政大資科系

作業系統

Operating System

廖峻鋒

cfliao@nccu.edu.tw

Operating System

Processes

Chun-Feng Liao

廖峻鋒

Department of Computer Science

National Chengchi University

Outline

- Process Structure
- Process Scheduling
- Process Operation
- Process Communication

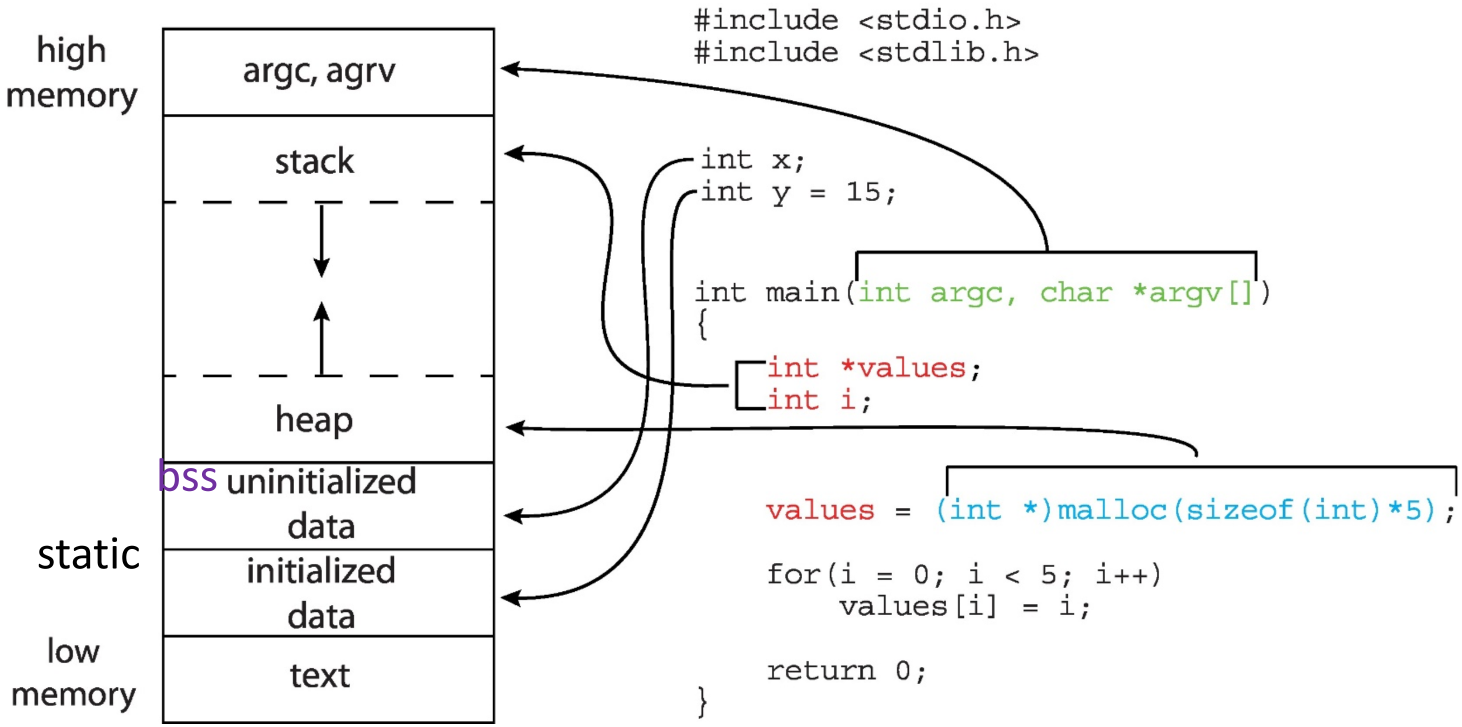
Process and Program

- Process is active
  - A program in execution
- Program is passive
  - A binary file stored in the disk
- Contents
  - Status: PC and other registers' values
  - Text (code)
  - Data (static / global variables)
  - Heap (動態配置的資料型態)
  - Stack (activation records)

Memory layout of a process

The activation record stores the parameters, local variables, return address of a function call

Memory Layout of a C Program



Demo: size -A

Understanding the Memory Layout of Linux Executables
https://gist.github.com/CMCDragonkai/10ab53654b2aa6ce55c11cfc5b2432a4

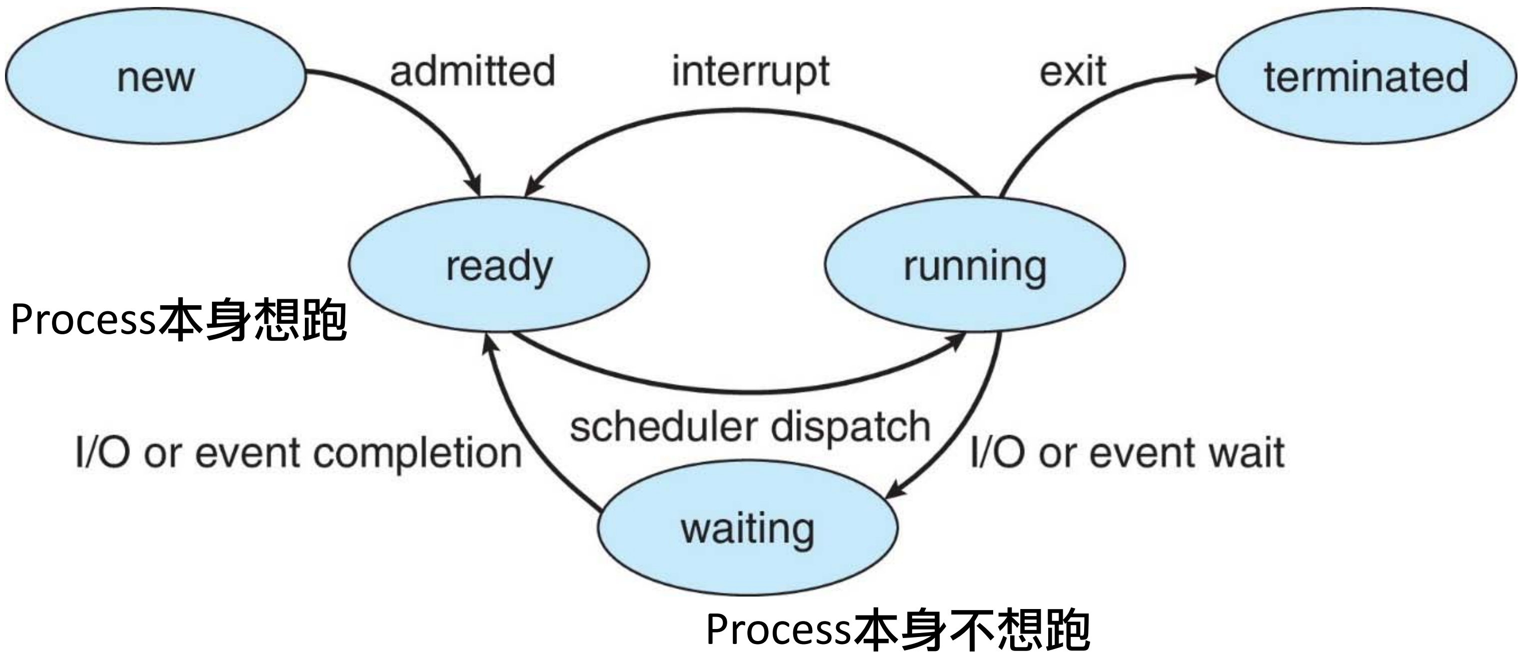
P.107, 3.1.2

Process state

- New
  - the process is being created
- Ready (沒在跑，但有意願跑)
  - the process is in the memory waiting to be assigned to a processor
- Running (正在跑)
  - instructions are being executed by CPU
- Waiting (沒在跑，也沒意願跑)
  - the process is waiting for events to occur
- Terminated
  - the process has finished execution

Process state

- One process is running on a processor at any instant
- Many processes may be ready or waiting



P. 109 3.1.3

Process Control Block (PCB)

- Kernel中維護的Process資訊，每個Process都有一個PCB
  - Process state
  - Program counter
    Linux中稱為EIP (instruction pointer)
  - CPU registers
  - CPU scheduling information (e.g. priority)
  - Memory-management information (e.g. base/limit register)
  - I/O status information
  - Accounting information

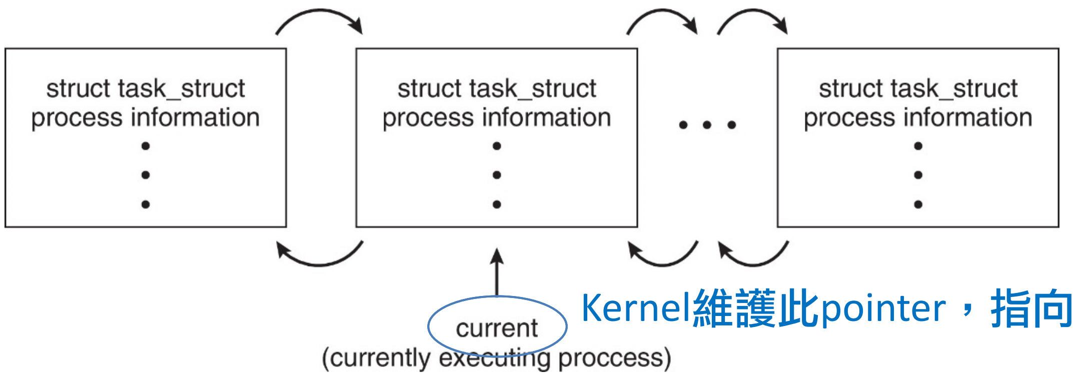
See /proc/process id
在/proc/<pid>/stat中, see: https://man7.org/linux/man-pages/man5/proc.5.html (search:/proc/[pid]/stat)

