Chapter 3 行程觀念 Process concept

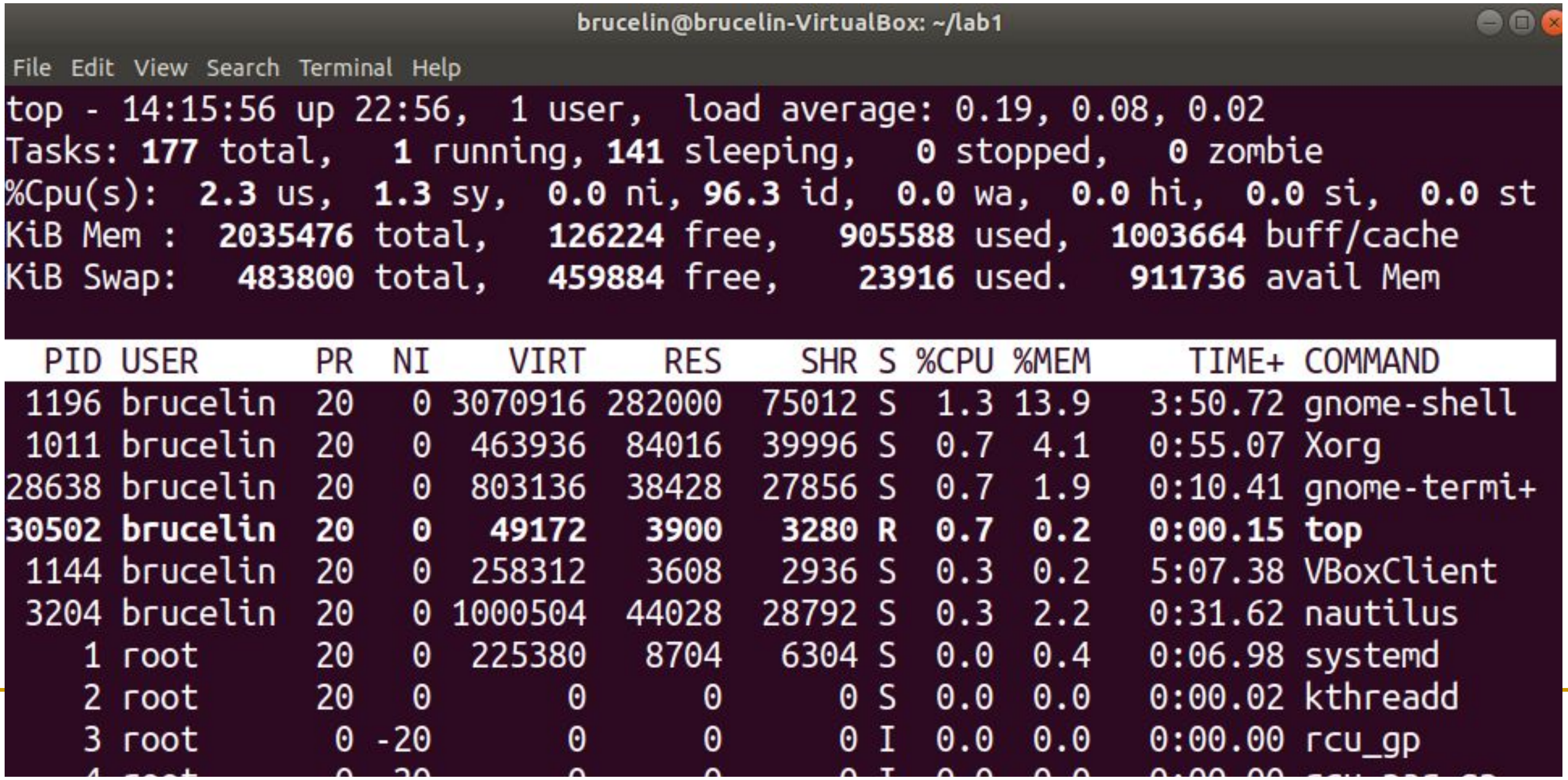
林政宏
國立臺灣師範大學電機工程學系

Chapter 3 行程觀念

- 3.1 行程概念
- 3.2 行程排班
- 3.3 行程的操作
- 3.4 行程間通訊
- 3.5 IPC系統的範例
- 3.6 客戶/伺服器的通信

Linux processes

- Process is a unit of running program
- Each process has some information, like process ID, owner, priority, etc
- Example: Output of "top" command


brucelin@brucelin-VirtualBox: ~/lab1
File Edit View Search Terminal Help

```
top - 14:15:56 up 22:56,  1 user,  load average: 0.19, 0.08, 0.02
Tasks: 177 total,   1 running, 141 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.3 us,  1.3 sy,  0.0 ni, 96.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2035476 total,   126224 free,   905588 used,  1003664 buff/cache
KiB Swap:   483800 total,   459884 free,    23916 used.   911736 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1196 brucelin  20   0 3070916 282000  75012 S  1.3 13.9   3:50.72 gnome-shell
 1011 brucelin  20   0  463936  84016  39996 S  0.7  4.1   0:55.07 Xorg
28638 brucelin  20   0  803136  38428  27856 S  0.7  1.9   0:10.41 gnome-termi+
30502 brucelin  20   0   49172   3900   3280 R  0.7  0.2   0:00.15 top
 1144 brucelin  20   0  258312   3608   2936 S  0.3  0.2   5:07.38 VBoxClient
 3204 brucelin  20   0 1000504  44028  28792 S  0.3  2.2   0:31.62 nautilus
    1 root      20   0  225380   8704   6304 S  0.0  0.4   0:06.98 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.02 kthreadd
    3 root       0 -20       0      0      0 I  0.0  0.0   0:00.00 rcu_gp
```

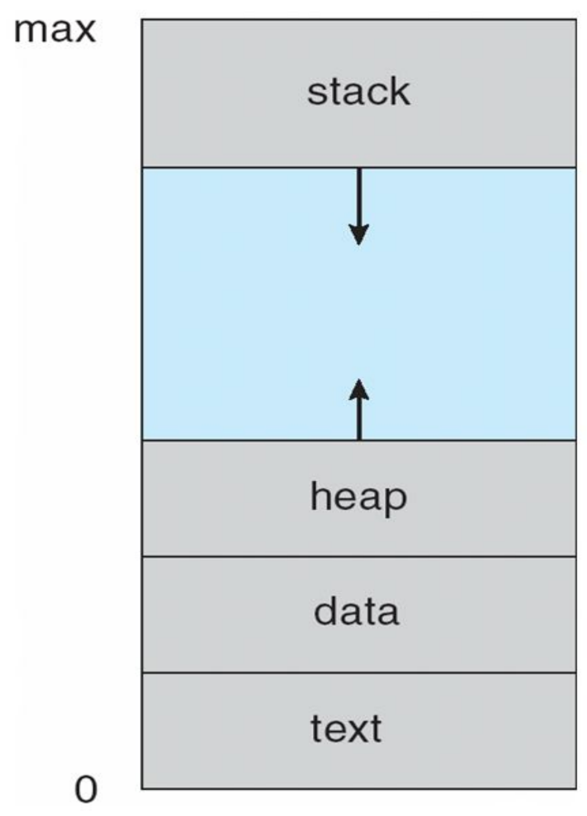
3.1 行程(Process)概念

- 討論作業系統的一個問題就是該如何稱呼CPU所有的運作項目。整批式系統(batch system)執行工作 (jobs)，而分時(time-shared)系統有使用者程式(user programs)，(或稱為任務，task)。
- 在單一使用者系統 (如Microsoft Windows)使用者仍可同時執行數個程式:一個文書處理程式、網頁瀏覽器、email套件程式。
- 就算使用者只能一次執行一個程式，作業系統仍須支援其內部一些工作，比方說是記憶體管理。從各方面看來，這些所有的事情都類似，所以稱之為行程。

