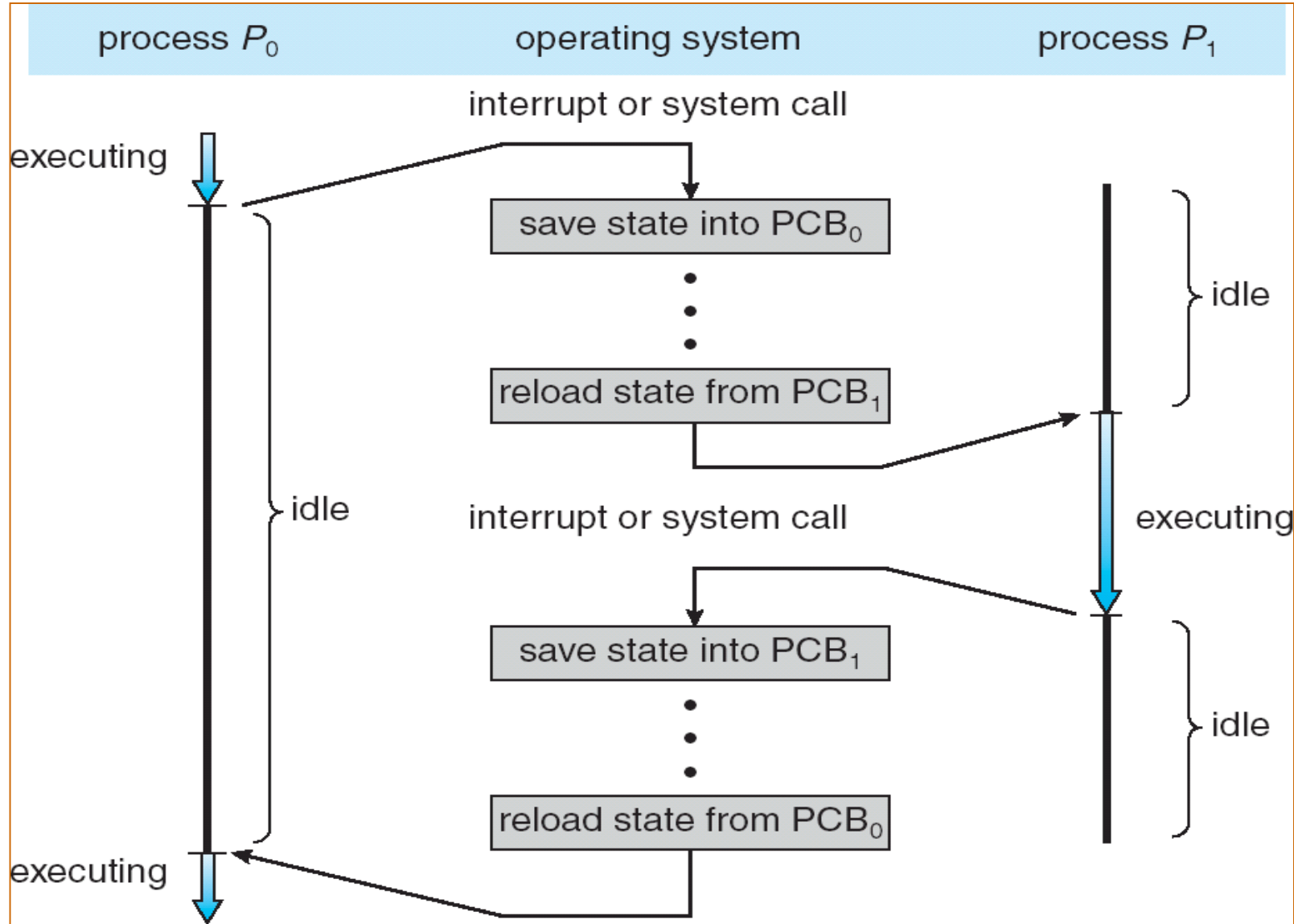
- **Medium-term Scheduler** (Swapping)
  - **swap out**: removing processes from memory to reduce the degree of multiprogramming.
  - **swap in**: reintroducing swap-out processes into memory



# Differences between These Schedulers

- **Long-term scheduling** - job scheduling, select job from external storage to memory and create a process
  - invoked **very infrequently** (seconds, minutes)  $\Rightarrow$  (may be **slow**)
- **Short-term scheduling** - process scheduling, select the ready process to run on the processor
  - invoked **very frequently** (milliseconds)  $\Rightarrow$  (must be **fast**)
- **Medium-term scheduling** - solves the problem of insufficient memory, using secondary storage to alleviate (controls the degree of multiprogramming)

# CPU Switch from Process to Process



# Context Switch

- When CPU switches to another process, the system must **save the state of the old process** and load **the saved state** for the new process
- This is also called a "**context switch**"
- Context-switch time is **overhead**; the system does no useful work while switching