PCB in Linux

Represented by the structure `task_struct`

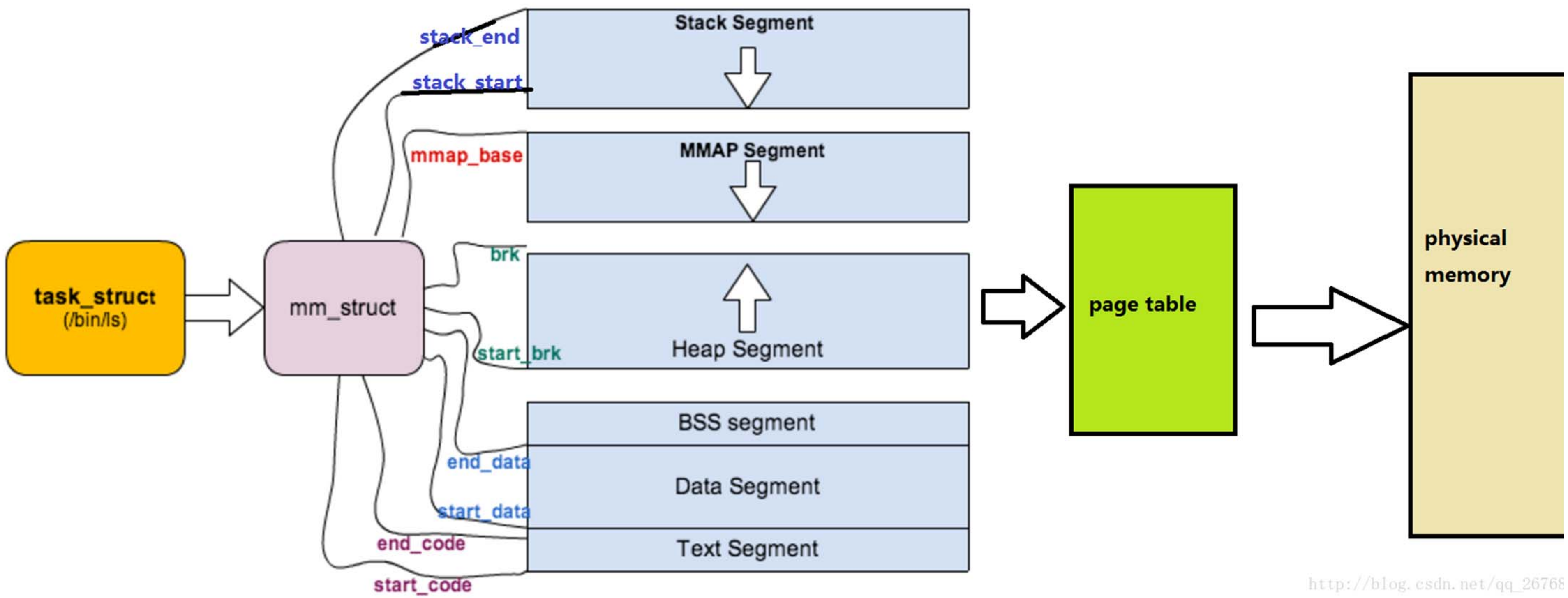
```
pid t_pid;                     /* process identifier */
long state;                    /* state of the process */
unsigned int time_slice        /* scheduling information */
struct task_struct *parent;/* this process' s parent */
struct list_head children; /* this process' s children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;          /* address space of this process */
```

(#L876) 指向Process實際存在的記憶體位址



Kernel維護此pointer，指向正在執行的process

https://github.com/torvalds/linux/blob/master/include/linux/sched.h#L743

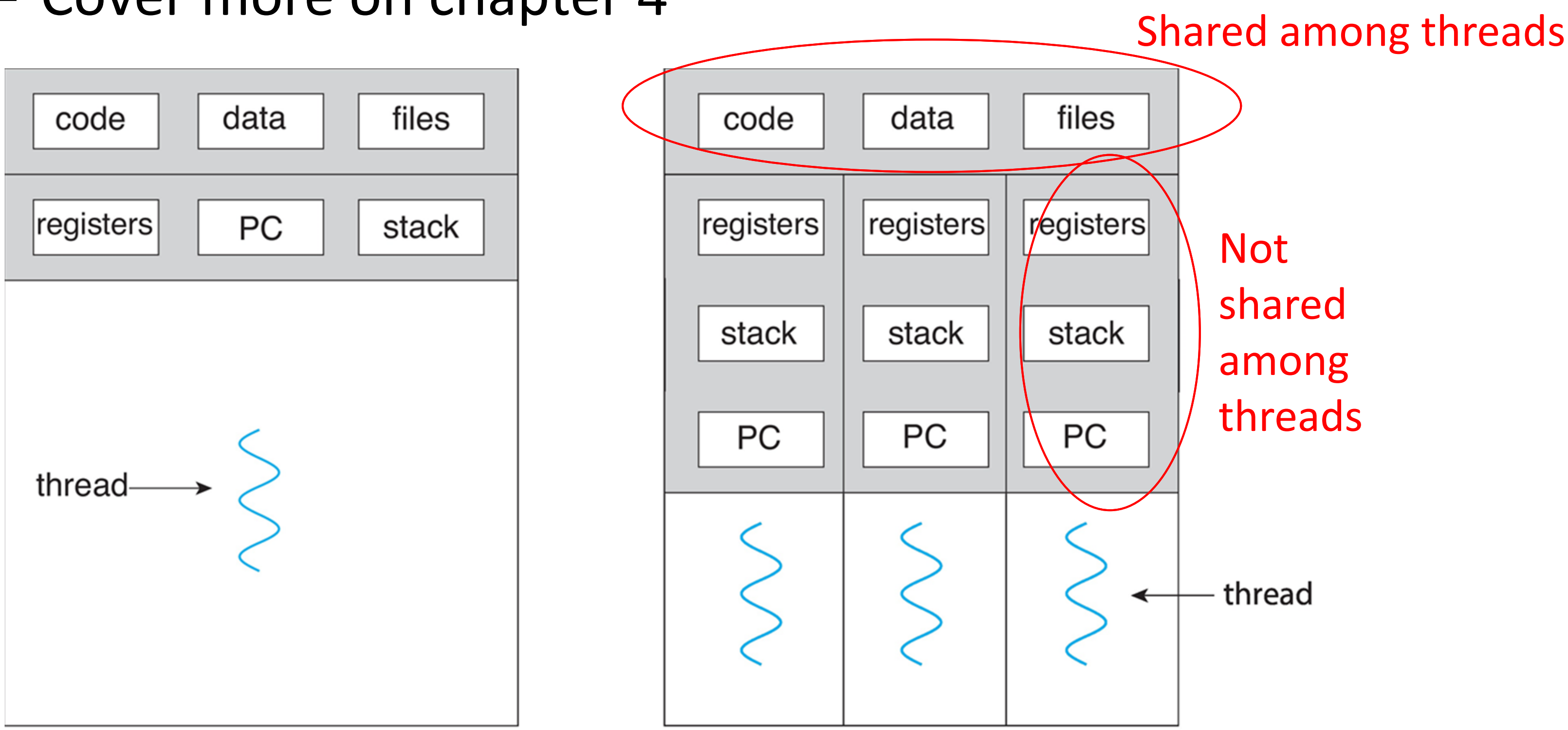


https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h

P. 110, P.160

Thread

- A process may have multiple PC and register states
  - Each represent a single thread of execution
  - Cover more on chapter 4

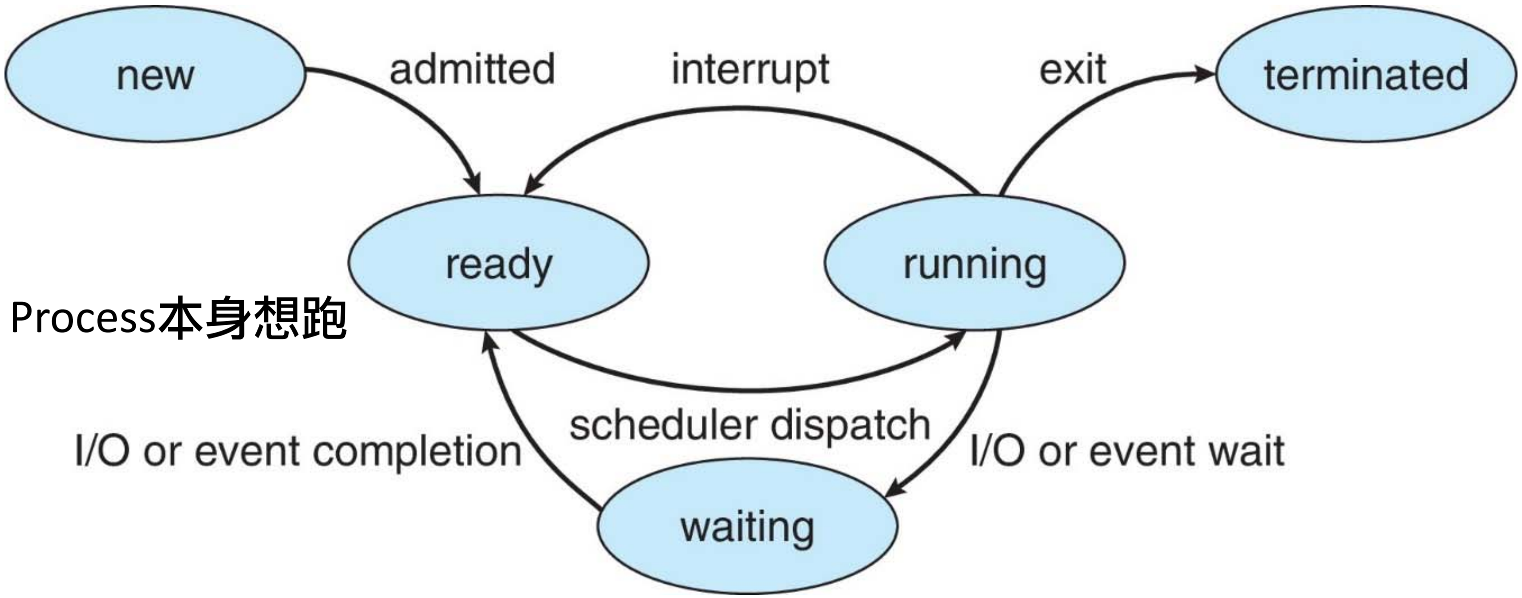


single-threaded process

multithreaded process

Process Scheduling

- Multiprogramming
  - CPU runs process at all times to maximize CPU utilization
  - Degree of multiprogramming: how many process in memory
- Time sharing
  - Switch CPU frequently such that users can interact with each program while it is running
- I/O bound vs. CPU bound
  - Most of the process running time spent on I/O or computation



