IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

进程间通信

上海交通大学并行与分布式系统研究所

https://ipads.se.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

回顾：调度

调度小结

- **调度指标**
  - 响应时间、周转时间、资源利用率、公平性 …
- **调度策略**
  - FCFS、SJF、RR
  - 优先级调度、MLFQ
  - 公平共享调度
- **多核调度**
  - CPU核心亲和性
  - 负载均衡

Schedule

Assume we have the following two jobs in the workload and **no I/O issues** are involved. Please fill in the following tables with the execution of CPU when we decide to use different schedule policies respectively. Suppose when a job arrives, it is added to the tail of a work queue. The RR policy selects the next job of the current job in the queue. The **RR** time-slice is 2ms. (**NOTE**: Time 0 means the task running during [0ms,1ms])

Job	Arrival Time	Length of Run-time
A	0ms	4ms
B	2ms	2ms
C	5ms	3ms
D	9ms	4ms

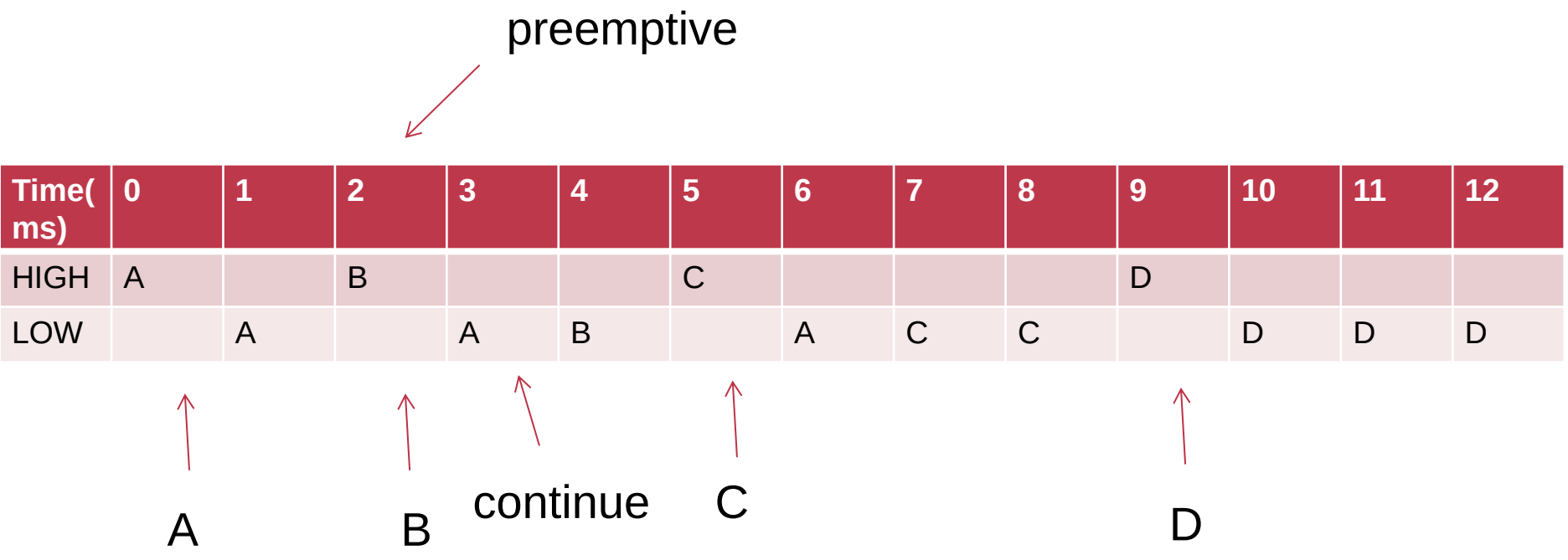
Schedule

2. We decide to use **MLFQ** scheduling policy with **two priority queues**, the highest one has time-slice of **1ms**, the lowest one has time-slice of **2ms**. We use RR in each queue and priority boost **isn't supported**. Following table shows the execution of CPU. Please fill in the blanks. (8′)