3.1.1 行程(Process)

- 行程指的是正在執行的程式。行程不只是程式碼 (有時也稱為本文區, text section)而已。它還包含代表目前運作的程式計數器 (Program counter)數值和處理器的暫存器內容。

- 行程還包括存放暫用資料 (譬如:**副程式的參數、返回位址**, 及暫時性變數)的**行程堆疊 (stack)**, 以及包含整體變數的**資料區間 (data section)**。行程也包含**堆積 (heap)**, 堆積就是在行程執行期間動態配置的記憶體, 行程記憶體結構如圖。



Text Section (Code Section or Text Segment)

- The text section contains the **executable code** of the program.
- It stores the **machine instructions**, typically in a format that the processor can directly execute.
- These instructions are the actual program code that specifies the sequence of operations to be performed.
- The text section is usually **read-only** and is shared among multiple instances of the same program that are running simultaneously.
- This section is mapped into memory from the program's executable file.

Data Section (also known as Data Segment)

- The data section contains initialized **global** and **static variables**, as well as **constants**.
- This section typically includes variables declared at the **global scope** (outside of any function) and **static** variables declared within functions.
- Initialized data, such as arrays or global variables with assigned values, is stored here.
- The data section is writable, allowing the program to modify the values of variables during execution.
- Additionally, uninitialized global and static variables (often referred to as "bss" or "common" section) are allocated within this segment. These variables are initialized to zero by the system before program execution.
- Constants defined in the source code, such as const variables, may also be stored in the data section.

Stack Memory

- Stack memory is a region of memory allocated for function calls and local variables within those functions.
- It operates in a last-in, first-out (LIFO) manner, meaning that the most recently allocated memory is the first to be deallocated.
- Memory allocation and deallocation in the stack are handled automatically by the compiler or runtime system, typically through stack pointers.
- Stack memory is limited in size and fixed at compile time. It's usually smaller than heap memory.
- Because of its limited size and automatic management, stack memory is generally faster and more efficient than heap memory.
- Stack memory is generally used for storing local variables, function parameters, return addresses, and other function-related data.

Heap memory

- Heap memory is a **dynamic memory allocation** area that is used for storing data that needs to persist beyond the scope of a single function call.
- Memory allocation and deallocation in the heap are managed explicitly by the programmer, using functions like malloc() and free() in languages like C, or new and delete in languages like C++.
- Heap memory is larger and less constrained than stack memory. It can grow and shrink dynamically during program execution.
- Because heap memory management involves manual allocation and deallocation, there's a risk of memory leaks (unreleased memory) and fragmentation (unused memory blocks scattered throughout the heap).
- Heap memory is commonly used for dynamic data structures such as linked lists, trees, and dynamic arrays, as well as for objects created with dynamic memory allocation in object-oriented programming languages.

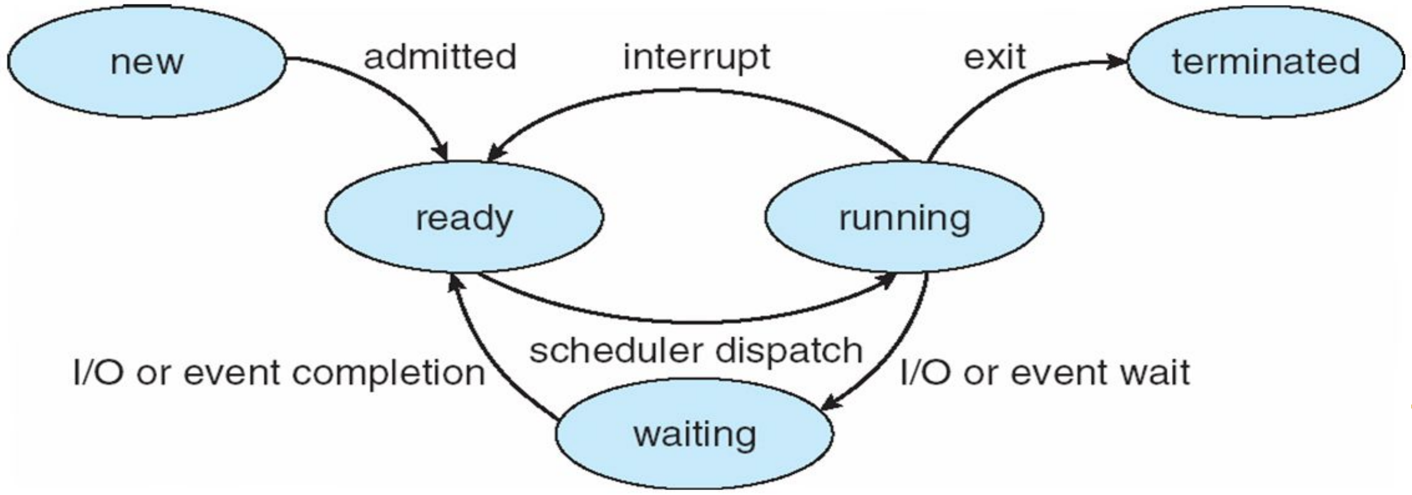
```
#include <stdio.h>

// Global variables initialized in the data section
int global_var1 = 10;
int global_var2 = 20;

// Static variables initialized in the data section
static int static_var1 = 30;
static int static_var2 = 40;

int main() {
    // Local variables stored on the stack, not in the data section
    int local_var1 = 50;
    int local_var2 = 60;

    printf("Global variables: %d, %d\n", global_var1, global_var2);
    printf("Static variables: %d, %d\n", static_var1, static_var2);
    printf("Local variables: %d, %d\n", local_var1, local_var2);

    return 0;
}
```

```
#include <stdio.h>

void function() {
    // Static local variable retains its value between function calls
    static int count = 0;

    // Increment the static local variable
    count++;

    // Print the value of the static local variable
    printf("Static local variable count: %d\n", count);
}

int main() {
    // Call the function multiple times
    function(); // Output: Static local variable count: 1
    function(); // Output: Static local variable count: 2
    function(); // Output: Static local variable count: 3

    return 0;
}
```

- Since count is declared as a **static local variable** within the function() function, it is stored in the **data segment**.
- Specifically, static local variables are stored in the initialized data segment because they have an initial value that persists across function calls.
- The static keyword affects the storage duration of the variable, ensuring that its value persists across function calls, unlike non-static local variables.