# Operations on Processes

---



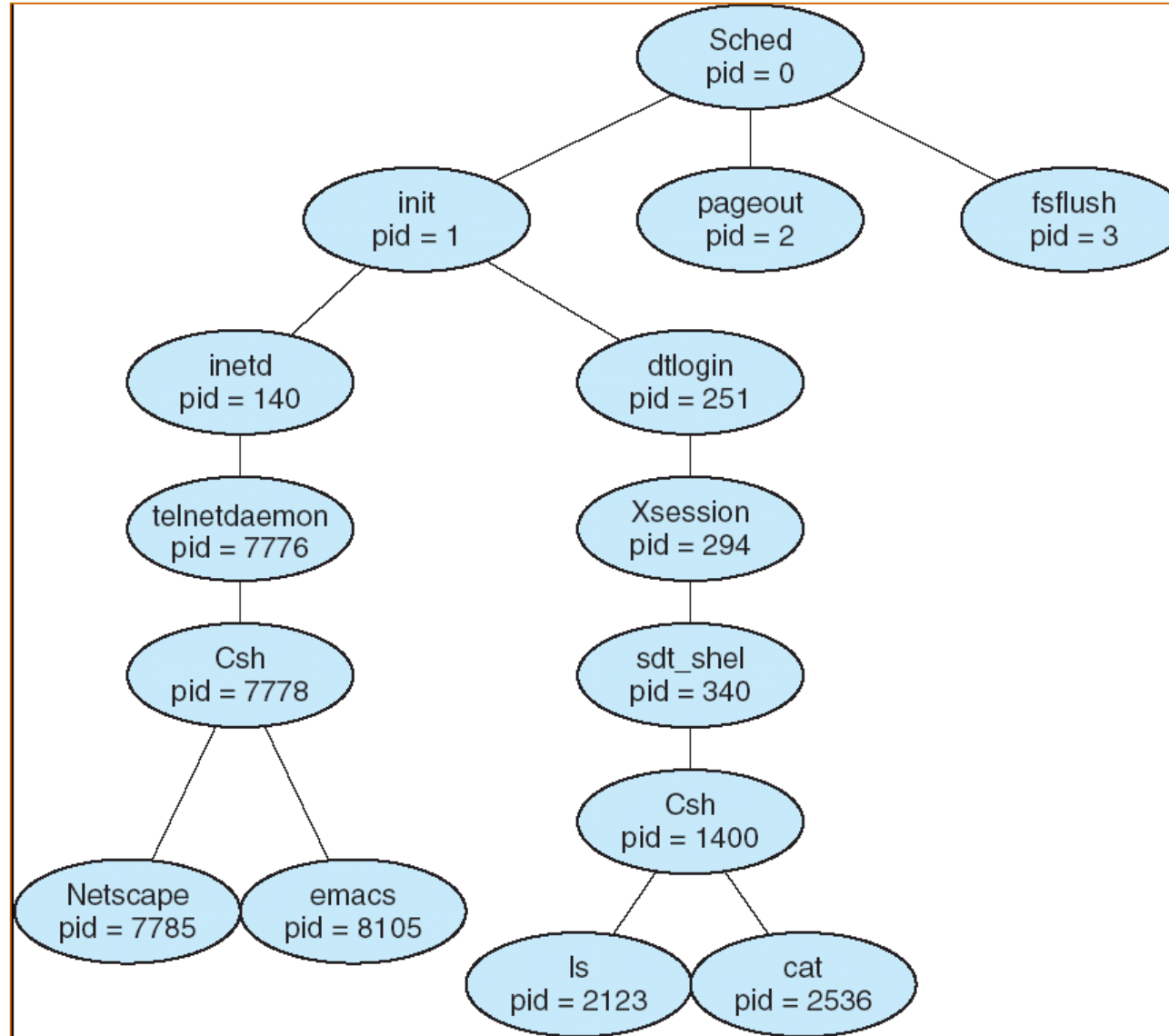
# Operations on Processes

- Five major activities of an operating system in regard to process management
  - The **creation** and **deletion** of both user and system processes
  - The **suspension** and **resumption** of processes
  - The provision of mechanisms for **process synchronization**
  - The provision of mechanisms for **process communication**
  - The provision of mechanisms for **deadlock handling**

# Process

- A process may **spawn** many processes as it runs
- **Parent process** create **children processes**, which, in turn create other processes, forming a tree of processes
- Forms a hierarchy
  - UNIX calls this a “ process group”
- Windows has no concept of process hierarchy
  - all processes are created equally (by using handle)

# A tree of processes on a typical Solaris



# Process Creation

- **Parent process** create **children processes**, thus forming a tree of processes
- **Resource sharing**
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- **Execution**
  - Parent and children execute concurrently
  - Parent waits until children terminate

# Process Termination

---

- Conditions which terminate processes
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
  - Killed by another process (involuntary)

# CreateProcessA

## BOOL CreateProcessA()

### CreateProcessA function (processthreadsapi.h)

• 13 minutes to read

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the [CreateProcessAsUser](#) or [CreateProcessWithLogonW](#) function.

### Syntax

```
BOOL CreateProcessA(  
    LPCSTR          lpApplicationName,  
    LPSTR           lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL            bInheritHandles,  
    DWORD           dwCreationFlags,  
    LPVOID          lpEnvironment,  
    LPCSTR          lpCurrentDirectory,  
    LPSTARTUPINFOA  lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```

# ExitProcess

## void ExitProcess(UINT uExitCode);

### ExitProcess function (processthreadsapi.h)

• 2 minutes to read

Ends the calling process and all its threads.

### Syntax

```
void ExitProcess(  
    UINT uExitCode  
);
```

### Parameters

**uExitCode**

The exit code for the process and all threads.

### Return value

None

### Remarks

Use the [GetExitCodeProcess](#) function to retrieve the process's exit value. Use the [GetExitCodeThread](#) function to retrieve a thread's exit value.

<https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess>

# TerminateProcess

- BOOL TerminateProcess(HANDLE hProcess,UINT uExitCode);
  - GetCurrentProcess()

## TerminateProcess function (processthreadsapi.h)

• 2 minutes to read

Terminates the specified process and all of its threads.

### Syntax