Time(ms)	**0**	**1**	**2**	**3**	**4**	**5**	**6**
CPU	A	A	[1]	[2]	[3]	[4]	A
Time(ms)	**7**	**8**	**9**	**10**	**11**	**12**	
CPU	C	[5]	[6]	[7]	[8]	D	

3. Please calculate the average turnaround time and average response time of the **MLFQ** scheduling policy we mentioned in the previous problem. (4′)

Schedule


preemptive
Time(ms) 0 1 2 3 4 5 6 7 8 9 10 11 12
HIGH A B C D
LOW A A B A C C D D D
A
B
continue
C
D

Schedule

preemptive

Time(ms)	0	1	2	3	4	5	6	7	8	9	10	11	12
HIGH	A		B			C				D			
LOW		A		A	B		A	C	C		D	D	D

A B continue C D

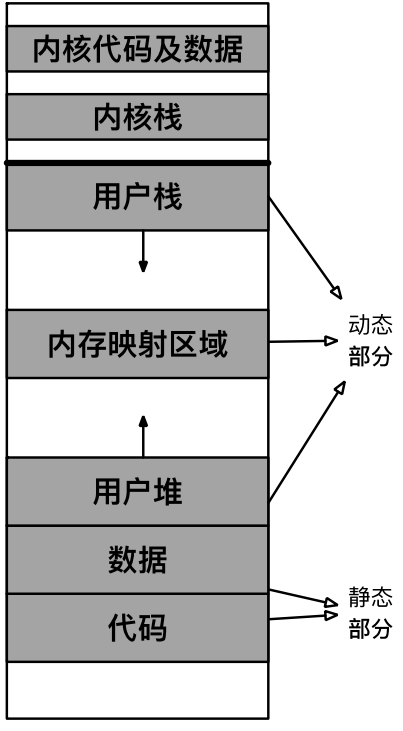
Turnaround Time = ((7-0)+(5-2)+(9-5)+(13-9)) / 4 = 4.5ms

Response Time = ((0-0)+(2-2)+(5-5)+(9-9)) / 4 = 0ms

回顾: 进程

- **进程是计算机程序运行时的抽象**
  - 静态部分: 程序运行需要的代码和数据
  - 动态部分: 程序运行期间的状态
    (程序计数器、堆、栈……)

- **进程具有独立的虚拟地址空间**
  - 每个进程都具有"独占全部内存"的假象
  - 内核中同样包含内核栈和内核代码、数据



使用多个进程的应用程序/系统

- **应用程序/系统可选择使用不同进程来运行不同模块**
  - 优势-1：功能模块化，避免重复造轮子（如数据库、界面绘制）
  - 优势-2：增强模块间隔离，增强安全保障（敏感数据的隔离）
  - 优势-3：提高应用容错能力，限制故障在模块间的传播
- **然而不同进程拥有不同的内存地址空间**
  - 进程与进程之间无法直接进行通信
  - 需要一种进程间通信的方式
    - **IPC**：Inter-Process Communication

示例：基于管道的进程间通信

当然可以！下面是一个简单的用Linux管道实现的示例代码：

```
#!/bin/bash

# 创建一个用于传输数据的命名管道
mkfifo mypipe

# 将命令 "ls" 的输出重定向到管道
ls -l > mypipe &

# 将管道中的数据传输给命令 "grep" 进行过滤
grep "example" < mypipe

# 删除命名管道
rm mypipe
```

以上示例代码通过创建一个命名管道 `mypipe`，将命令 `ls` 的输出重定向至管道，并在后台运行。然后，从管道中读取数据，并使用 `grep` 命令过滤出包含关键字 "example" 的行。

请注意，上述代码仅作为示例，实际使用时需要根据具体需求进行修改和扩展。在使用管道时，确保管道的写端和读端配对使用，并在完成后删除管道以释放资源。

管道pipe

- **管道是Linux等系统中常见的进程间通信机制**
  - 两个进程间的一根通信通道（无格式的数据）
  - 一端向里投递，另一端接收
- **命令行常用: 常见的命令 ls | grep**

```
➜  os-textbook git:(master) ls | grep ipc
ipc.tex
```