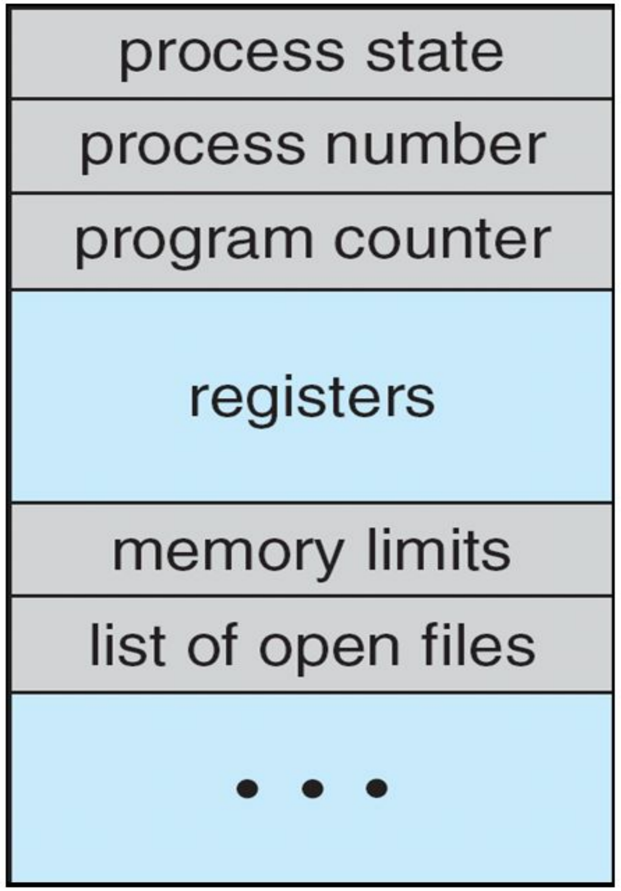
3.1.2 行程狀態

- 行程在執行時會改變其狀態。行程的狀態 (state)部份是指該行程目前的動作, 每一個行程可能會處於以下數種狀態之一:
  - **新產生 (new):**該行程正在產生中。
  - **執行 (running):**該行程正在執行。
  - **等待(waiting):**等待某件事件的發生(譬如輸入出完成或接收到一個信號)。
  - **就緒 (ready):**該行程正等待指定一個處理器。
  - **結束 (terminated):**該行程完成執行。



3.1.3 行程控制表

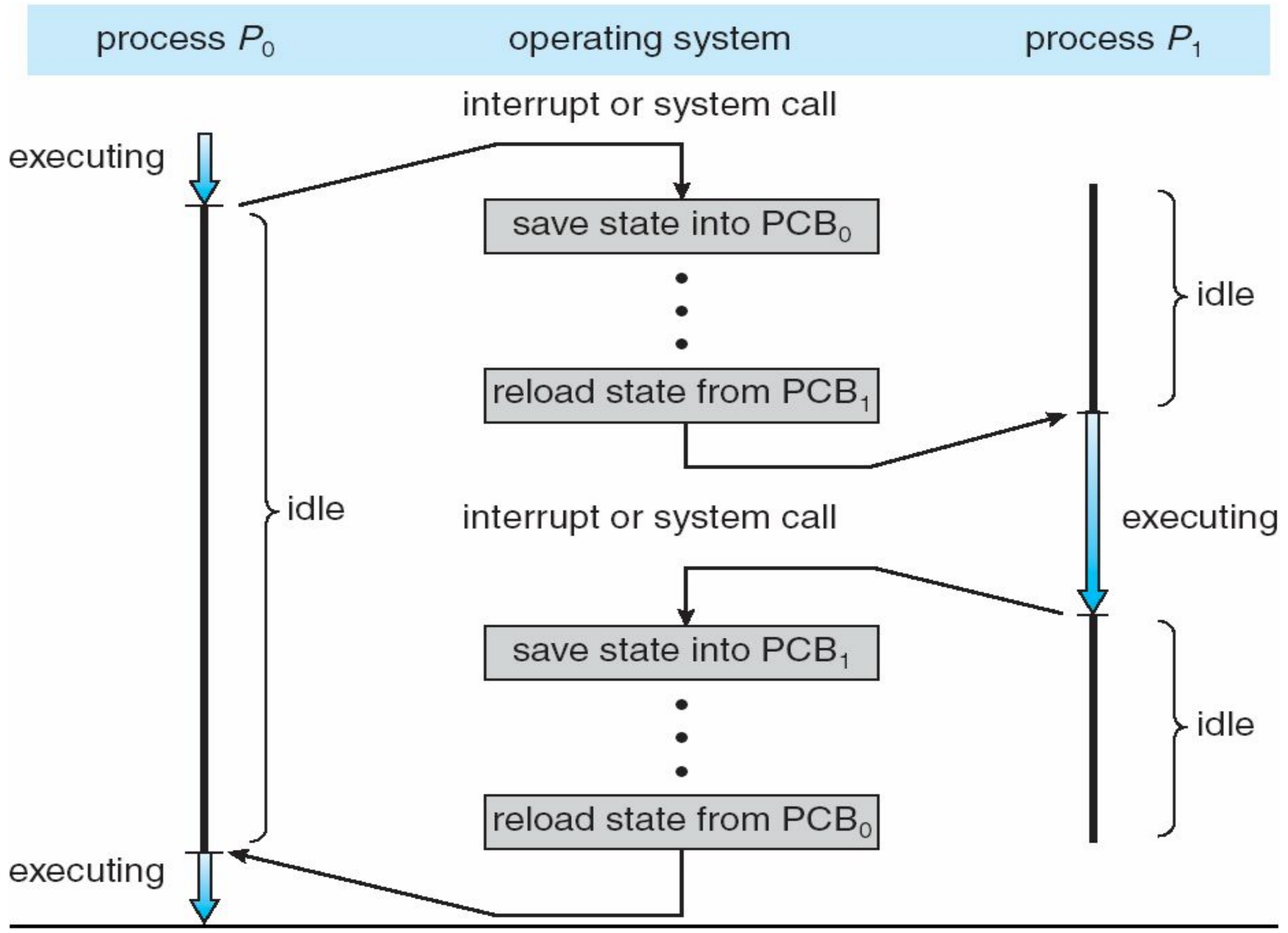
- 每一個行程在作業系統之中都對應著一個**行程控制表 (Process control block (PCB)**或稱**任務控制表 (task control block)**, 如圖。
- 行程控制表 (PCB)記載所代表的行程之相關資訊, 包括:
  - **Process Identifier (PID)**
  - **Process State**:可以是new、ready、running、waiting或halted等。
  - **Program Counter (PC)**:指明該行程接著要執行的指令位址。
  - **CPU Registers**:其數量和類別, 完全因電腦架構而異。包括累加器 (accumulator)、索引暫存器 (index register)、堆疊指標 (stack pointer)以及一般用途暫存器 (general-purpose register)等, 還有一些狀況代碼 (condition code)。當中斷發生時, 這些狀態資訊以及程式執行計數器必須儲存起來, 以便稍後利用這些儲存的資訊, 使程式能於中斷之後順利地繼續執行 。



3.1.3 行程控制表(cnt.)