Process本身想跑

Process本身不想跑

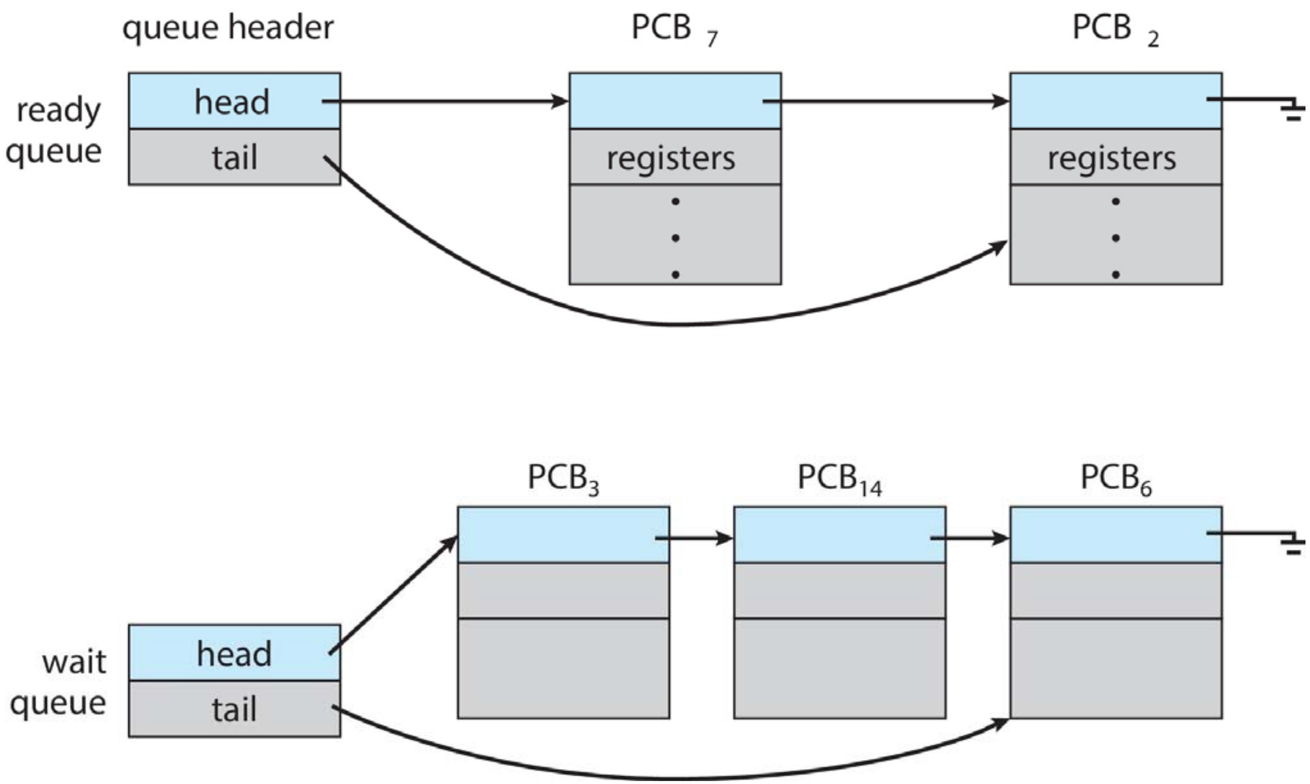
I/O bound的process可能花很多時間在這裡

P. 112

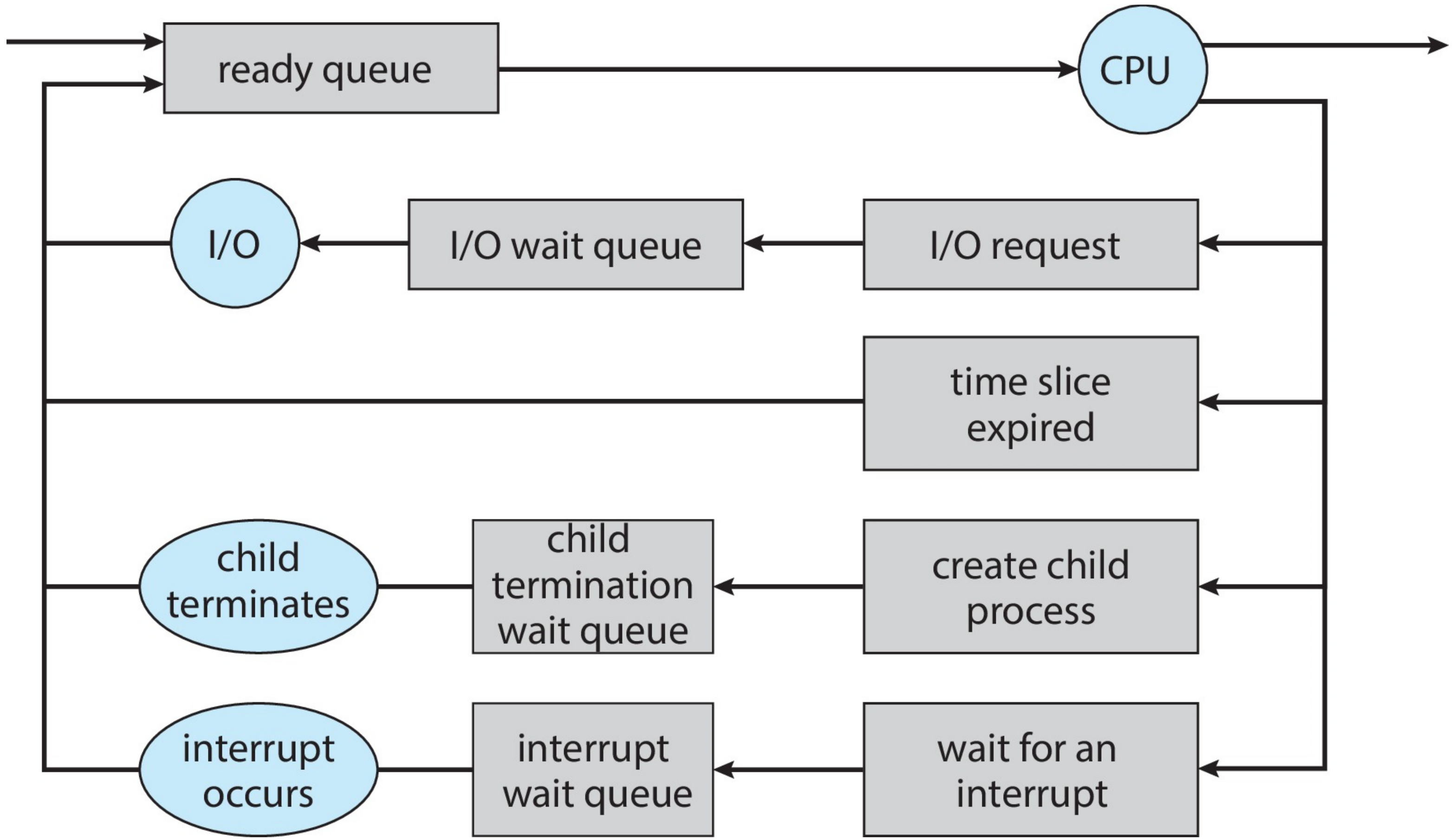
Process Scheduling Queues

- Processes migrate between the various queues (i.e. switch among states)
  - Job queue (New State) – set of all processes in the system
  - Ready queue (Ready State) – set of all processes residing in main memory, ready and waiting to execute
  - Device queue (Wait State)– set of processes waiting for an I/O device

Ready and Wait Queues



Process Scheduling Diagram



Note:此圖顯示了task移出CPU的四種情況

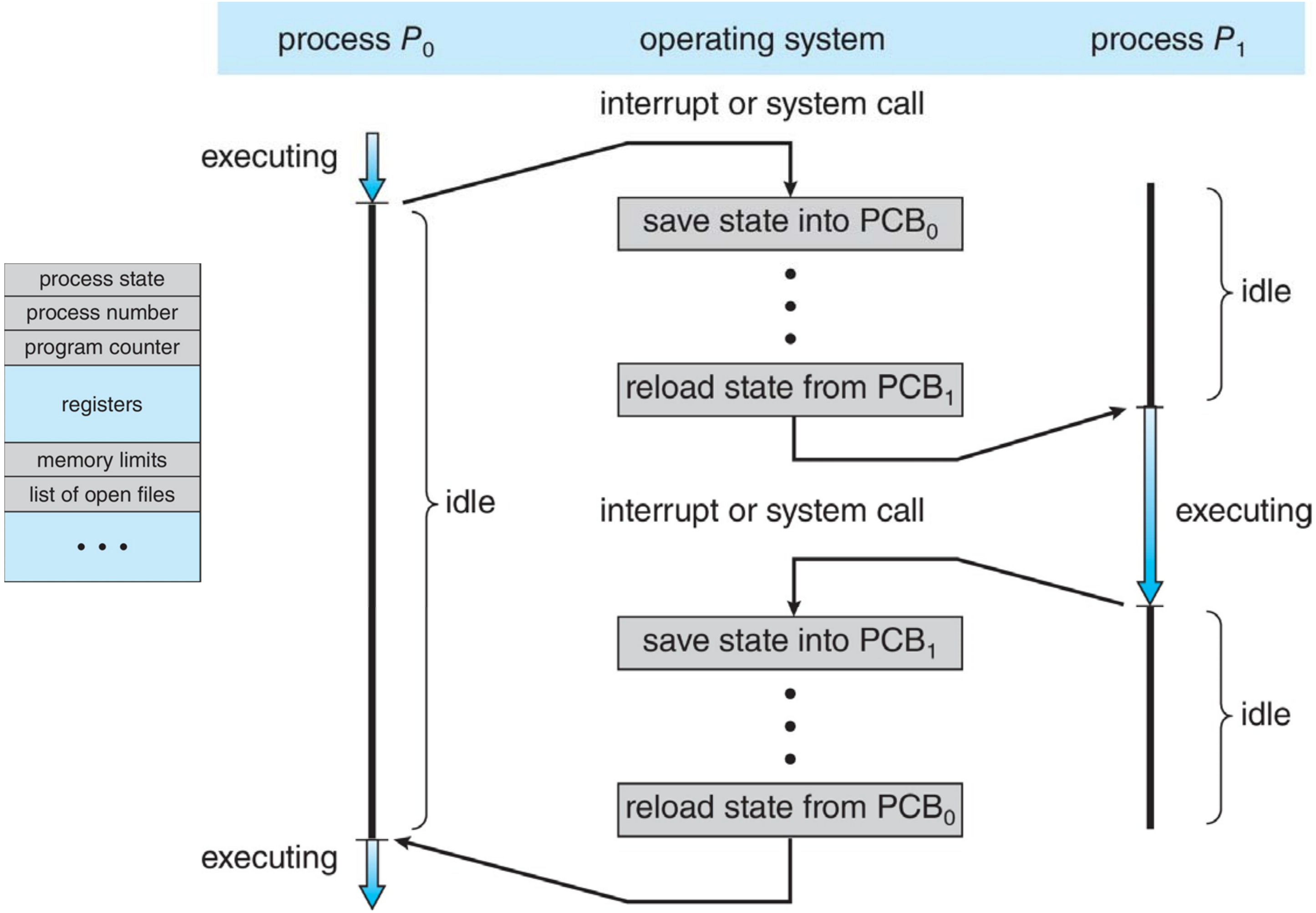
CPU Scheduling

- The role of the CPU scheduler
  - Select among the processes that are in the ready queue and allocate a CPU core to one of them
  - More on chapter 5
- Swapping
  - an intermediate form of scheduling
  - remove a process from memory → reduce the degree of multiprogramming
  - More on chapter 9

Context Switch

- Context Switch
  - Saves the state of the old process
  - Loads the saved state for the new process
  - Context-switch time is purely overhead
- Switch time (about 1~1000 ms) depends on
  - Memory speed
  - Number of registers
  - Existence of special instructions
    - a single instruction to save/load all registers
- Hardware support
  - Prepare multiple sets of registers
  - A context switch only changes the pointer to the register