- **Linux系统调用**
  - pipe() 用于创建一个匿名管道
  - mkfifo() 用于创建一个命名管道

示例：基于消息队列的进程间通信

```c
// 定义消息结构体
struct message {
    long mtype;
    char mtext[100];
};

int main() {
    // 创建消息队列
    key_t key = ftok(".", 'a');
    int msg_id = msgget(key, IPC_CREAT | 0666);

    // 准备消息
    struct message msg;
    msg.mtype = 1;  // 消息类型
    sprintf(msg.mtext, "Hello, Process 2!");  // 消息内容

    // 发送消息
    msgsnd(msg_id, &msg, sizeof(msg.mtext), 0);

    // 关闭消息队列
    msgctl(msg_id, IPC_RMID, NULL);

    return 0;
}
```

发送进程

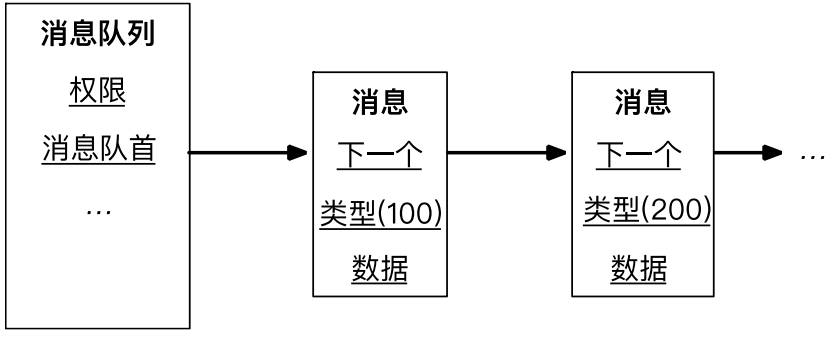
```c
// 定义消息结构体
struct message {
    long mtype;
    char mtext[100];
};

int main() {
    // 获取消息队列
    key_t key = ftok(".", 'a');
    int msg_id = msgget(key, IPC_CREAT | 0666);

    // 接收消息
    struct message msg;
    msgrcv(msg_id, &msg, sizeof(msg.mtext), 1, 0);

    // 打印消息内容
    printf("Received message: %s\n", msg.mtext);

    // 关闭消息队列
    msgctl(msg_id, IPC_RMID, NULL);

    return 0;
}
```

接收进程

消息队列: 带类型的消息传递

ftok();
msgget();
msgsnd();
msgrcv();
msgctl();

- **消息队列: 以链表的方式组织消息**
  - 信箱为内核中维护的消息队列结构体
  - 任何有权限的进程都可以访问队列，写入或者读取
  - 支持异步通信 (非阻塞)



- **消息的格式: 类型 + 数据**
  - 类型：由一个整型表示，具体的意义由应用程序决定

消息队列：带类型的消息传递

- **消息队列的组织**
  - 基本遵循FIFO (First-In-First-Out)先进先出原则
  - 消息队列的写入：增加在队列尾部
  - 消息队列的读取：默认从队首获取消息
- **允许按照类型查询: Recv(A, type, message)**
  - 类型为0时返回第一个消息 (FIFO)
  - 类型有值时按照类型查询消息
    - 如type为正数，则返回第一个类型为type的消息

消息队列 VS. 管道

- **消息格式:**
  - 消息队列: 带类型的数据
  - 管道: 数据 (字节流)
- **通信进程数量:**
  - 消息队列: 可以有多个发送者和接收者
  - 管道: 两个端口，最多对应两个进程
- **消息的管理:**
  - 消息队列: FIFO + 基于类型的查询
  - 管道: FIFO
- **缓存区设计:**
  - 消息队列: 链表的组织方式，动态分配资源，可以设置很大的上限
  - 管道: 固定的缓冲区间，分配过大资源容易造成浪费

示例：基于信号Signal的通信

```c
void handle_signal(int signum) {
    if (signum == SIGUSR1) {
        printf("Received SIGUSR1 signal\n");
    } else if (signum == SIGUSR2) {
        printf("Received SIGUSR2 signal\n");
    }
}

int main() {
    pid_t pid = getpid();
    printf("Process ID: %d\n", pid);

    // 注册信号处理函数
    signal(SIGUSR1, handle_signal);
    signal(SIGUSR2, handle_signal);

    while (1) {
        printf("Waiting for signal...\n");
        sleep(1);
    }

    return 0;
}
```