- **Process Priority**:包括行程的優先順序 (Priority)、排班佇列(scheduling queue)的指標以及其它的排班參數。
- **Memory Management Information**:這些資訊包括如基底暫存器(base register)和限制暫存器(limit register), 分頁表(Page table)值的資訊統所使用的記憶系統區段表(segment table)。
- **Pointer to the Parent Process:** Points to the PCB of the parent process if applicable.
- **Accounting Information**帳號資訊:包括了CPU和實際時間的使用數量、時限、帳號工作或行程號碼。
- **Pointers to Open Files:** Contains pointers to files or I/O devices opened by the process.。
- **Interprocess Communication Information:** Contains details about communication channels or message queues used by the process for interprocess communication.

CPU switch from process to process

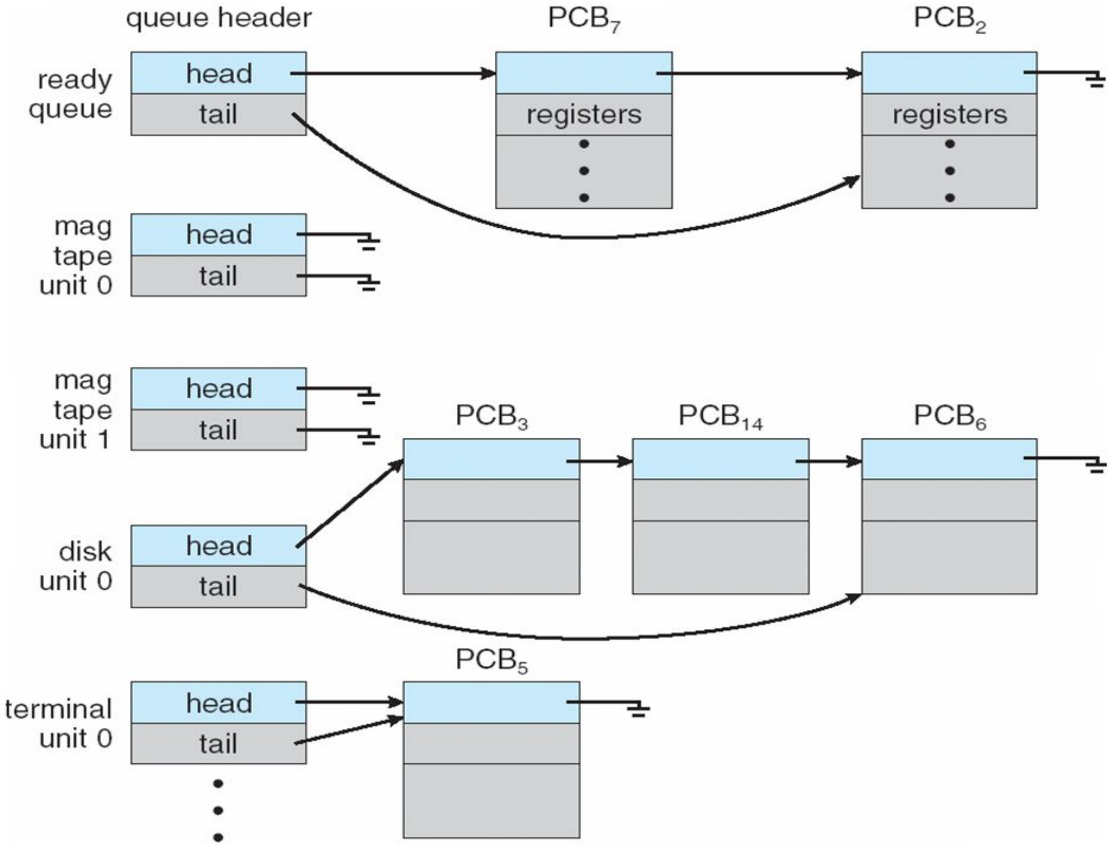


3.2 行程排班(Process Scheduling)

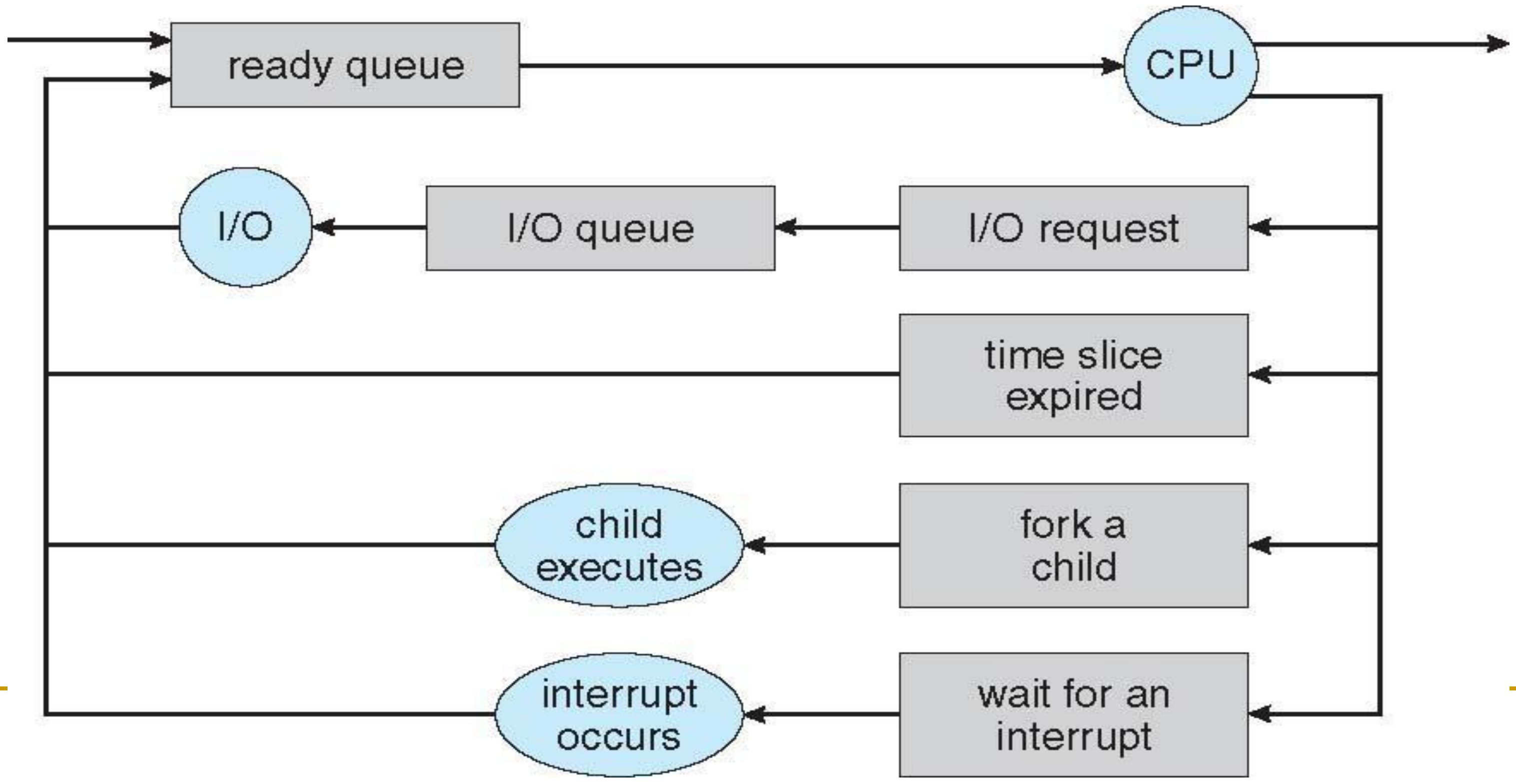
- 多元程式規劃(multiprogramming)系統的主要目的，是隨時有一個行程在執行，藉以提高CPU的使用率。分時(time sharing)系統的目的是將CPU在不同行程之間不斷地轉換，以便讓使用者可以在自己的行程執行時**與它交談**。
- 為了達到這個目的，行程排班程式(process scheduler)為CPU選擇一個可用的行程(可能由一組可用行程)。
- 單一處理器系統，不可能有一個以上的行程同時執行。如果有多個行程，其它的都必須在旁邊等待一直到CPU有空，才可能重新排列。