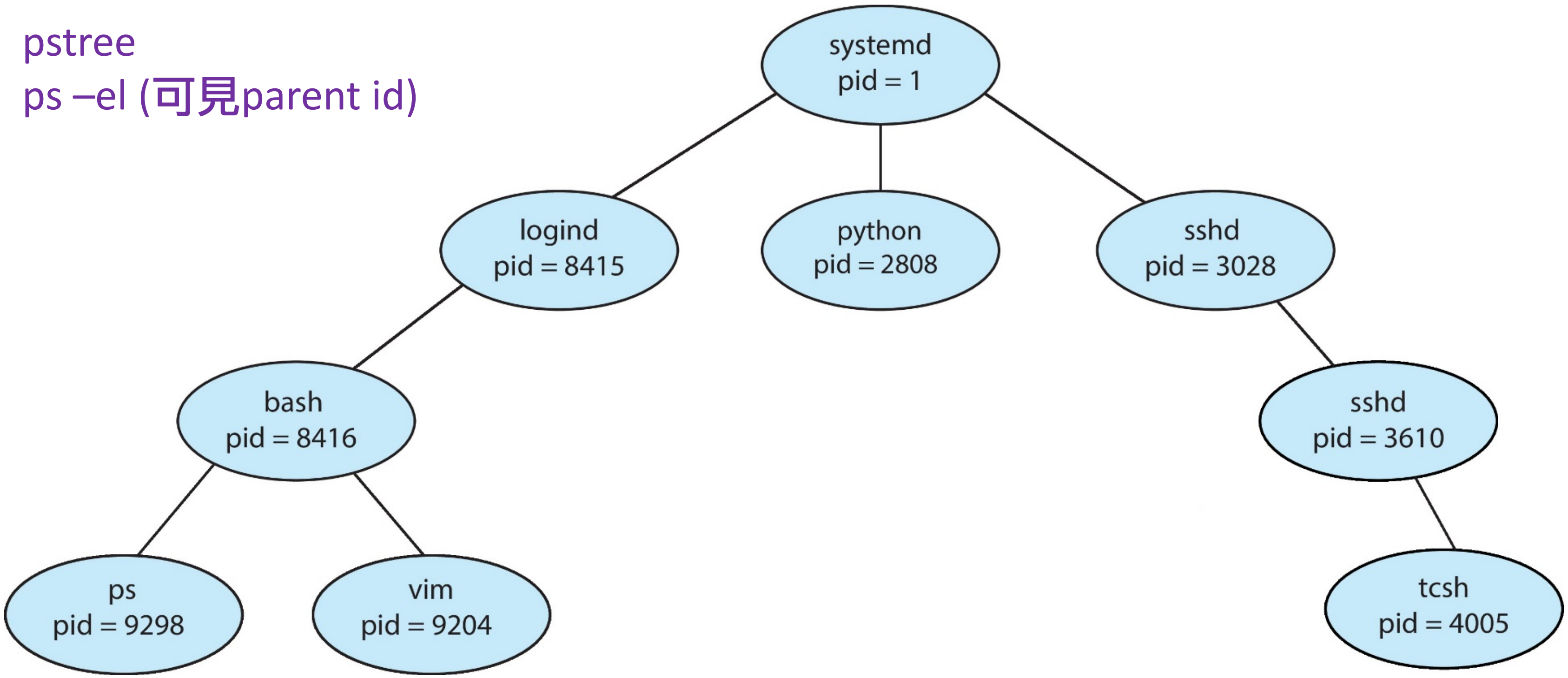
Context Switch



Process Tree

- Each process is identified by a unique process identifier (pid)
- Each process can create new child processes

pstree
ps –el (可見parent id)

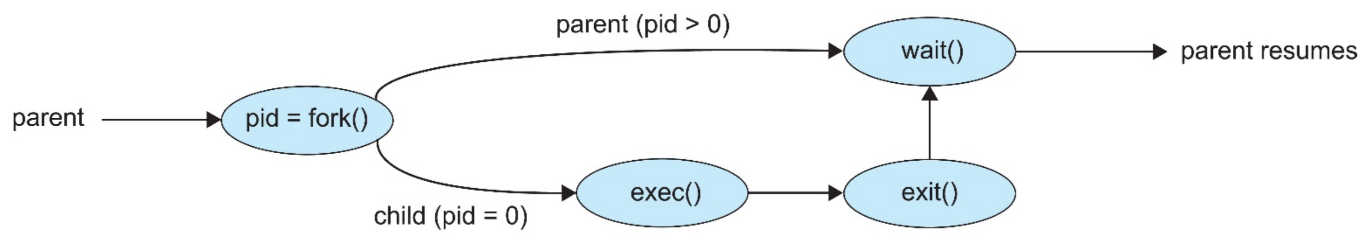


UNIX/Linux Process Creation

- fork system call
  - Create a new (child) process
  - The new process duplicates the address space of its parent
  - Child & Parent execute concurrently after fork
  - Child: return value of fork is 0
  - Parent: return value of fork is PID of the child process
- (child) execlp system call
  - Load a new binary file into memory – destroying the old code
- (parent) wait system call
  - The parent waits for one of its child processes to complete

Process Creation

- An UNIX example
  - fork() system call creates new process
  - execlp() system call used after a fork() to replace the process’ memory space with a new program
  - Parent process calls wait() for the child to terminate



```c
int main()
{
        pid_t child_pid;
        child_pid = fork();

        if (child_pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                return 1;
        }
        else if (child_pid == 0) { /* child process */
                printf("I am the child %d\n", child_pid);
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                printf("I am the parent of %d\n",child_pid);
                wait(NULL);

                printf("Child Complete\n");
        }

   return 0;
}
```

Parent process
child_pid == 1234

```
int main()
{
        pid_t child_pid;
        child_pid = fork();

        if (child_pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                return 1;
        }
        else if (child_pid == 0) { /* child process */
                printf("I am the child %d\n", child_pid);
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                printf("I am the parent of %d\n",child_pid);
                wait(NULL);

                printf("Child Complete\n");
        }

   return 0;
}
```

Child process
child_pid == 0


```
int main()
{
        pid_t child_pid;
        child_pid = fork();

        if (child_pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                return 1;
        }
        else if (child_pid == 0) { /* child process */
                printf("I am the child %d\n", child_pid);
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                printf("I am the parent of %d\n",child_pid);
                wait(NULL);

                printf("Child Complete\n");
        }

   return 0;
}
```

```
int main()
{
        pid_t child_pid;
        child_pid = fork();

        if (child_pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                return 1;
        }
        else if (child_pid == 0) { /* child process */
                printf("I am the child %d\n", child_pid);
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                printf("I am the parent of %d\n",child_pid);
                wait(NULL);

                printf("Child Complete\n");
        }

    return 0;
}
```

Parent process
child_pid == 1234

```c
int main()
{
        pid_t child_pid;
        child_pid = fork();

        if (child_pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                return 1;
        }
        else if (child_pid == 0) { /* child process */
                printf("I am the child %d\n", child_pid);
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                printf("I am the parent of %d\n",child_pid);
                wait(NULL);

                printf("Child Complete\n");
        }

   return 0;
}
```

Child process
child_pid == 0

Load a new binary file into memory – destroying the old code

問題: 會不會輸出test?

```
int main()
{
        pid_t child_pid;
        child_pid = fork();

        if (child_pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                return 1;
        }
        else if (child_pid == 0) { /* child process */
                printf("I am the child %d\n", child_pid);
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                printf("I am the parent of %d\n",child_pid);
                wait(NULL);

                printf("Child Complete\n");
        }

    return 0;
}
```

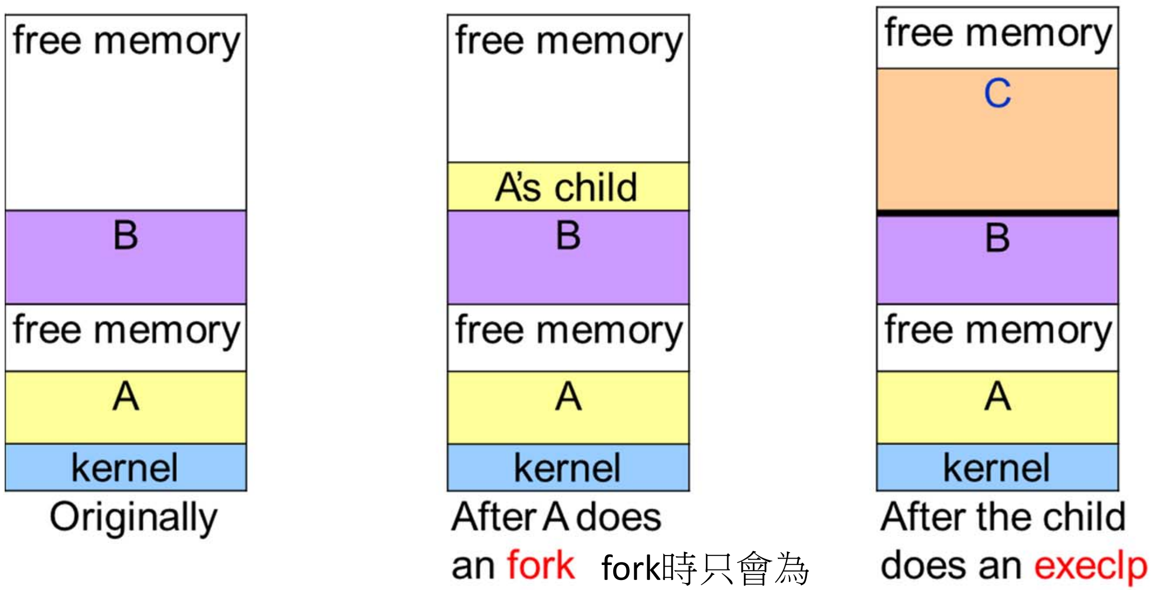
Parent process
child_pid == 1234

Parent and Child Process

- Resource sharing: 3 modes
  - Parent and child processes share all resources
  - Child process shares subset of parent's resources
  - Parent and child share no resources
- Execution: 2 modes
  - 非同步 Parent and children execute concurrently
  - 同步 Parent waits until children terminate
- Address space: 2 modes
  - Child duplicate the parent
  - Child has a program loaded into it (execlp)

UNIX/Linux Process Creation

- Memory space of fork():
  - Old implementation: A's child is an exact copy of parent
  - Current implementation: use copy-on-write technique to store differences in A's child address space



fork時只會為child創建新的stack seg

第10章會再提到copy-on-write

Process Termination

- Terminate when the last statement is executed or exit()
  - All resources of the process, including physical & virtual memory, open files, I/O buffers, are deallocated
- Parent may terminate execution of children processes by specifying its PID (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
- Cascading termination (some systems enforce this)
  - killing (exiting) parent → killing (exiting) all its children

Process Termination

- Cascading termination
  - If a process terminates, then all its children must also be terminated
- Wait for child process to terminate

```
pid = wait(&status);
```

- Zombie (child**早死**)
  - Child process ends before the parent calls wait()
  - Child process **資源被回收，但**PCB**仍在**