```
BOOL TerminateProcess(  
    HANDLE hProcess,  
    UINT uExitCode  
);
```

### Parameters

**hProcess**

A handle to the process to be terminated.

The handle must have the **PROCESS\_TERMINATE** access right. For more information, see [Process Security and Access Rights](#).

**uExitCode**

The exit code to be used by the process and threads terminated as a result of this call. Use the [GetExitCodeProcess](#) function to retrieve a process's exit value. Use the [GetExitCodeThread](#) function to retrieve a thread's exit value.

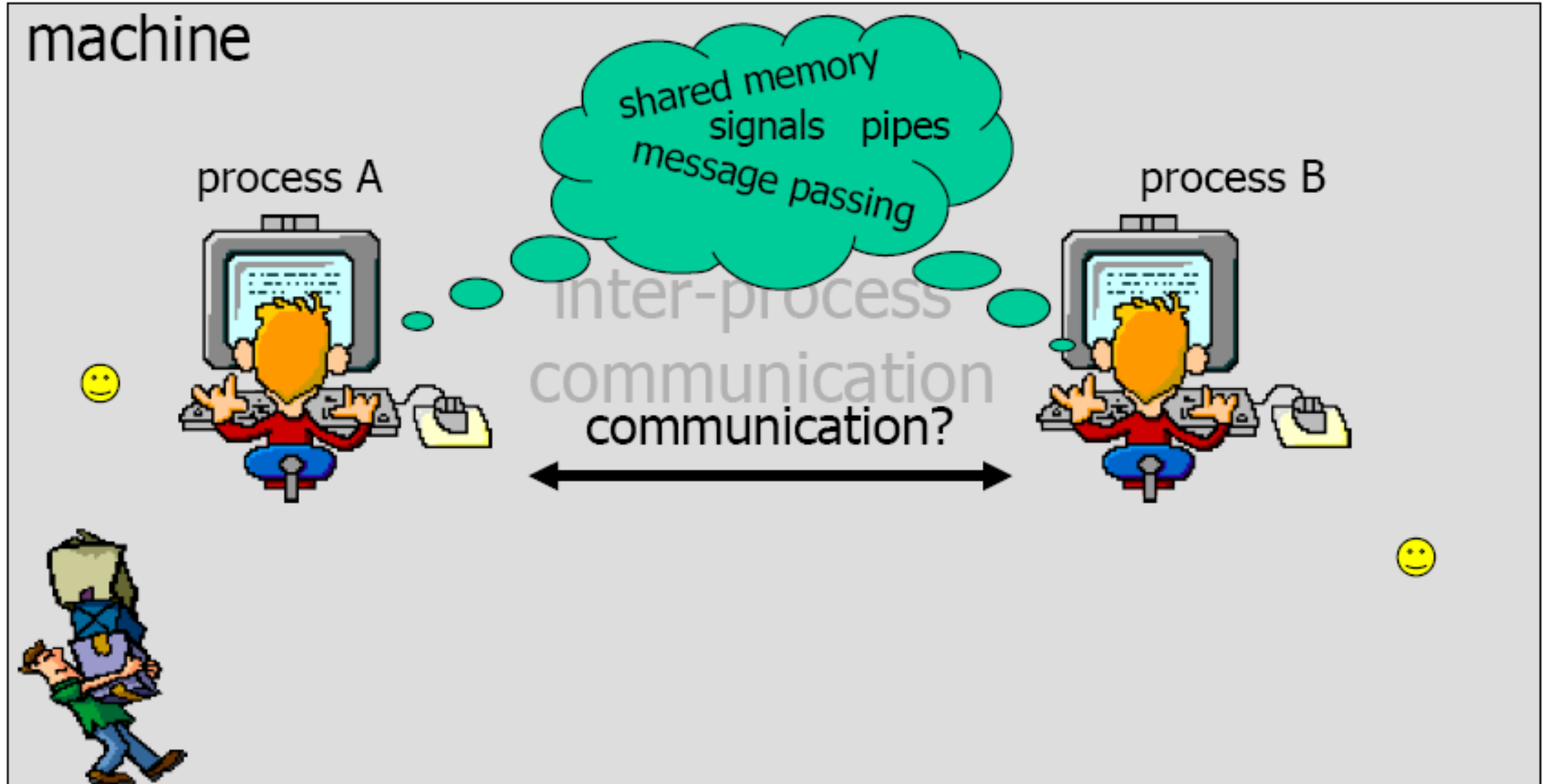




# Interprocess Communication

---

# Cooperating Processes



# Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience, e.g., A user can performs several tasks at the same time (editing, printing, compiling)

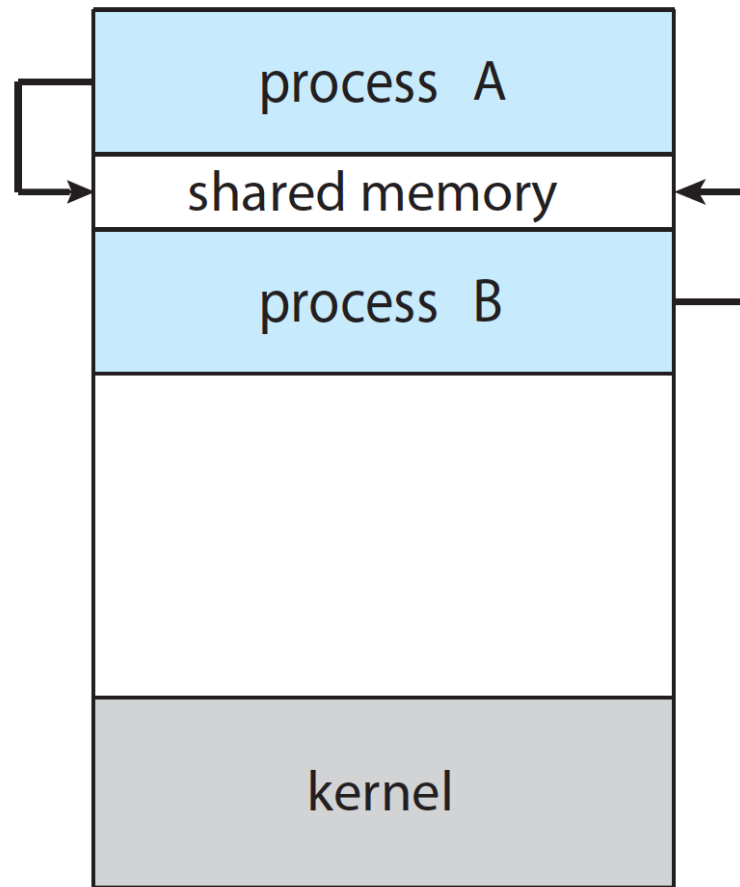
# Interprocess Communication (IPC)

- Mechanism for processes to **communicate** and to **synchronize** their actions