3.2.1 排班佇列(scheduling queue)

- 行程進入系統時，是放在工作佇列(job queue)之中，此佇列是由所有系統中的行程所組成。位於主記憶體中且就緒等待執行的行程是保存在一個所謂就緒佇列 (ready queue)的串列。這個佇列一般都是用鏈接串列的方式儲存。在ready queue前端保存著指向這個串列的第一個和最後一個PCB的指標。



- 一個新的行程最初是置於ready queue中。就一直在ready queue中等待，直到選來執行或被分派 (dispatched)。一旦這個行程配置CPU並且進行執行，則會有若干事件之一可能發生:
  - 行程可發出I/0要求，然後置於一個I/0佇列中。
  - 行程可產生出一個新的子行程並等待後者的結束。
  - 行程可強行地移離CPU(如用中斷的結果一樣)，然後放回ready queue中。



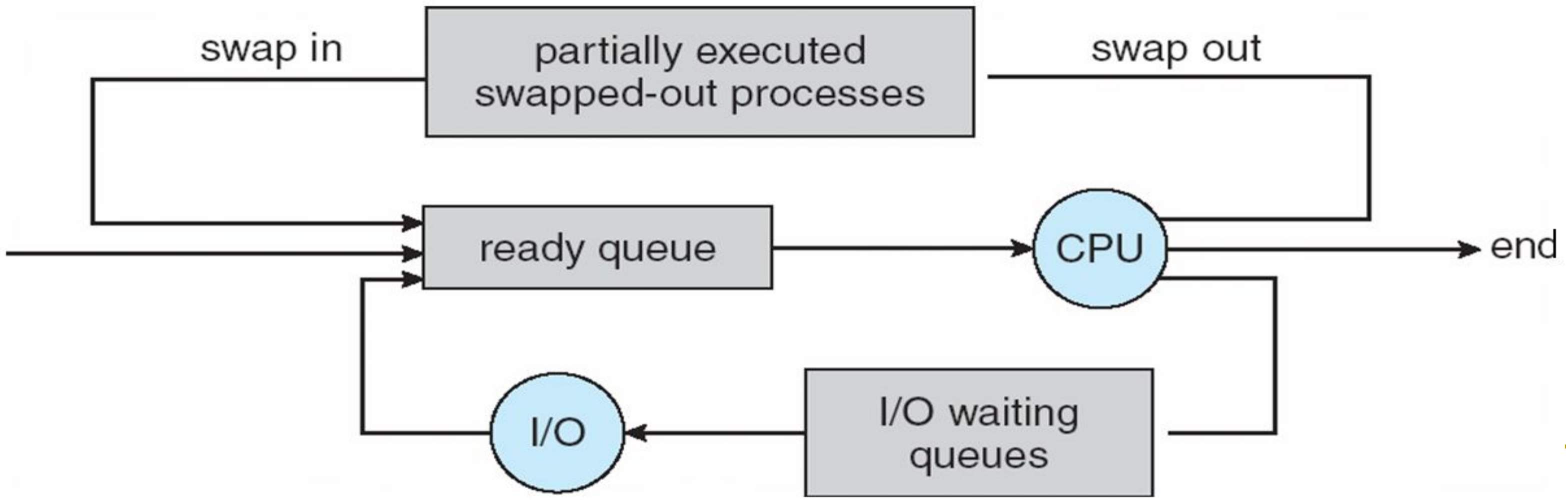
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
- The long-term scheduler controls the **degree of multiprogramming**

- 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; few very long CPU bursts

3.2.2 排班程式

- 一個行程在它整個生命期裏將在各個不同的排班佇列間遷移。作業系統必須按排班次序從這些佇列選取行程。行程的選取將由適當的排班程式 (scheduler)來執行。
- 分時系統，可能會採用一種額外的、間接方式來排班。
- 中程排班程式 (medium-term scheduler)背後的最主要觀念就是有時後可以將行程從記憶體中有效地移開(並且從對CPU的競爭中移開)、並減低多元程式規劃的程度。
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



3.2.3 內容轉換Context Switch

- 中斷使作業系統改變CPU目前的工作而執行核心常式，這樣的作業常發生在一般用途系統上。當中斷發生時，系統需要儲存目前在CPU上執行行程的內容 (context)，所以當作業完成時，它可以還原內容，本質就是暫停行程，再取回行程。

- 轉換 CPU至另一項行程時必須將舊行程的狀態儲存起來，然後再載入新行程的儲存狀態。這項任務稱為內容轉換(context switch)。

- 內容轉換是額外負擔(overhead); 系統在做內容轉換時，沒辦法作任何有用的工作
  - 作業系統與PCB越複雜，內容轉換所需的時間越長

- 內容轉換所需的時間取決於硬體支援
  - 有些硬體提供多組暫存器可以允許多個內容轉換同時執行

3.3 行程的操作

- 系統中的各個行程可以並行 (concurrently) 地執行，而且也要能動態地產生或刪除。作業系統必須提供行程產生和結束的功能。

3.3.1 行程的產生

- 一個行程的執行期間，可以利用產生行程的系統呼叫來產生數個新的行程。原先的行程就叫做父行程 (Parent process)，而新的行程則叫做子行程(children process)。
- 每一個新產生的行程可以再產生其它的行程，這可以形成一幅行程樹 (tree of processes)。
- 每個行程有自己的 process identifier (pid)