Ps. 在parent calls wait或parent結束後，pcb就會被回收

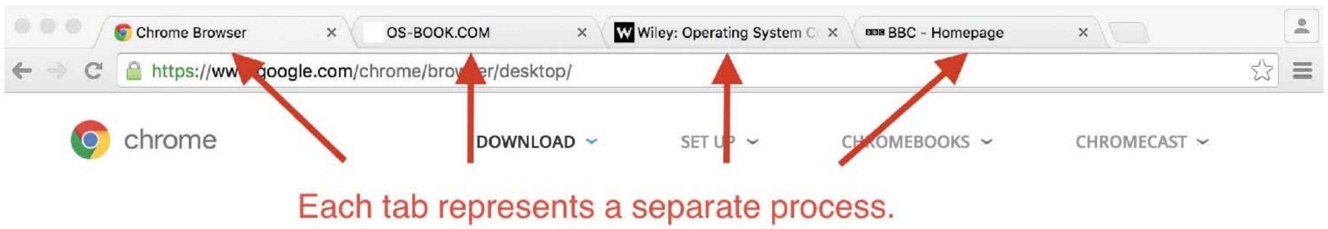
- Orphan (parent**早死**)
  - Parent terminated without calling wait(), leaving child alone
  - **換**parent: Linux systemd can be the parent of all process
    - call wait() periodically to collect the orphan child
    - Also allows other process to be the parent of the orphan child

Android Process Hierarchy

- In android, the system maintains an importance hierarchy of processes
  - Terminate based on the importance of processes
- Importance hierarchy
  - Foreground process
  - Visible process
  - Service process (apparent to the user; ex: music)
  - Background process
  - Empty process

Chrome Browser

- Browsers with single process
  - If one web site causes trouble, entire browser can hang or crash
- Chrome is multi-process; 3 different types of processes:
  - Browser process manages user interface, disk and network I/O
  - Renderer process renders web pages, deals with HTML, JavaScript. A new renderer created for each website opened
    - Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
  - Plug-in process for each type of plug-in



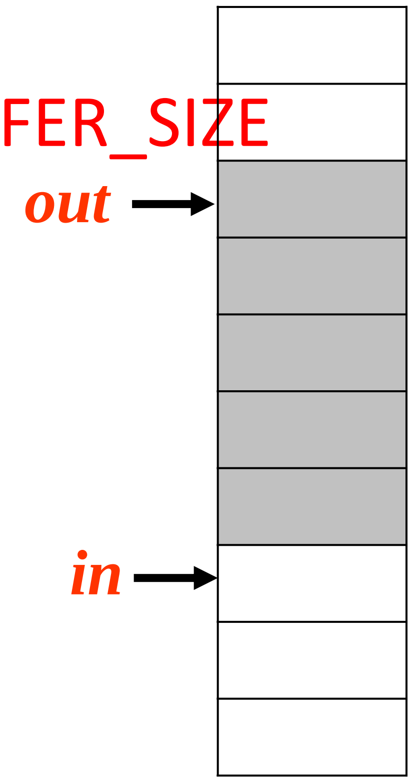
Inter-process Communication

- IPC
  - Exchanging data among multiple processes
    - Shared memory; message passing
- Cooperating process
  - Share data with other processes
- Purposes
  - information sharing
  - computation speedup (viable only in multi-core CPUs)
  - modularity

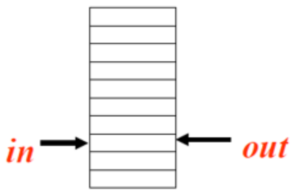
Consumer & Producer Problem

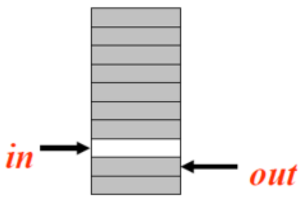
- Producer process produces information that is consumed by a consumer process
  - A buffer as the shared memory
- The buffer is a circular array with size BUFFER_SIZE
  - first free: in
  - first available: out
  - empty: in == out
  - full:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



(in+1) % BUFFER SIZE == out



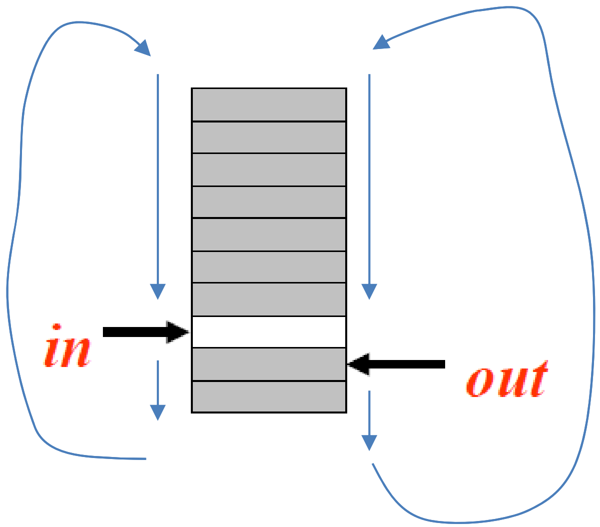


Shared-Memory Solution

```
/*producer*/
while (1) {
    while (((in + 1) % BUFFER_SIZE) == out)
          ; //wait if buffer is full
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

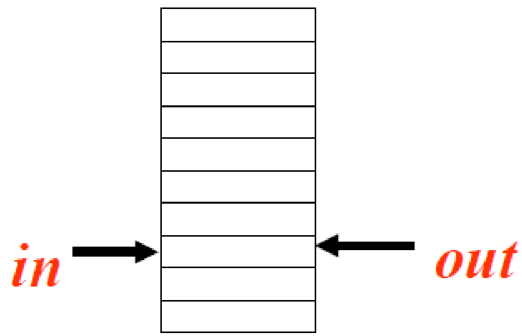
"in" only modified by producer

the buffer is full



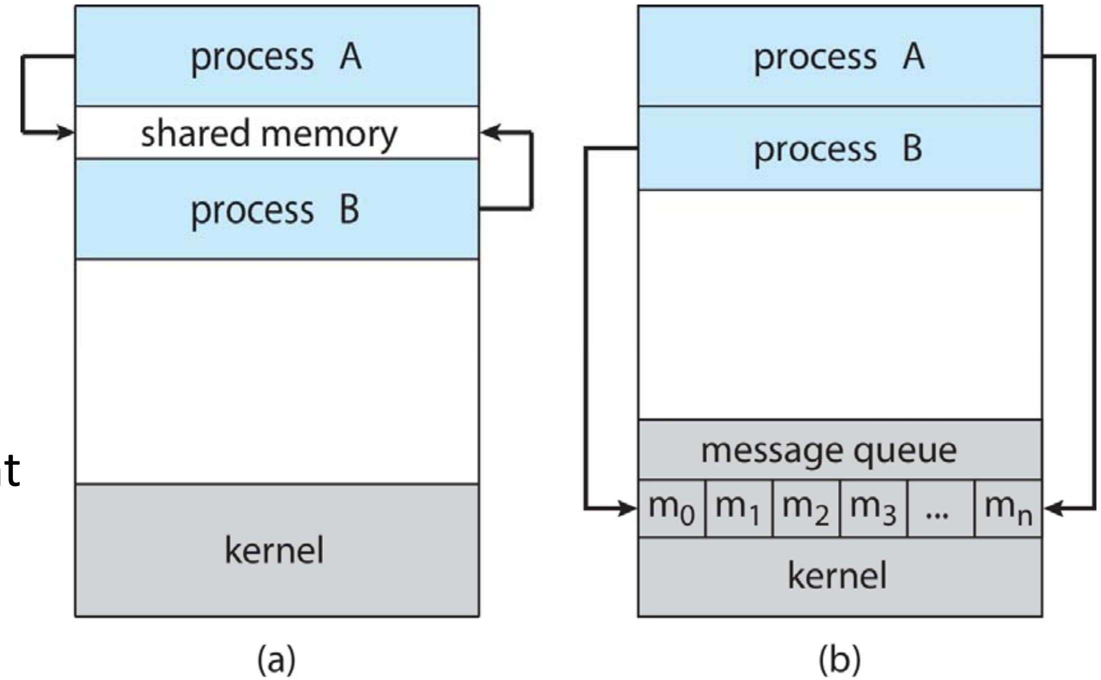

```
/*consumer*/
while (1) {
    while (in == out); //wait if buffer is empty
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

"out" only modified by consumer



Communication Methods

- Shared memory:
  - Require more careful user synchronization
  - Implemented by memory access: faster speed
  - 不需要OS支援
- Message passing:
  - No sync issues: more efficient for small data
  - Use send/recv (sys call) to transmit messages
    - slower than shared memory
    - 需要OS支援



Shared memory. Message passing

P.125, Sec. 3.5

Shared Memory

- Processes are responsible for
  - Establishing a region of shared memory
    - The hosting process creates a shared-memory region resides in the address space of the process
    - The communicating processes must agree to remove memory access constraint from OS
  - Determining the form of the data and the location
  - Ensuring data are not written simultaneously by processes
    - i.e. critical section problem
    - OS不介入，通訊二方要自己實現

Examples of IPC Systems - POSIX

See https://chmodcommand.com/

- POSIX Shared Memory
  - Process first creates shared memory segment
    `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

    int

  - Set the size of the object

    `ftruncate(shm_fd, 4096);`

  - Use **`mmap()`** to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by **`mmap()`**.

	Owner Rights (u)	Group Rights (g)	Others Rights (o)
Read (4)	☑	☑	☑
Write (2)	☑	☑	☑
Execute (1)	☐	☐	☐