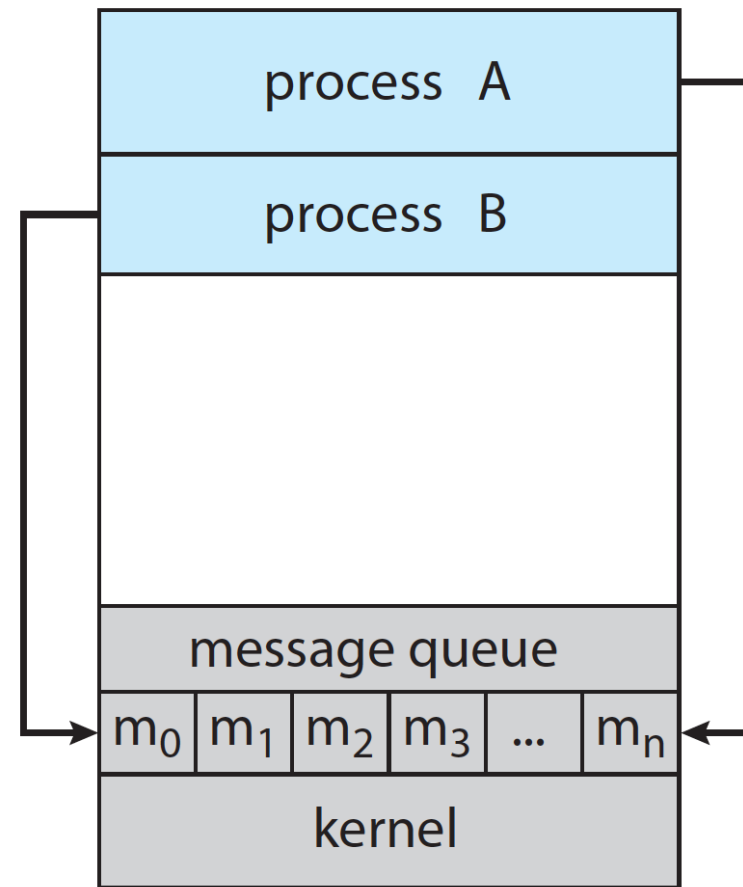
## Inter-Process Communication (IPC)



# IPC Communication Models



(a)  
Shared memory

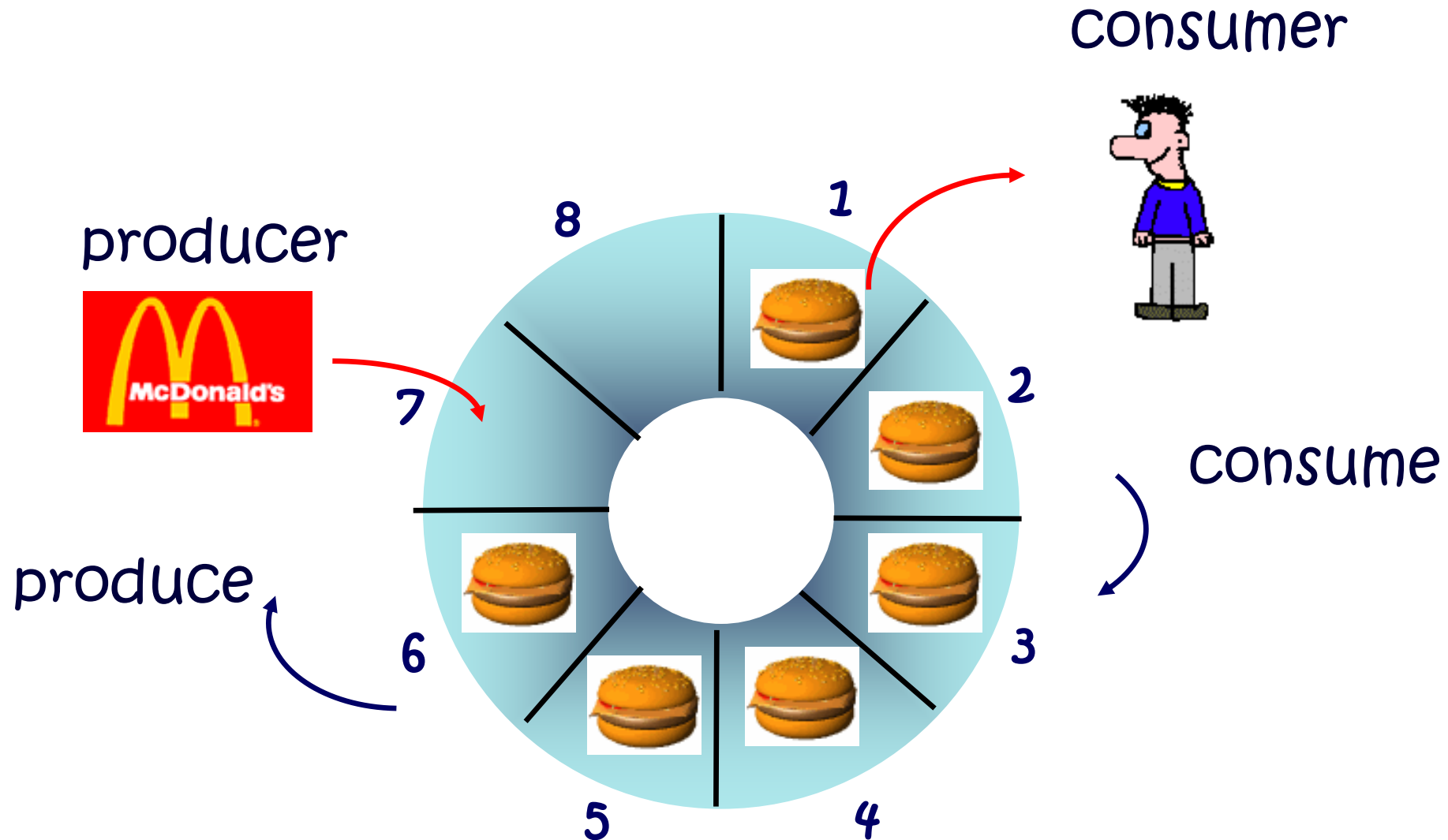


(b)  
Message passing

# Producer-Consumer Problem

- Paradigm for cooperating processes, **producer** process produces information that is consumed by a **consumer** process
  - **unbounded-buffer** places no practical limit on the size of the buffer
    - **producer**: no wait
    - **consumer**: wait when buffer is empty
  - **bounded-buffer** assumes that there is a fixed buffer size
    - **producer**: wait when buffer is full
    - **consumer**: wait when buffer is empty

# Example of Producer-Consumer Problem



# Bounded-Buffer: Shared-Memory Solution

- Shared data: Buffer as a circular array

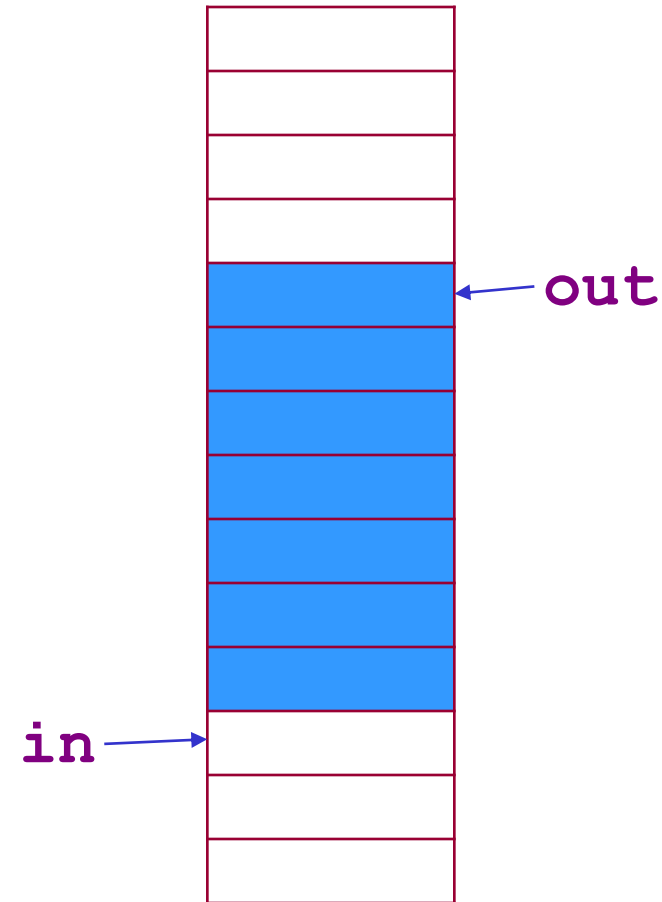
```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];