Shell> kill –SIGUSER1 PID

示例：共享内存通信

- **基础实现: 共享区域**

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
volatile int buffer_write_cnt = 0;
volatile int buffer_read_cnt = 0;
volatile int empty_slot = BUFFER_SIZE;
volatile int filled_slot = 0;
```

共享数据区域，容量为10

共享状态

volatile关键字的目的?

基于共享内存的生产者消费者问题实现

- **基础实现: 发送者 (生产者)**

```
while (true) {
    /* Produce an item/msg */
    while (empty_slot == 0)
        ;   /* do nothing -- no free buffers */
    empty_slot --;
    buffer[buffer_write_cnt] = msg;
    buffer_write_cnt = (buffer_write_cnt + 1) % BUFFER_SIZE;
    filled_slot ++;
    …
}
```

当没有空间时，发送者盲等

发送者放置消息

基于共享内存的生产者消费者问题实现

- **基础实现: 接收者**

当没有新消息时，接收者盲目等待

```
while (true) {
    while (filled_slot == 0)
        ; // do nothing -- nothing to consume
    filled_slot--; // remove an item from the buffer
    item = buffer[buffer_read_cnt];
    buffer_read_cnt = (buffer_read_cnt + 1) % BUFFER SIZE;
    empty_slot++;
    return item;
}
```

接收者获取消息

共享内存通信方式的不足

- **缺少通知机制**
  - 若轮询检查，则导致CPU资源浪费
  - 若周期性检查，则可能导致较长的等待时延
  - 根本原因：共享内存的抽象过于底层；缺少OS更多支持
- **TOCTTOU （Time-of-check to Time-of-use）问题**
  - 当接收者直接用共享内存上的数据时，可能存在被发送者恶意篡改的情况（发生在接收者检查完数据之后，使用数据之前）
  - 这可能导致buffer overflow等问题

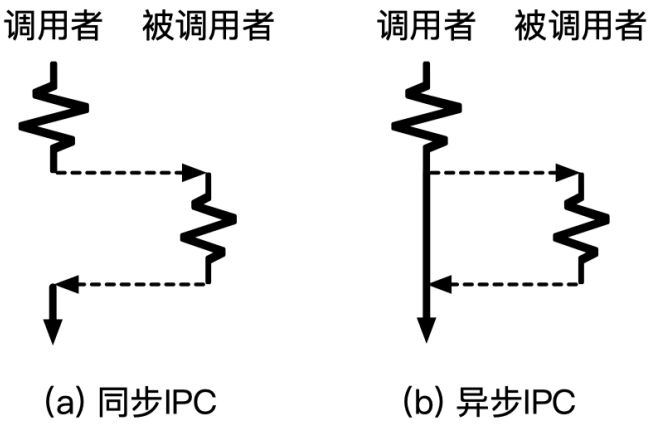
常见IPC的类型

IPC机制	数据抽象	参与者	方向
管道	文件接口	两个进程	单向
共享内存	内存接口	多进程	单向/双向
消息队列	消息接口	多进程	单向/双向
信号	信号接口	多进程	单向
套接字	文件接口	两个进程	单向/双向

IPC的两种通信连接

- **方法-1：直接通信**
  - 通信的一方需要显示地标识另一方，每一方都拥有唯一标识
  - 如：Send(P, message), Recv(Q, message)
  - 连接的建立是自动完成的（由内核完成）
- **方法-2：间接通信**
  - 通信双方通过"信箱"的抽象来完成通信
  - 每个信箱有自己唯一的标识符
  - 通信双方并不直接知道在与谁通信
  - 进程间连接的建立发生在共享一个信箱时

IPC控制流：同步和异步



- **同步IPC**
  - IPC操作会阻塞进程直到操作完成
  - 线性的控制流
  - 调用者继续运行时，返回结果已经就绪
- **异步IPC**
  - 进程发起IPC操作后即可返回而不需要等待其完成
  - 通过轮询或回调函数（需内核支持）来获取返回结果

IPC基本设计与实现

IPC的基本功能目标

- **数据传递**
  - 发送消息：Send
  - 接收消息：Recv
  - 例如：pipe，msgqueue，网络socket， 消息订阅发布publish/subscribe
- **过程调用**
  - 包括数据传递
  - 远程方法调用：RPC
  - 调用结果返回：Reply
  - 例如：进程1调用进程2的方法（与远程过程调用RPC类似）

简单IPC：消息的发送