https://github.com/greggagne/osc10e/blob/master/ch3/shm-posix-producer.c

IPC POSIX Producer

```c
int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

使用mmap回傳的指標來寫入共享記憶體

https://github.com/greggagne/osc10e/blob/master/ch3/shm-posix-producer.c

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

此敍述執行後，shared memory立即失效

https://github.com/greggagne/osc10e/blob/master/ch3/shm-posix-consumer.c

P.127, Sec. 3.6

Message-Passing System

- Enable processes to communicate without sharing the same address space
  - Communicating via a queue (pipe) provided by OS
- IPC facility provides two operations:
  - Send(message) – message size fixed or variable (P.128中)
  - Receive(message)
- To communicate, processes need to
  - Establish a communication link (pipe)
  - Exchange a message via send/receive

Message-Passing System

- Three issues
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

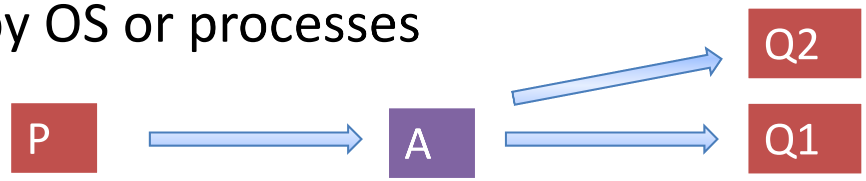
Direct communication

- Processes must name each other explicitly:
  - Send (P, message) – send a message to proc P
  - Receive (Q, message) – receive a message from process Q
- Properties of communication link
  - Links are established automatically among processes
  - One-to-One relationship between links and processes
- Drawback
  - The communicating parties are hard coded

Read: p.129 asymmetric communication

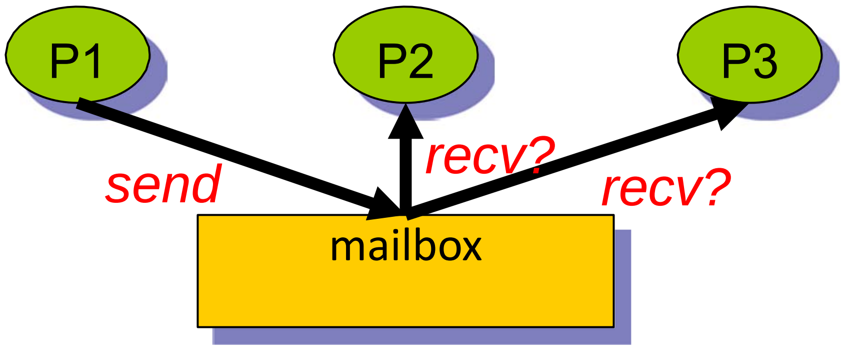
Indirect communication

- Messages are exchanged via mailboxes (ports)
  - Each mailbox has a unique ID
  - Processes can communicate if they share a mailbox
  - Send (A, message) – send a message to mailbox A
  - Receive (A, message) – receive a message from mailbox A
- Properties of communication link
  - Link established only if processes share a common mailbox
  - Many-to-Many relationship between links and processes
  - Link may be unidirectional or bi-directional
  - Mailbox can be owned either by OS or processes



Indirect communication

- Which will receive the message sent by P1?



- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select the receiver

P.130, 3.6.2

Synchronization

- Different send/recv sync combinations are possible
- Synchronous messaging passing (blocking)
  - Blocking send: sender is blocked until the message is received by receiver or by the mailbox
  - Blocking receive: receiver is blocked until the message is available
- Asynchronous messaging passing (non-blocking)
  - Non-blocking send: sender sends the message and resumes operation
  - Non-blocking receive: receiver receives a valid message or a null

P.131, 3.6.3

Queue Buffering

- Buffer
  - Queue of messages attached to the link
- Implementation alternatives
  - Zero capacity: blocking send/receive
  - Bounded capacity: if full, sender will be blocked
  - Unbounded capacity: sender never blocks

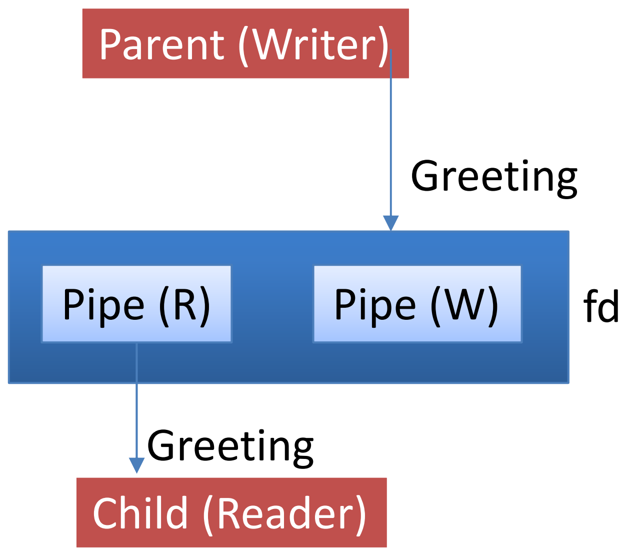
P.139, 3.7.4

Pipes

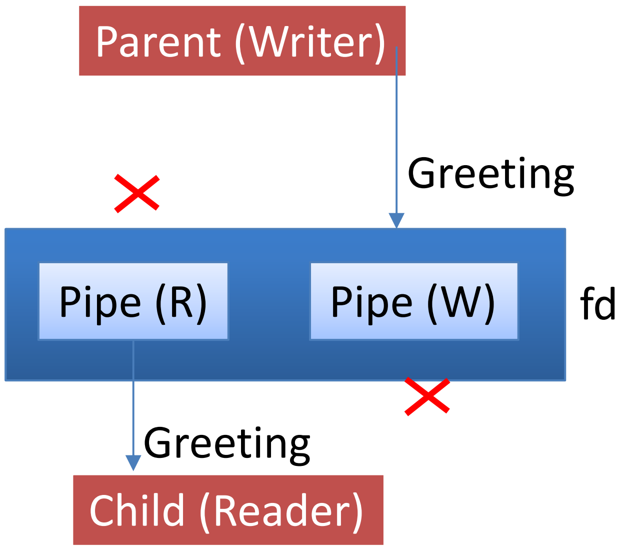
- A conduit (導管) allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes (parent-child)**
  - A parent process creates a pipe
    - Uses the pipe to communicate with a child process
- **Named pipes**
  - can be accessed without a parent-child relationship

Ordinary Pipes

- Feature
  - Ordinary pipes are unidirectional
  - Require parent-child relationship between communicating processes
- Producer writes to one end
  - the **write-end** of the pipe
- Consumer reads from the other end
  - the **read-end** of the pipe



```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Greetings";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;
```




```c
   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
   }

   /* fork a child process */
   pid = fork();
```

Child

Parent

```c
   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }

   if (pid > 0) { /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
   }
   else { /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      printf("read %s",read_msg);

      /* close the read end of the pipe */
      close(fd[READ_END]);
   }

   return 0;
}
```

Named Pipes

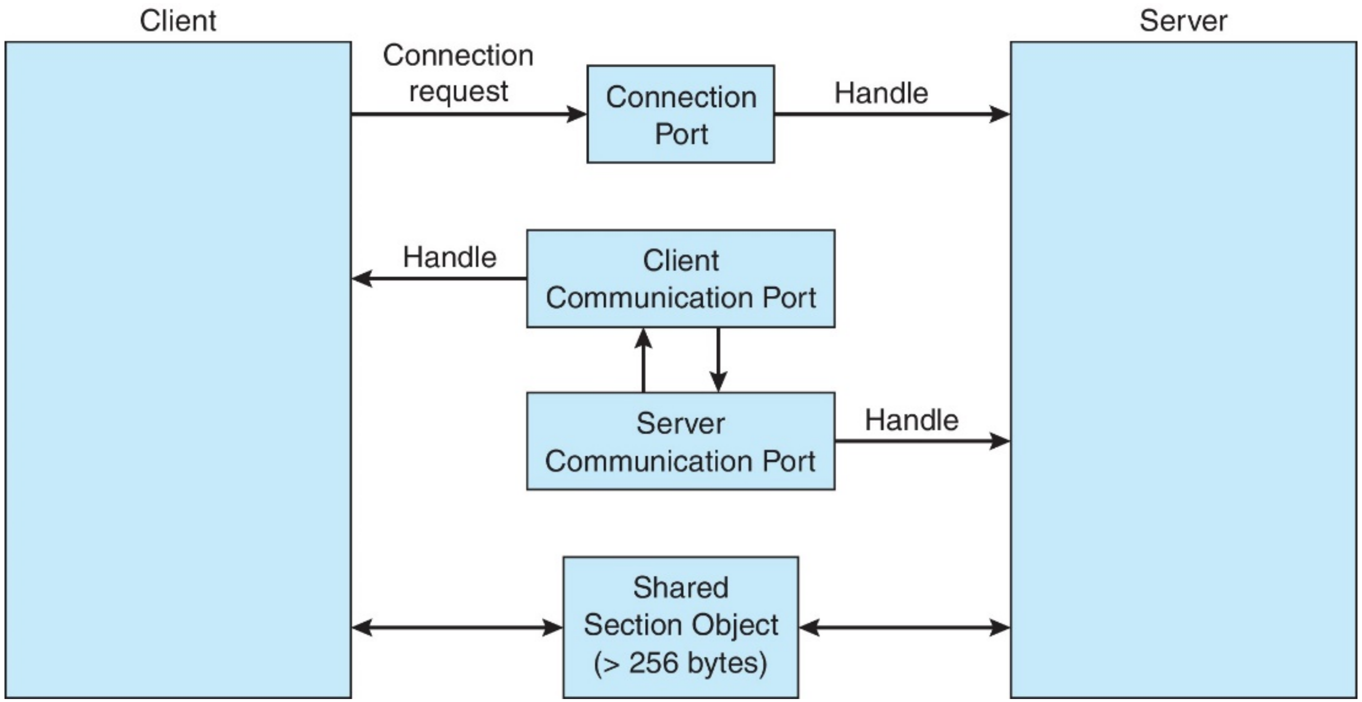
- More powerful than ordinary pipes
  - Communication is bidirectional
  - No parent-child relationship is necessary
  - Several processes can use the named pipe
- Provided on both UNIX and Windows systems
  - UNIX only supports half-duplex
  - Windows supports full-duplex

https://github.com/3p3r/node-named-pipe
mkfifo

Windows ALPC

- ALPC
  - Advanced Local Procedure Call
- Steps
  - Server publishes a connection port publicly
  - Client sends a request to the connection port
    - Server setup a private channel (contains 2 communication ports)
    - Client obtain the handle of the communication port from server
  - Messaging
    - Small messages: (<= 256 bytes) directly via communication ports
    - Big messages: section object (shared memory)
    - Huge messages: direct memory access via specific API

Windows ALPC



Socket

- 網路通訊程式
  - 位於二台不同電腦的程式互相傳送訊息
- Internet上每台電腦至少有一個固定位址
  - IP + Port
  - 例如
    - 140.119.168.10: 80

http://www.nccu.edu.tw

0-255

0-65534

每個IP都有65535個port(埠)

網路名稱轉譯
www.nccu.edu.tw →140.119.168.10
http:// → 80

IP網路通訊程式基礎

- 不同Port代表不同服務
- Well-known port
  - 80: HTTP
  - 443: HTTPS
  - 21: FTP
  - 22: SSH
  - 23: Telnet

IP網路通訊程式基礎

- 特殊位址
  - 127.0.0.1 →localhost
    - 指向自己
    - 同一台電腦不同port間也可以通訊
  - 192.168.
    - 虛擬位址 (只在LAN中有效)
  - 239.
    - 群播 (multicast)

127.0.0.1: 12345 →
127.0.0.1: 8080

Socket Server