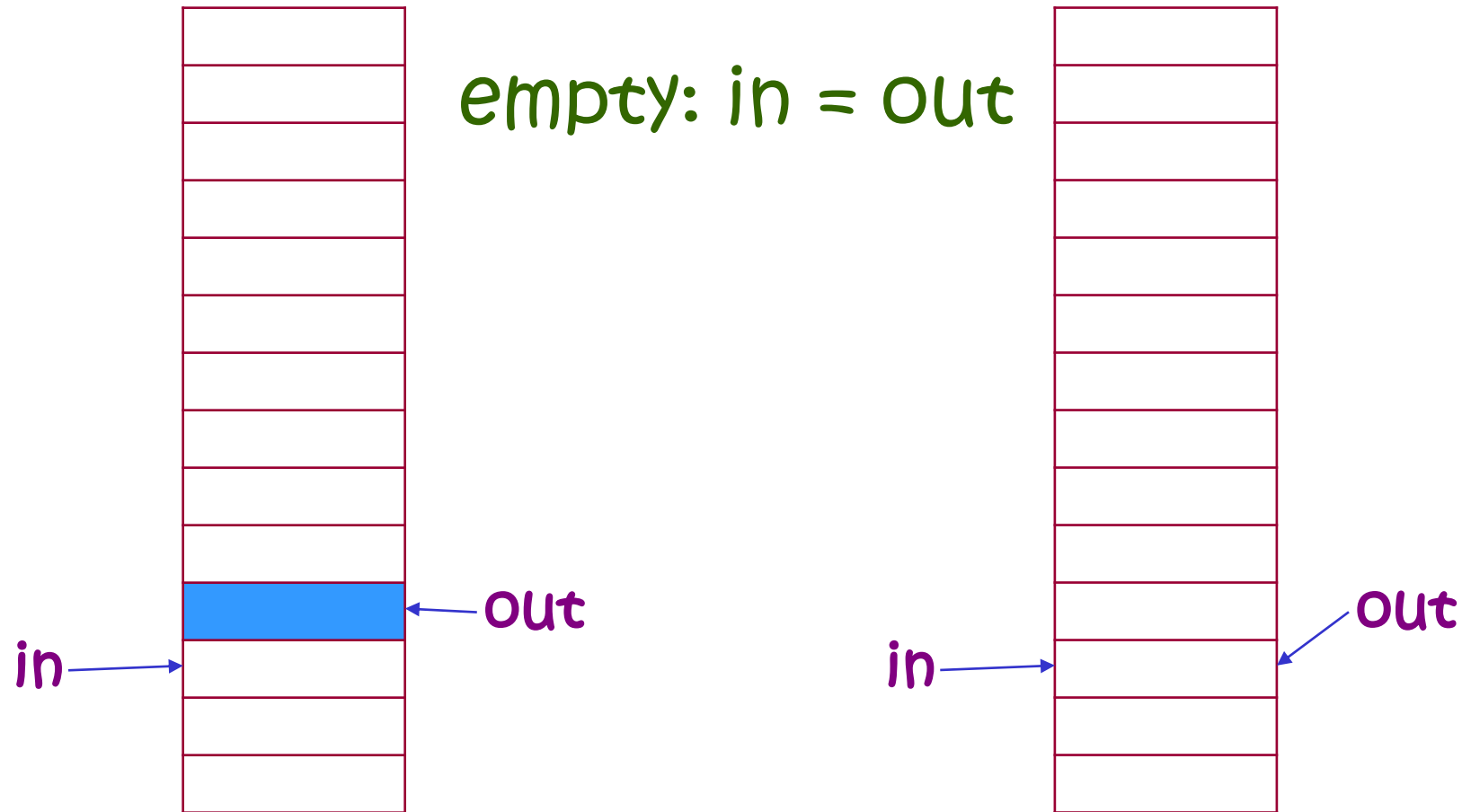
int in = 0;
int out = 0;
```

- Shared data: Buffer as a circular array
  - next free: `in`
  - first available: `out`
  - empty: `in = out`
  - full: `(in+1) % B = out`

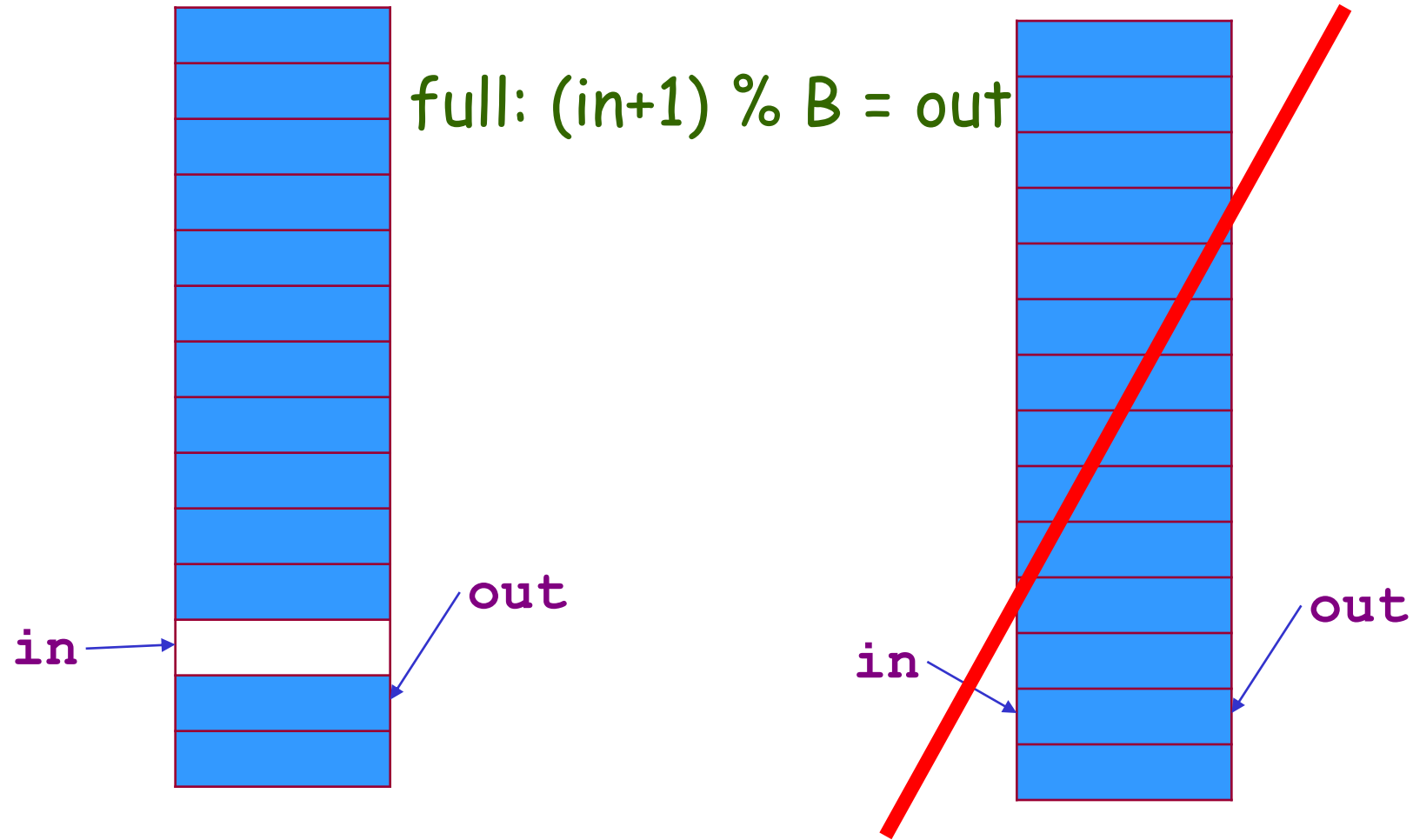




# Bounded-Buffer: Shared-Memory Solution



# Bounded-Buffer: Shared-Memory Solution



# Bounded-Buffer: Producer Process–Insert()