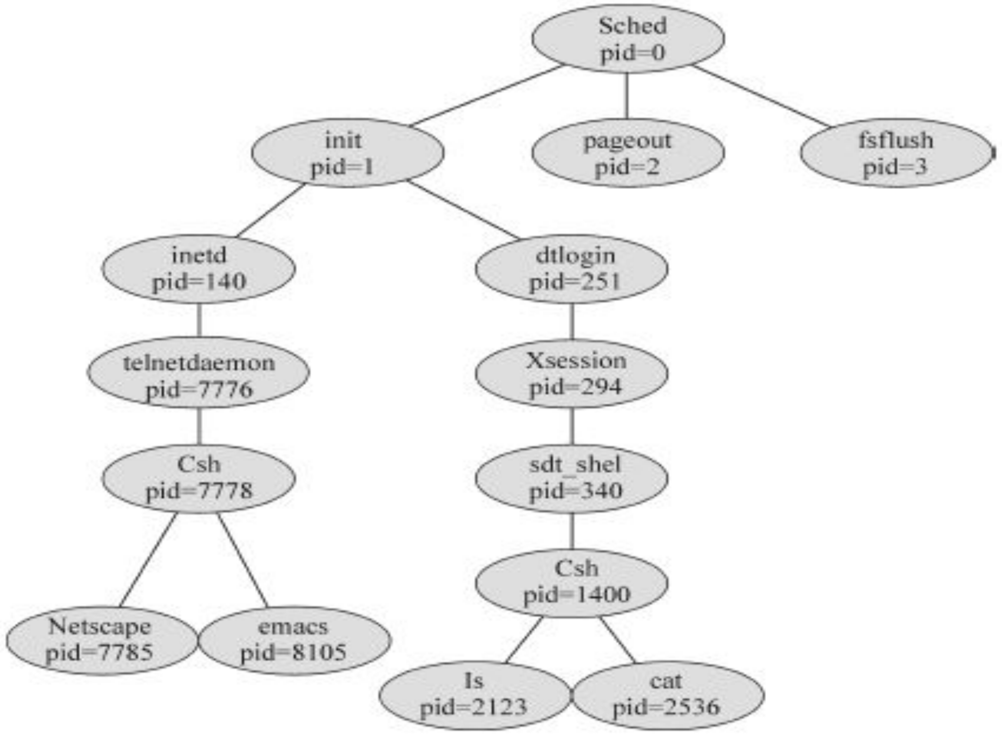


圖 3.9 典型的 Solaris 系統的行程樹

Process Creation

- 當一個父行程產生一個子行程, 有兩種執行的可能性
  - 父行程與子行程同時執行
  - 父行程等待直到子行程結束

- 對於資源分享, 有三種可能性
  - 父行程與子行程共享所有資源
  - 子行程分享父行程的部分資源
  - 父行程與子行程不共享資源

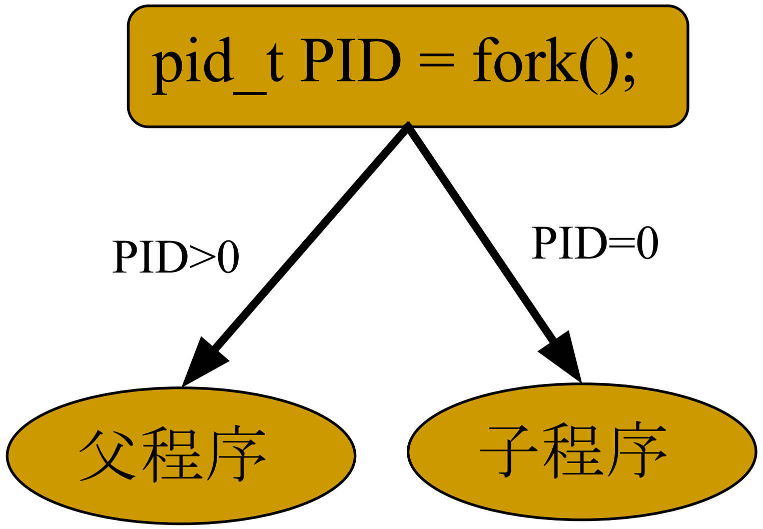
Process Creation (Cont)

- 記憶體空間(address-space)
  - 子行程複製父行程的記憶體空間
  - 子行程負載新的程式

- UNIX examples
  - **fork** system call creates new process
  - A new process is created by the fork() system call.
  - The new process consists of a copy of the address space of the original process.
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

fork 的函數雛型 (man page 定義)

- #include <unistd.h>
- pid_t fork(void);
- fork() 可能會有以下三種回傳值 :
  - -1 : 發生錯誤
  - 0 : 代表為子程序
  - 大於 0 : 代表為父程序, 其回傳值為子程序的 Process ID
  - 注意 : 其回傳值是 pid_t , 不是 int 哦 !



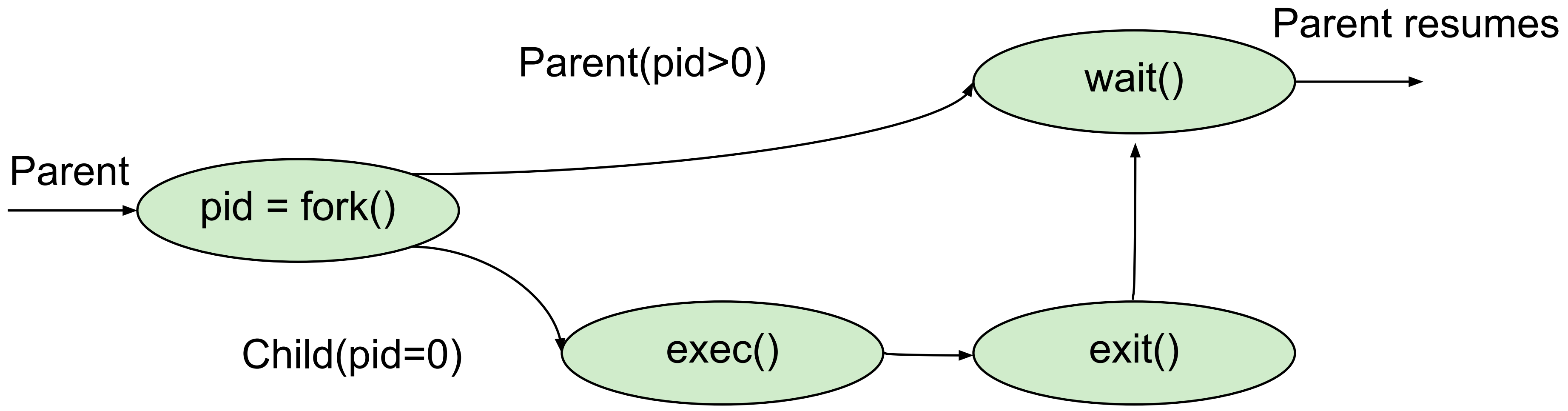
```
#include <sys/wait.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
      pid_t  pid;
      /* fork another process */
      pid = fork();
      if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
      }
      else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
      }
      else { /* parent process */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
      }
}
```

Child process

Parent will wait for the child to complete

3.3.2 行程的結束(Process termination)

- 一個行程在執行完最後一個敘述，以及使用**系統呼叫 exit()** 要求作業系統將自己刪除時結束。這個行程的所有資源 (包括實體記憶體、虛擬記憶體、開啟檔案，以及輸入輸出緩衝區) 都由作業系統收回。
- 一個父行程可以基於若干理由將子行程中止：
  - 子行程已經使用超過配置的資源數量。
  - 指派給子行程的工作已經不再需要。
  - 父行程結束，而作業系統不允許子行程在父行程結束之後繼續執行。