```
1  // IPC 的发送者
2  int main(void)
3  {
4    Message msg;
5    // chan 表示发送者和消费者之间的一个“通信连接”
6    Channel chan = simple_ipc_channel(...);
7    // 按照语义生成请求消息
8    msg = construct_request(...);
9
10   // 通过通信连接发送一个消息出去
11   Send(chan, &msg);
12   ...
13 }
```

发送者和消费者需要依赖于一个通信连接channel，作为一个媒介进行消息的传输

简单IPC：消息的接收

```
1  // IPC 的接收者
2  int main(void)
3  {
4    Message msg;
5    // chan 表示发送者和消费者之间的一个“通信连接”
6    Channel chan = simple_ipc_channel(...);
7
8    while (1) {
9      // 等待一个消息的到来，这里会收到 Send 发送的消息
10     Recv(chan, &msg);
11     // 处理消息
12     results = handle_msg(&msg);
13     ...
14   }
15   ...
16 }
```

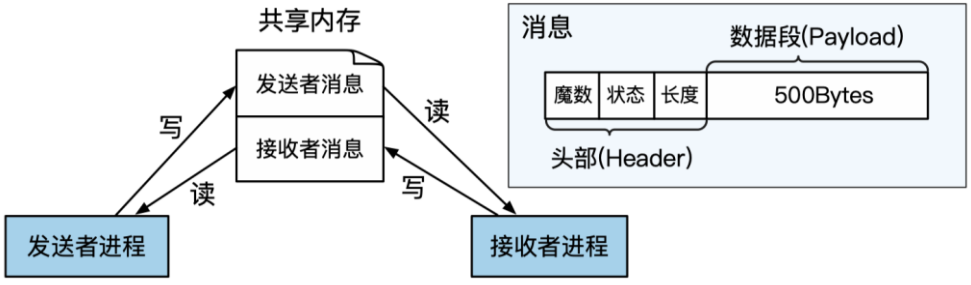
简单IPC：消息的远程方法调用（发送者）

```c
#include <simple-ipc.h> // 使用后续章节的简单 IPC 设计
...

// IPC 的发送者
int main(void)
{
  Message req_msg, resp_msg;
  // chan 表示发送者和消费者之间的一个“通信连接”
  Channel chan = simple_ipc_channel(...);
  // 按照语义生成请求消息
  req_msg = construct_request(...);

  // 以 RPC 的方式调用接收者，并阻塞等待一个结果的返回
  RPC(chan, &req_msg, &resp_msg);

  printf("The response is: %s", msg_to_str(resp_msg));
  ...
}
```

简单IPC：消息的远程方法调用（接收者）

```c
#include <simple-ipc.h> // 使用后续章节的简单 IPC 设计
...

// IPC 的接收者
int main(void)
{
  Message req_msg, resp_msg;
  // chan 表示发送者和消费者之间的一个“通信连接”
  Channel chan = simple_ipc_channel(...);