```
while (true) {  
    /* Produce an item */  
    while (((in+1) % BUFFER SIZE) == out)  
        ; /*do nothing--no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```

# Bounded Buffer: Consumer Process – Remove()

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    return item;  
}
```

# Message system

- **Message system** – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(message) – message size fixed or variable
  - **receive**(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive

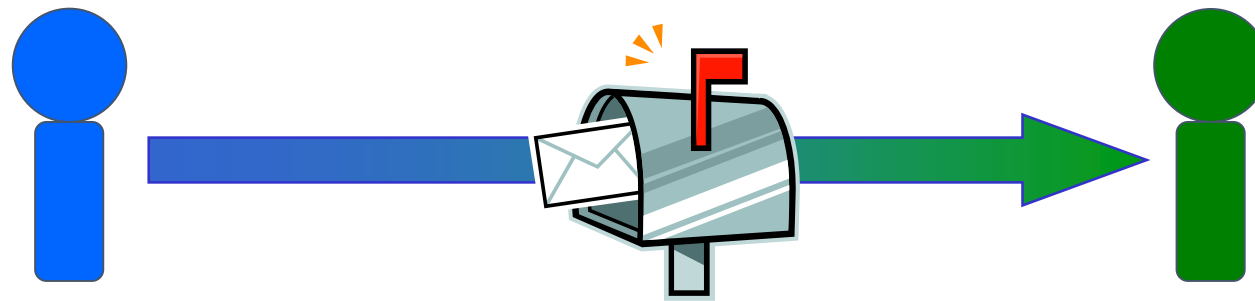


# Direct Communication

- Processes must name each other explicitly:
  - **send** (P, message) – send a message to process P
  - **receive** (Q, message) – receive a message from process Q
- **limited modularity**: if the name of a process is changed, all old names should be found. It is not easy for separate compilation.