```
let net = require('net');

let server = net.createServer(function(s) {
        s.setEncoding(‘utf8’);
        s.on('data', function(data) {
                console.log(s.remoteAddress);
                console.log(s.remotePort);
                console.log(data);
        });
});

server.listen(1337);
```

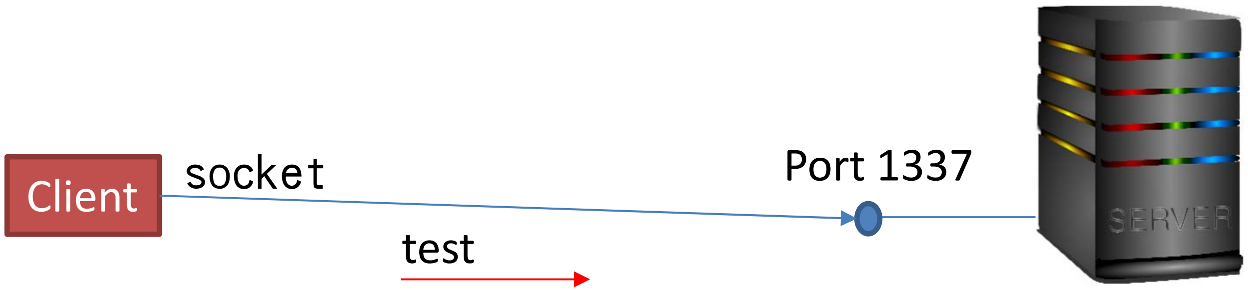
設定編碼為字串

Port 1337

若沒有設定編碼為字串，則傳入的data會是Buffer物件
若是如此，則要使用data.toString()，資料才能正常顯示。

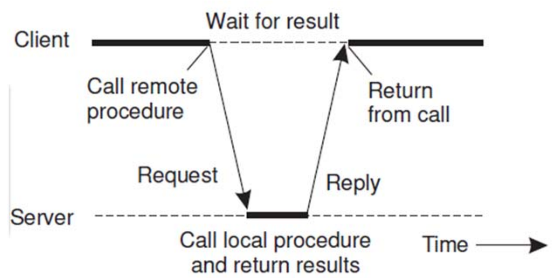
Socket Client

```
let net = require('net');

let client = new net.Socket();