  while (1) {
    // 等待一个消息的到来
    Recv(chan, &req_msg);
    // 处理消息并构建结果消息
    resp_msg = handle_msg(&req_msg);
    Reply(chan, &resp_msg);
  }
  ...
}
```

简单IPC的两个阶段



- **阶段-1：准备阶段**
  - 建立通信连接，即进程间的信道
    - 假设内核已经为两个进程映射了一段共享内存
- **阶段-2：通信阶段**
  - 数据传递
    - “消息”抽象：通常包含头部（元数据）和数据内容（例如500字节）
  - 通信机制
    - 两个消息保存在共享内存中：发送者消息、接收者消息
    - 发送者和接收者通过**轮询**消息的状态作为通知机制

是否能够包含指针？

简单IPC数据传递的两种方法

- **方法-1：通过共享内存的数据传递**
  - 操作系统在通信过程中不干预数据传输
  - 操作系统仅负责准备阶段的映射

- **方法-2：通过操作系统内核的数据传递**
  - 操作系统提供接口（系统调用）
  - 通过内核态内存来传递数据，无需在用户态建立共享内存

两种数据传递方法的对比

- **基于共享内存的优势**
  - 完全由用户态控制，定制能力更强
  - 无需内核进行额外的内存拷贝

思考：如何避免TOCTOU?

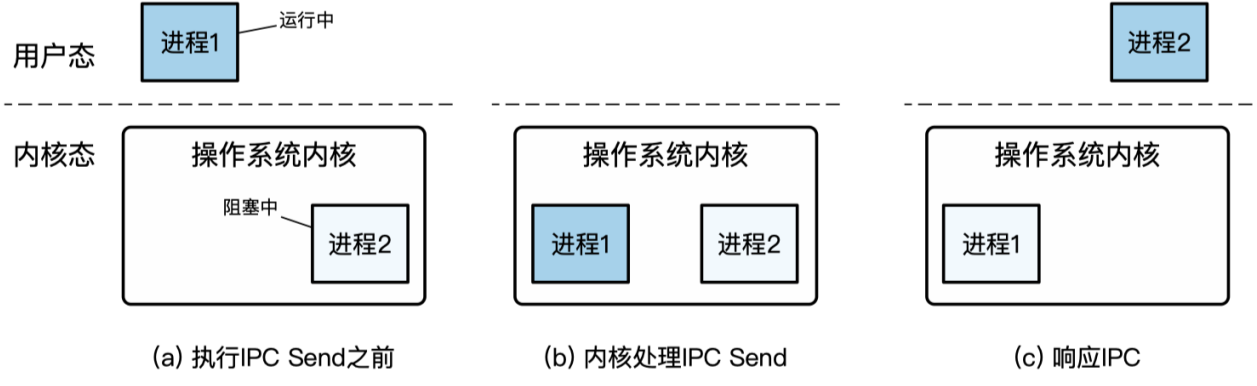
思考：如何避免内存拷贝?

- **基于系统调用的优势**
  - 抽象更简单，用户态直接调用接口，使用更方便
  - 安全性保证更强，发送者在消息被接收时通常无法修改消息

简单IPC的通知机制

- **方法-1：基于轮询（消息头部的状态信息）**
  - 缺点：大量CPU计算资源的浪费
- **方法-2：基于控制流转移**
  - 由内核控制进程的运行状态
  - 优点：进程只有在条件满足的情况下才运行，避免CPU浪费

思考：相比于方法2，方法-1有什么优势？



(a) 执行IPC Send之前　(b) 内核处理IPC Send　(c) 响应IPC

IPC的超时机制

- **一种新的错误：超时**
  - 传统的函数调用不存在超时问题
  - IPC涉及两个进程，分别有独立的控制流
- **超时可能的原因**
  - 被调用者是恶意的：故意不返回
  - 被调用者不是恶意的：运行时间过长、调度时间过长、请求丢失等
- **超时机制**
  - 应用可自行设置超时的阈值，但如何选择合适的阈值却很难
  - 特殊的超时机制：阻塞、立即返回（要求被调用者处于可立即响应的状态）

小知识：IPC的命名服务

- **命名服务：一个单独的进程**
  - 类似一个全局的看板，协调服务端与客户端之间的信息
  - 服务端可以将自己提供的服务注册到命名服务中
  - 客户端可以通过命名服务进程获取当前可用的服务
- **命名服务的功能：分发权限**
  - 例如：文件系统进程允许命名服务将连接文件系统的权限任意分发，因此所有进程都可以访问全局的文件系统
  - 例如：数据库进程只允许拥有特定证书的客户端连接
- **权限控制：A进程能够向哪些进程发起IPC**

以ChCore IPC为例

微内核进程间通信代码讲解

ChCore进程间通信

- **同步通信**
- **传递数据：共享内存**
  - 进程1与进程2共享一块内存
- **传递控制流：内核协助**
  - 线程A（隶属于进程1）切换到线程B （隶属于进程2）