# Indirect Communication

- Messages are sent and received from **mailboxes** (also referred to as **ports**)
  - Each mailbox has a unique **id**
  - Processes can communicate only if they share a mailbox



# Indirect Communication

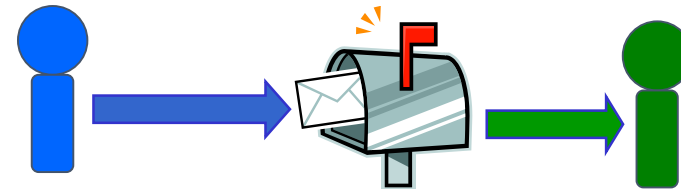
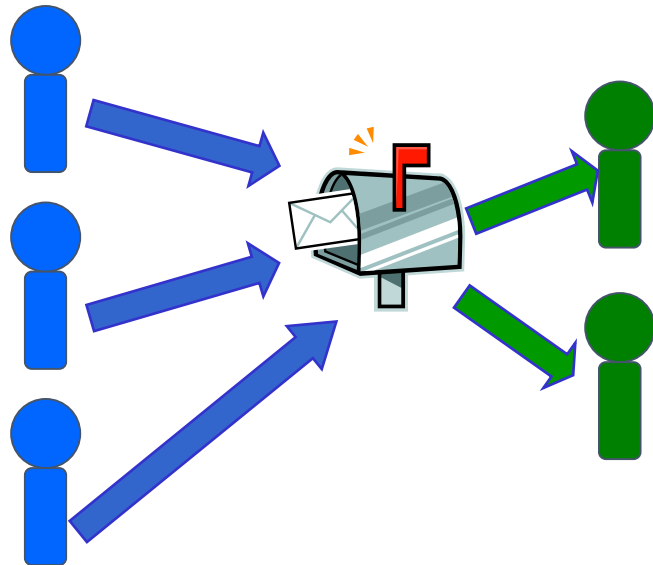
- Operations
  - create a new **mailbox**
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send**(A, message) – send a message to mailbox A
  - receive**(A, message) – receive a message from mailbox A





# Indirect Communication

- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional



# Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - Blocking send: the sender is blocked until the message is received
  - Blocking receive: the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - Non-blocking send: the sender sends the message and continue