Process Termination

- Wait for termination, returning the pid:

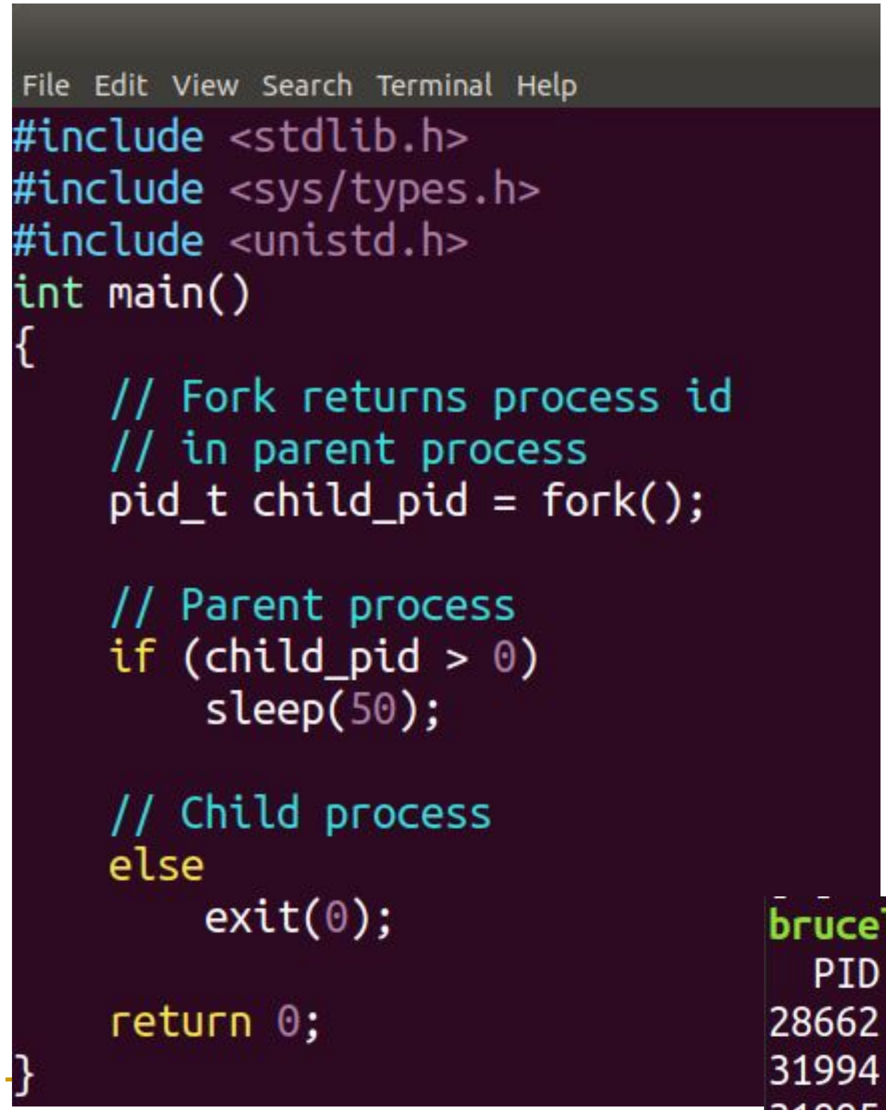
```
pid_t pid;
int status;
pid = wait(&status);
```

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

僵屍行程 (Zombie Process)

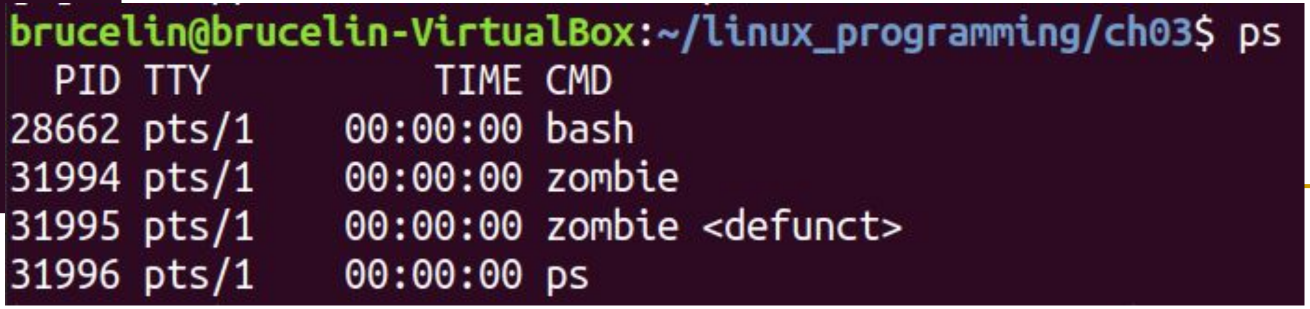
- 在UNIX中使用fork()創建行程時，將複製父行程的地址空間。如果父行程使用wait()，那麼父行程將暫停執行，直到子進程終止。
- 在子行程終止時，會發出一個“SIGCHLD”信號，該信號會由內核傳遞給父行程。父行程在收到“SIGCHLD””後，會從程序表(process table)中刪除子行程資訊。
- 若其父行程不使用wait()等待子行程結束，當子行程結束時，父行程不會知道子行程狀態，因此程序表(process table)裡面仍會保留子行程資訊，這個子行程狀態就是所謂的僵屍行程。

Example



```c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```

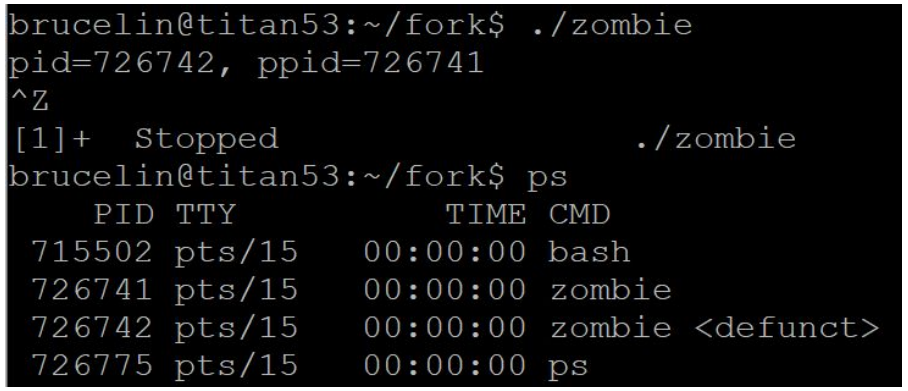
- The child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn’t call wait() and the child process’s entry still exists in the process table.
- Ctrl+z 背景執行
- $ps 顯示執行中的行程
- $fg 前景執行



```
brucelin@brucelin-VirtualBox:~/linux_programming/ch03$ ps
  PID TTY          TIME CMD
28662 pts/1    00:00:00 bash
31994 pts/1    00:00:00 zombie
31995 pts/1    00:00:00 zombie <defunct>
31996 pts/1    00:00:00 ps
```

Example of Zombie process

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
        int i;
        pid_t pid;

        pid=fork();

        if(pid==0){
                printf("pid=%d, ppid=%d\n", getpid(), getppid());
                exit(0);
        }
        else if(pid>0){
                sleep(50);
                exit(0);
        }
        else{
                exit(1);
        }
        return 0;
}
```