client.connect(1337, '127.0.0.1', function () {
        console.log('connected');
        client.write('test', () => console.log('written'));
    }
);
```



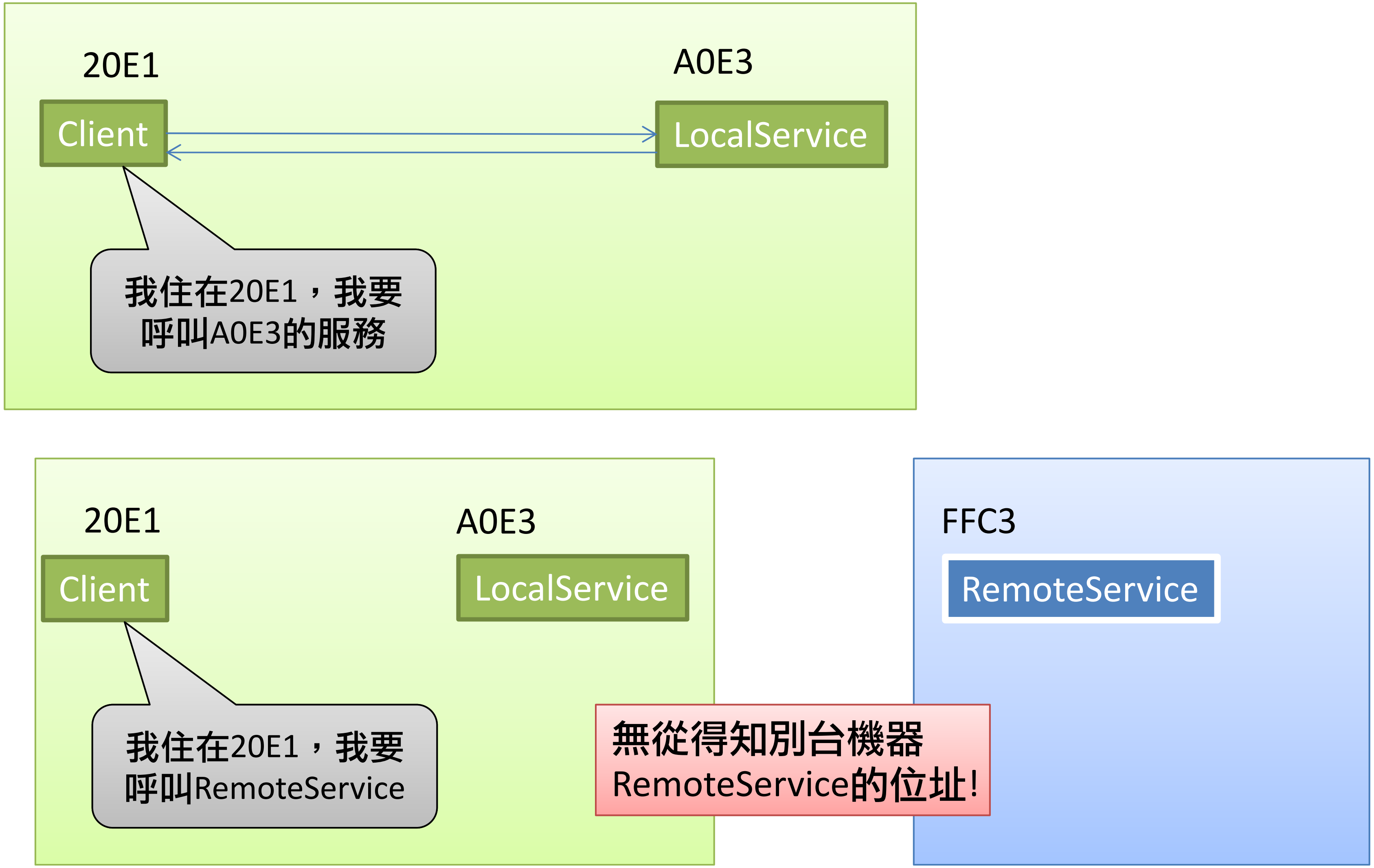
Remote Procedure Call (RPC)

- Motivation
  - A type of the direct communication mechanism
  - Calling procedure on a remote host “as if” calling a local procedure (Saif and Greaves, 2001)
  - Warning: unaware of “remoting” is considered harmful
- Approach
  - Communication between caller & callee can be hidden by using a proxied procedure-call mechanism

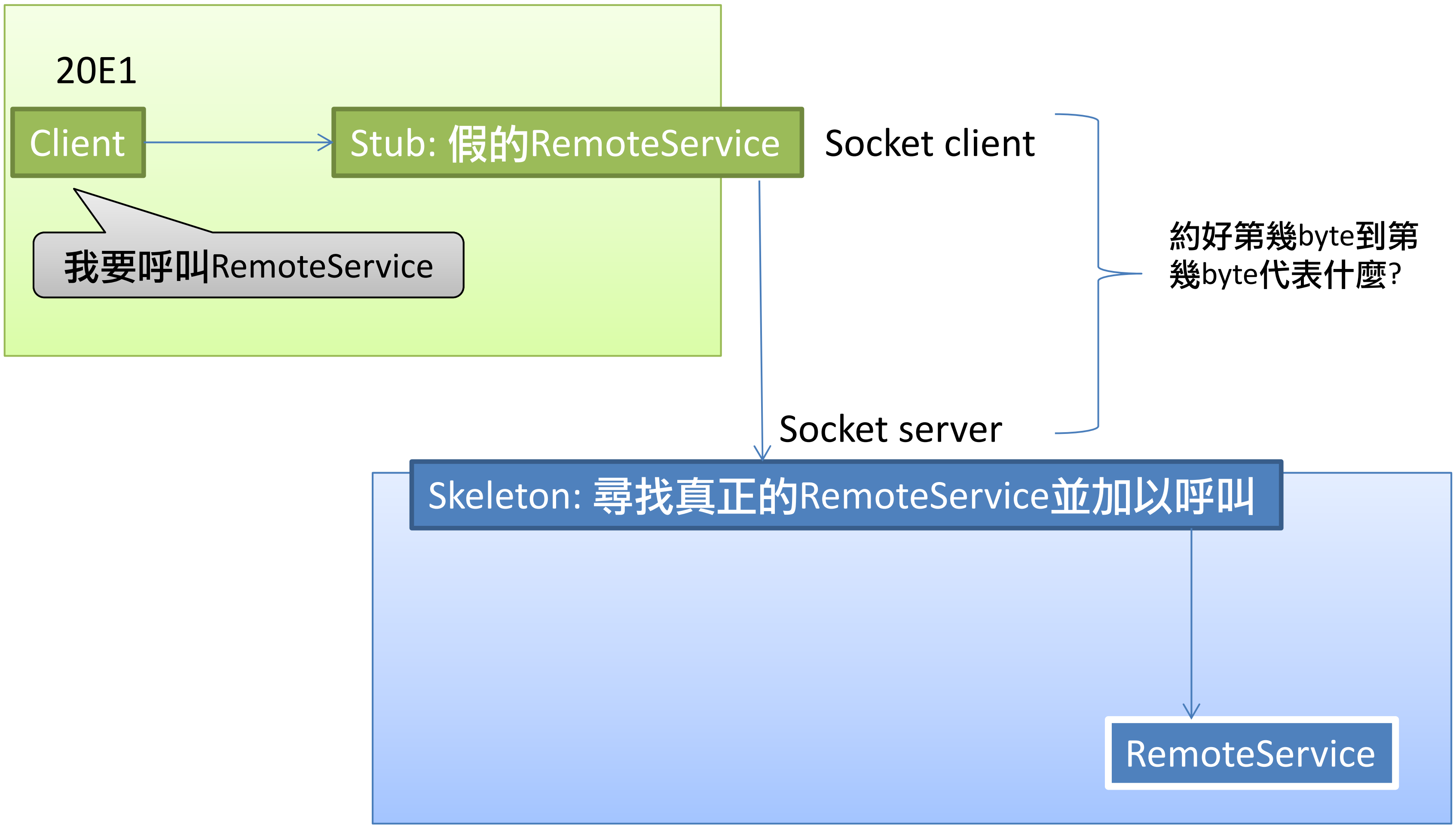


U. Saif and D. J. Greaves, "Communication primitives for ubiquitous systems or RPC considered harmful," Proceedings 21st International Conference on Distributed Computing Systems Workshops, Mesa, AZ, USA, 2001, pp. 240-245, doi: 10.1109/CDCS.2001.918712.
https://ieeexplore.ieee.org/abstract/document/918712

Remote Procedure Calls



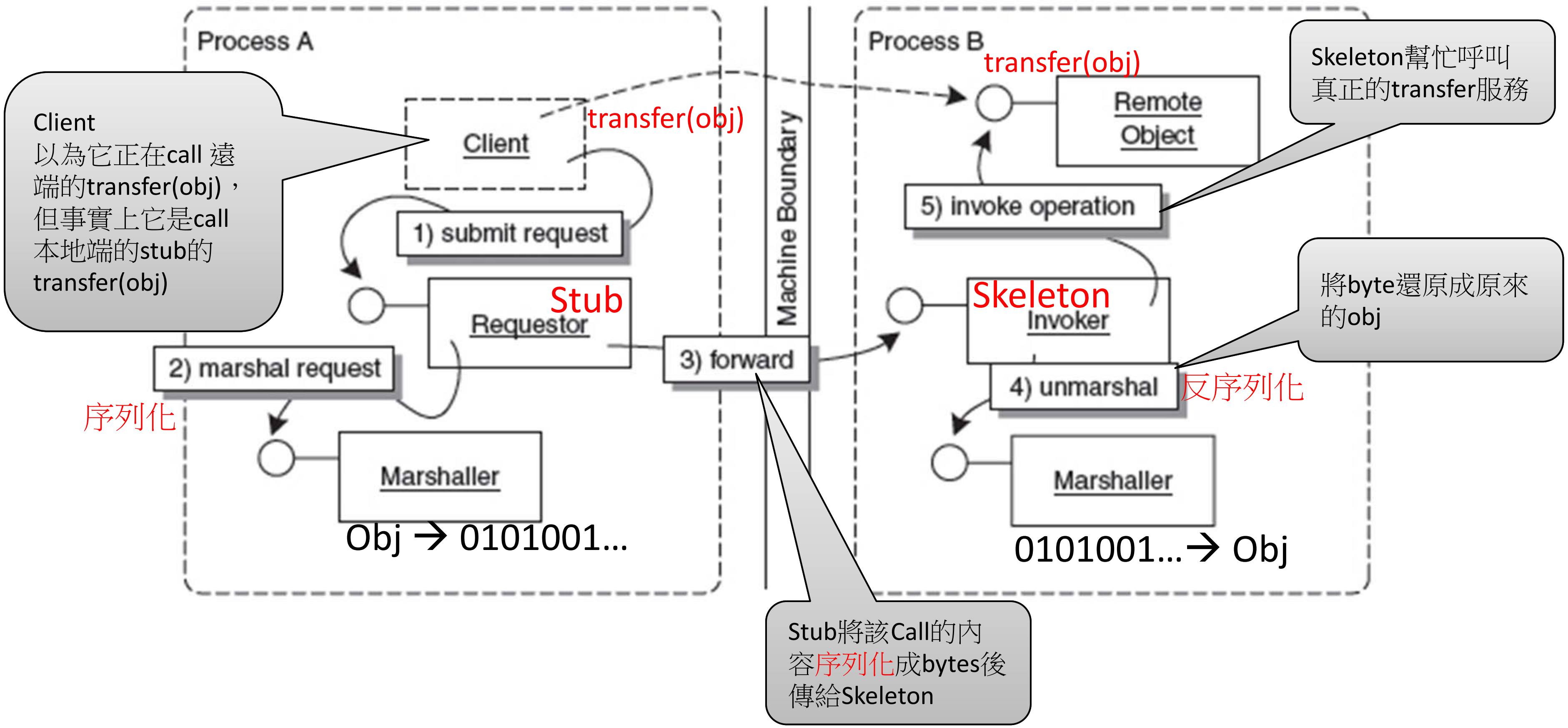
Remote Procedure Calls



Serialization (Marshalling) 序列化

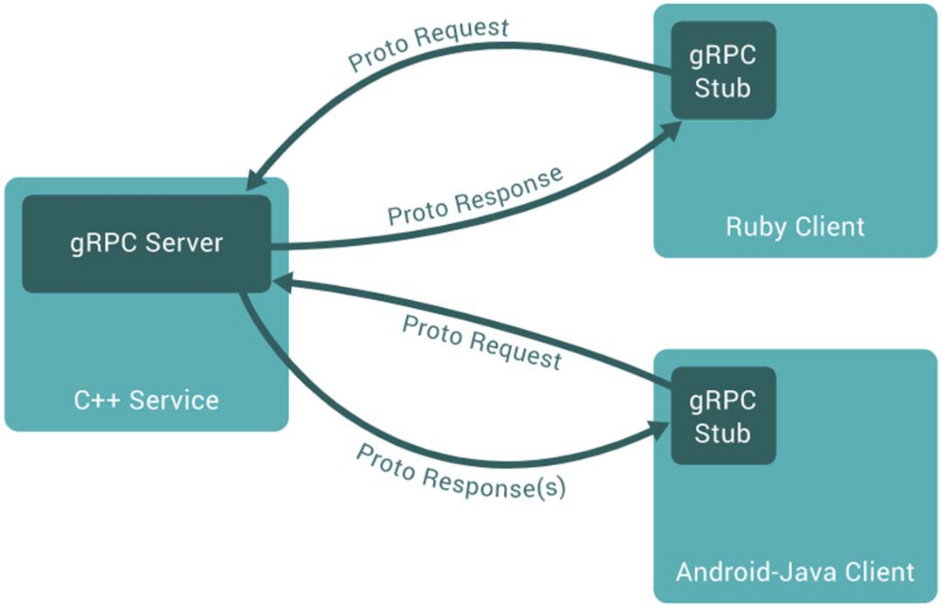
- 參數如何傳送?
  - Client and server machines may have different data representations
    - E.g., byte ordering
  - Serialization序列化: transforming a value into a sequence of bytes
    - Client and server have to agree on the <u>same encoding standard</u>
  - 議題
    - How are basic data values represented (integers, floats, characters)
    - How are complex data values represented (arrays, unions)

呼叫遠端函式(物件)的架構



Case: gRPC

- A modern open source high performance RPC framework that can run in any environment
  - Efficiently connect services in and across data centers
  - Technology stack
    - HTTP 2
    - Protocol Buffers



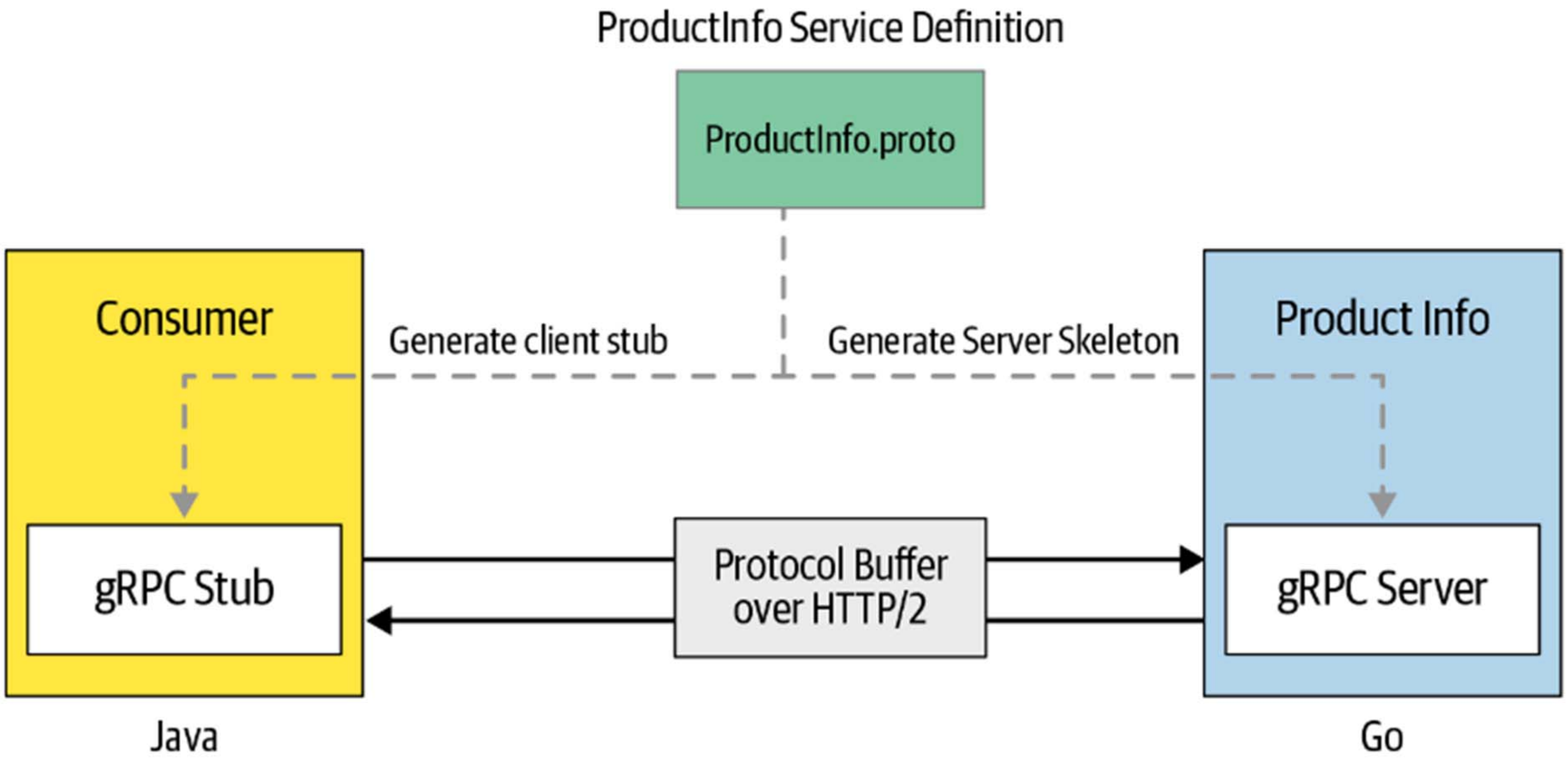
gRPC in the Realworld

- gRPC has been widely adopted for building microservices and cloud native applications
- Netflix
  - Initially using in-house RESTful solution on HTTP/1.1
  - With the adoption of gRPC, Netflix has seen a massive boost in developer productivity
    - Creating a client, which could take up to two to three weeks, takes a matter of minutes with gRPC.

gRPC in the Realworld

- Dropbox
  - Dropbox runs hundreds of polyglot microservices, which exchange millions of requests per second
  - Initial solution
    - A homegrown RPC framework with a custom protocol for manual serialization
    - Apache Thrift
    - Legacy HTTP/1.1-based RPC framework + protobuf
  - New solution: Courier
    - A gRPC-based RPC framework
    - A customized solution to meet specific requirements like authentication, authorization, service discovery, service statistics, event logging, and tracing tools

A Server and a Client based on gRPC



Service Definition

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

```
function sayHello(call, callback) {
  callback(null, {message: 'Hello ' + call.request.name});
}

function main() {
  var server = new grpc.Server();
  server.addService(hello_proto.Greeter.service,
                    {sayHello: sayHello});
  server.bind('0.0.0.0:50051', grpc.ServerCredentials.createInsecure());
  server.start();
}
```

```
function main() {

  var client = new hello_proto.Greeter('localhost:50051',
                                       grpc.credentials.createInsecure());
  client.sayHello({name: 'you'}, function(err, response) {
    console.log('Greeting:', response.message);
  });

}
```

課後閱讀 (會考)

- P115 手持裝置的Multitasking
- P135 Mach Messaging Passing: Task Self port的用途
- P136 什麼是Bootstrap server
- P136 Mach Message Passing如何傳送大檔?
- P138 ALPC是什麼? 作用為何?
- P139 ALPC為什麼需要section object?
- P151 Android中提供什麼支援來開發RPC?

Q & A