  - Non-blocking receive: the receiver receives a valid message or null

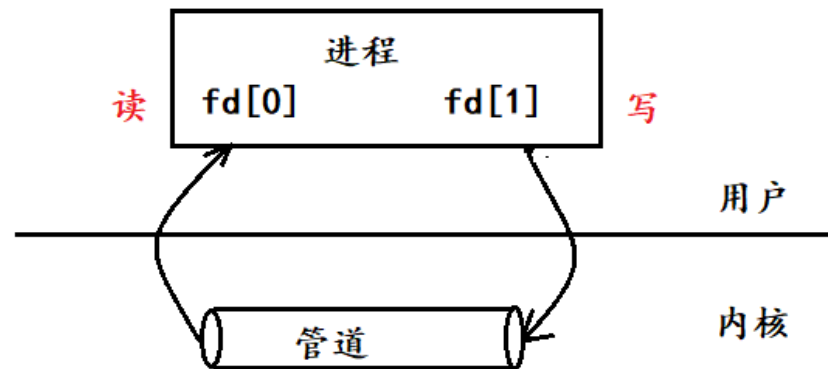
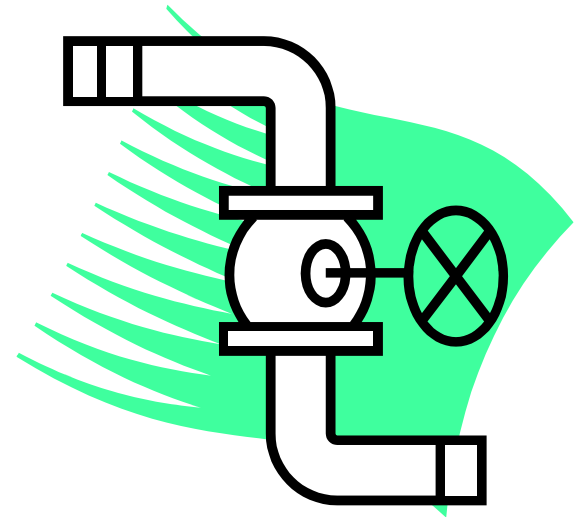
# Mailslots

---

- Mailslots are supported by three specialized functions: **CreateMailslot**, **GetMailslotInfo**, and **SetMailslotInfo**.

# Pipe

- CreatePipe
- CreateNamedPipe
- ReadFile
- WriteFile





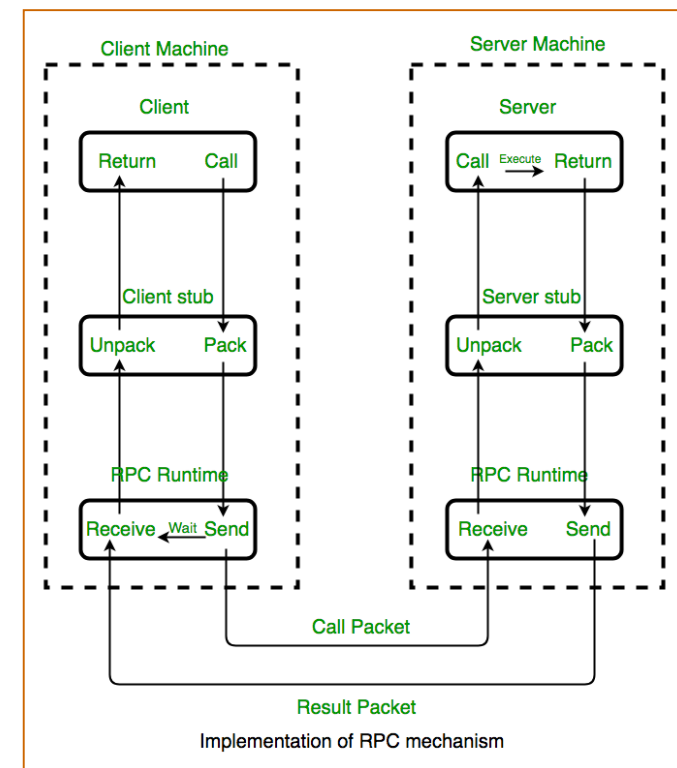
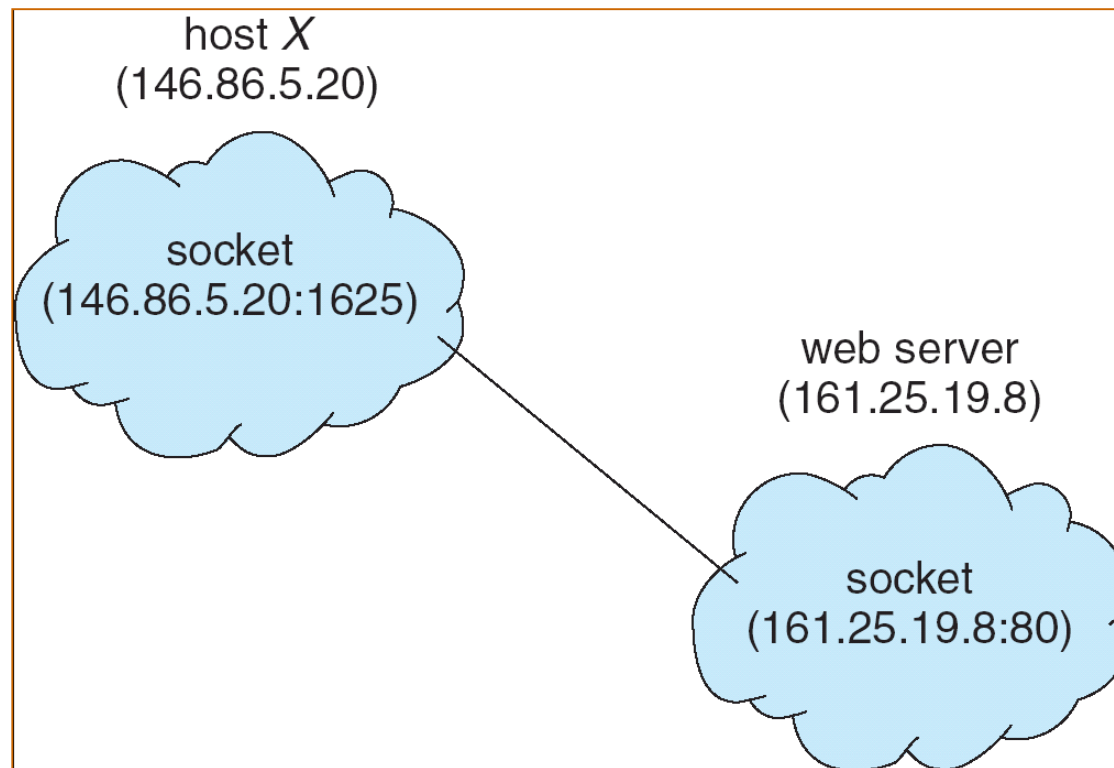
# Communication in Client-Server Systems

---

# Client-Server Communication

- Sockets
- Remote Procedure Calls

<https://www.cnblogs.com/swordfall/p/8683905.html>

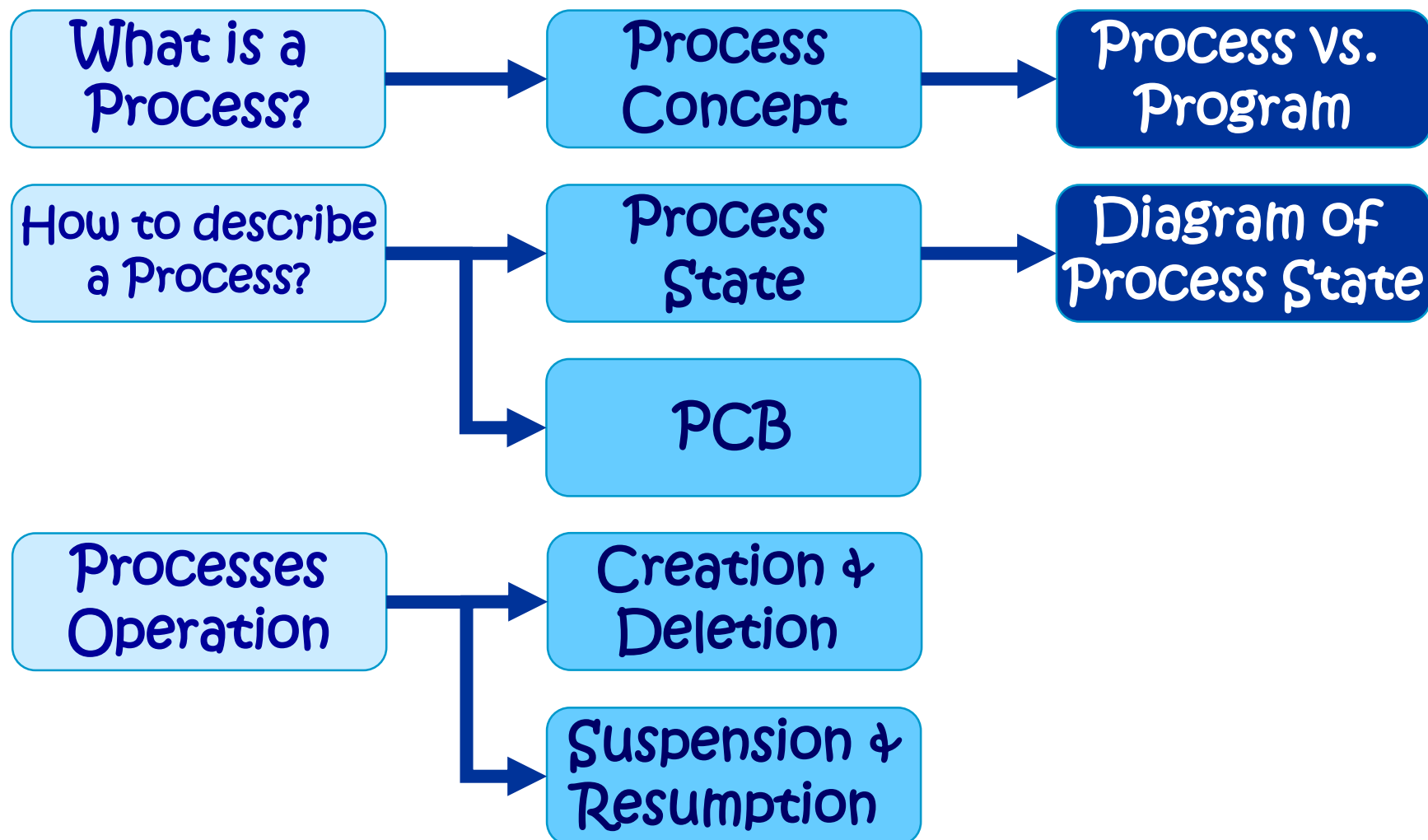




# Summary

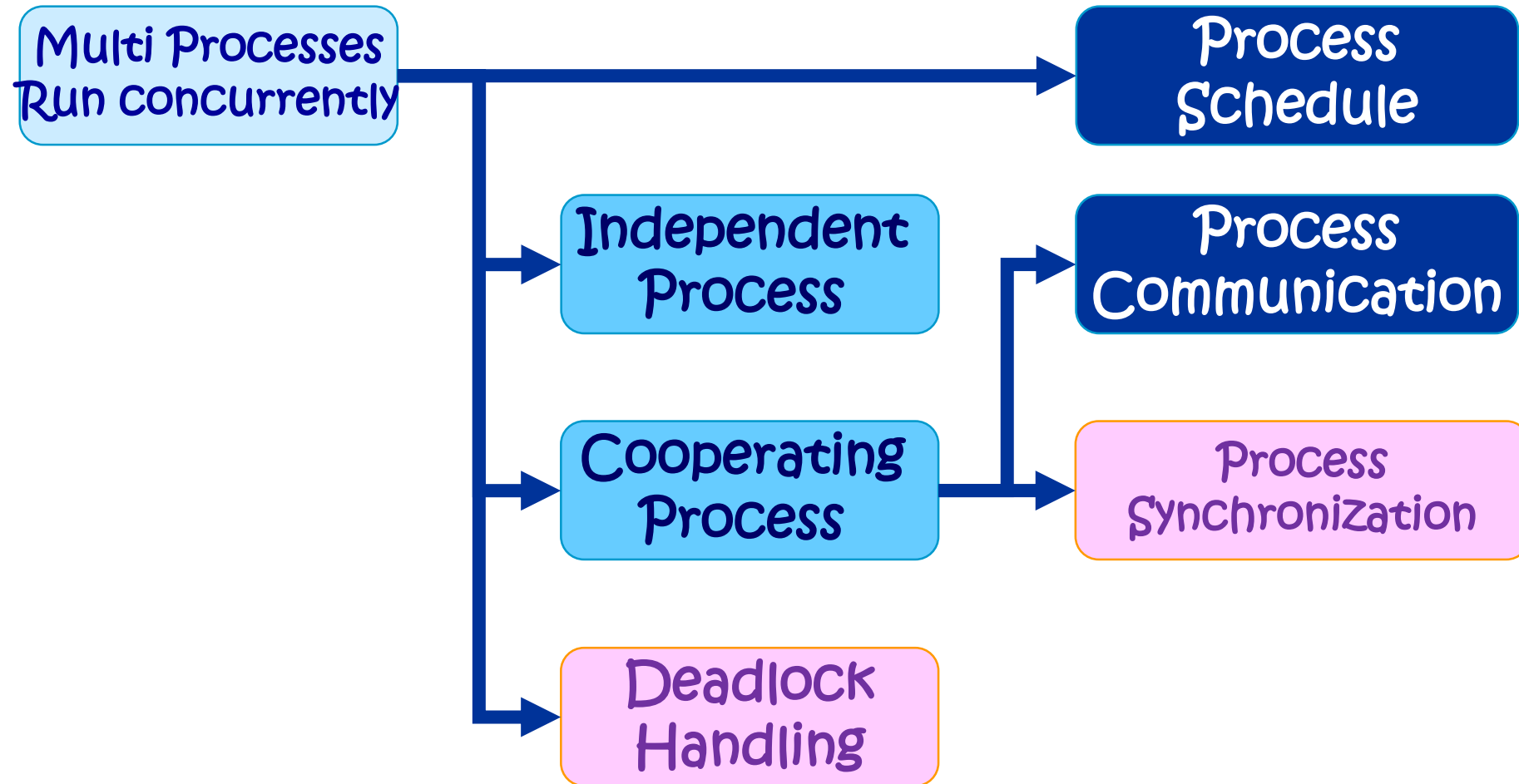
---

# Process

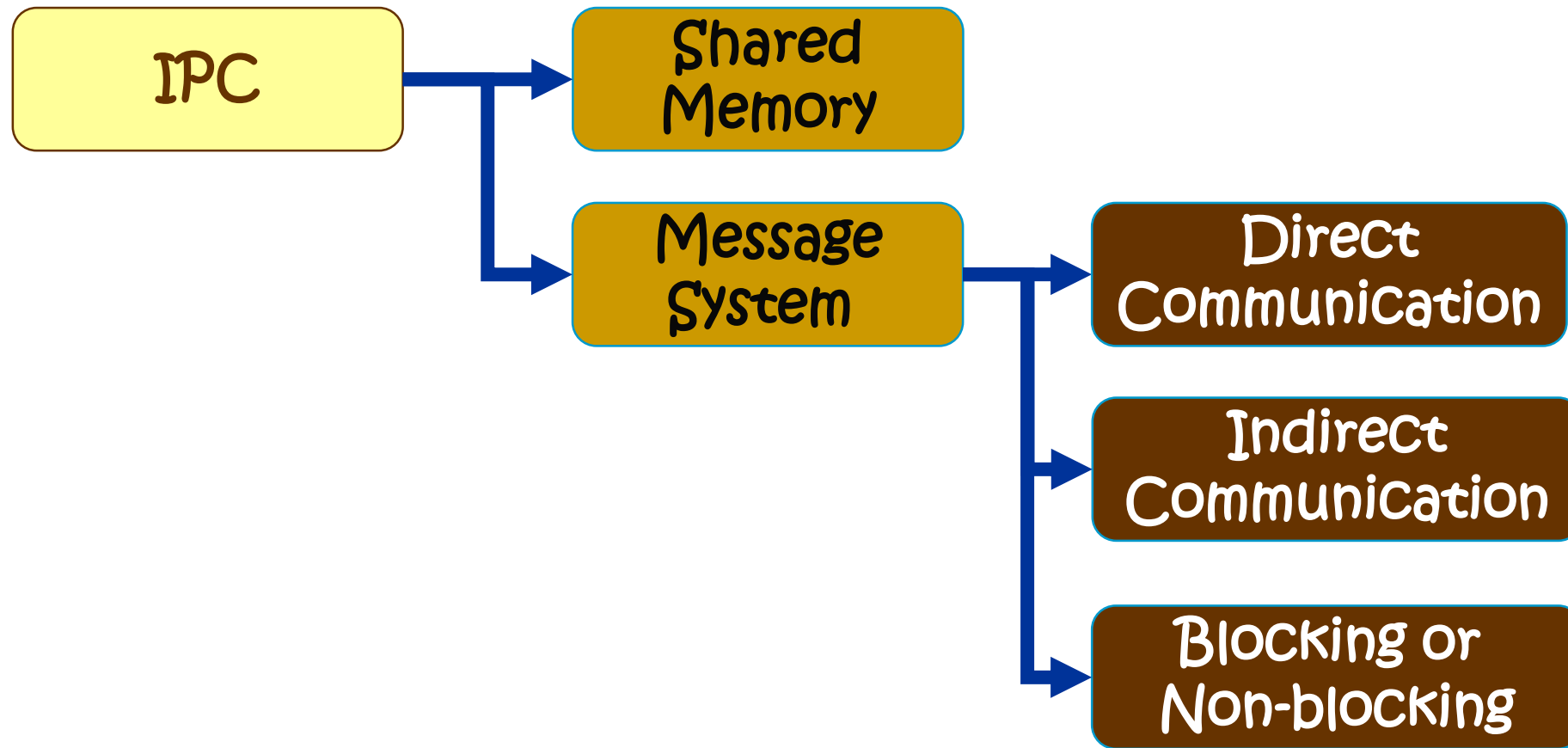




# Process/Thread



# Interprocess Communication





北京交通大学

# Thank you !

Q & A