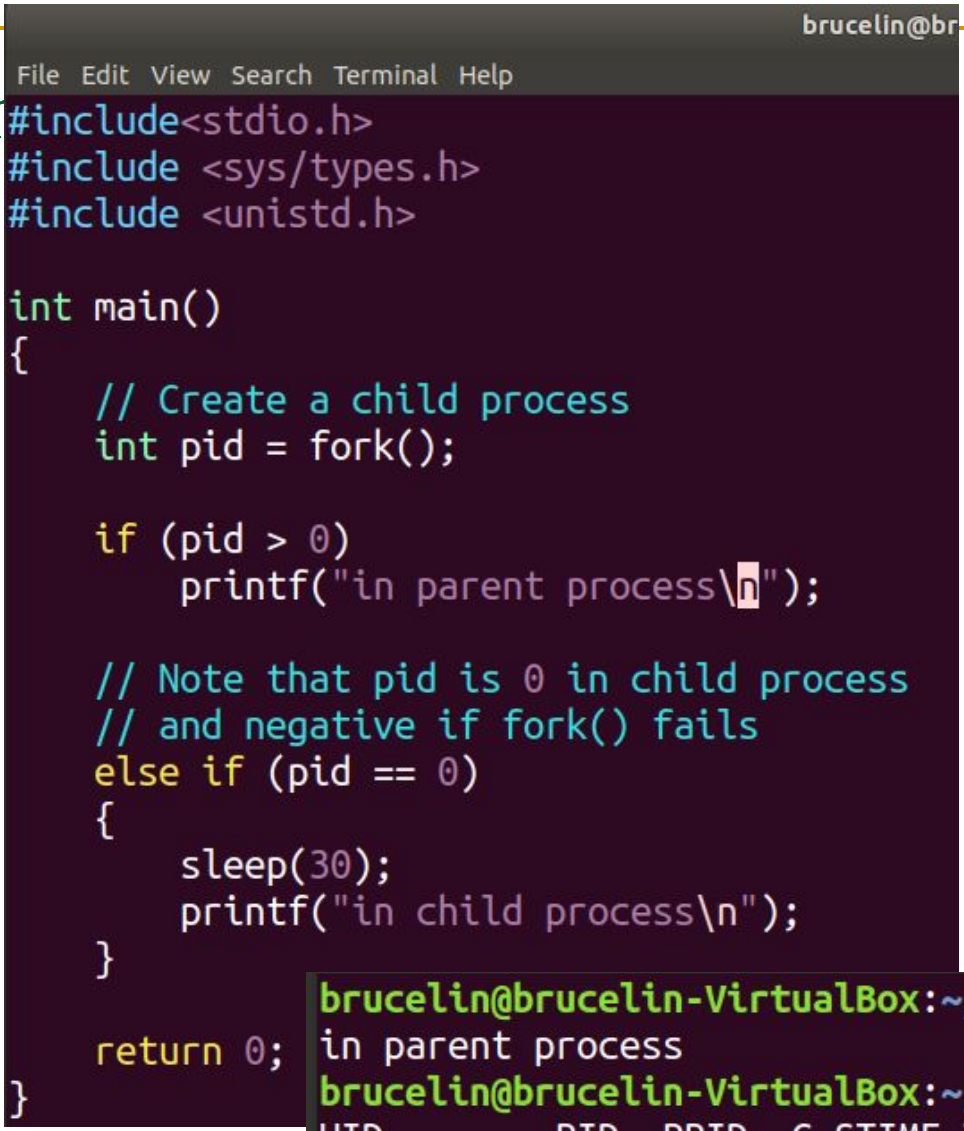


```
brucelin@titan53:~/fork$ ./zombie
pid=726742, ppid=726741
^Z
[1]+  Stopped                 ./zombie
brucelin@titan53:~/fork$ ps
    PID TTY          TIME CMD
 715502 pts/15   00:00:00 bash
 726741 pts/15   00:00:00 zombie
 726742 pts/15   00:00:00 zombie <defunct>
 726775 pts/15   00:00:00 ps
```

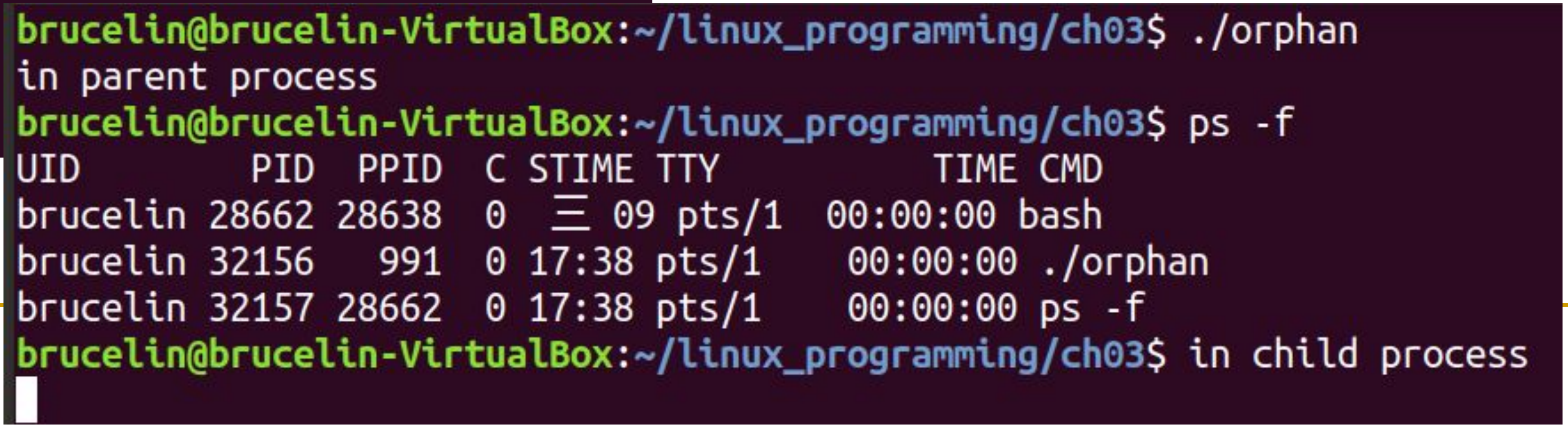
孤兒行程 (Orphan Process)

- 當一個父行程沒有等待子行程結束就自行結束，而其子行程尚未結束，這個子行程就是所謂孤兒行程(orphan process)。
- 但是，一旦其父行程死亡，孤兒行程將很快被init行程認養。

Exam



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

int main()
{
    // Create a child process
    int pid = fork();

    if (pid > 0)
        printf("in parent process\n");

    // Note that pid is 0 in child process
    // and negative if fork() fails
    else if (pid == 0)
    {
        sleep(30);
        printf("in child process\n");
    }

    return 0;
}
```

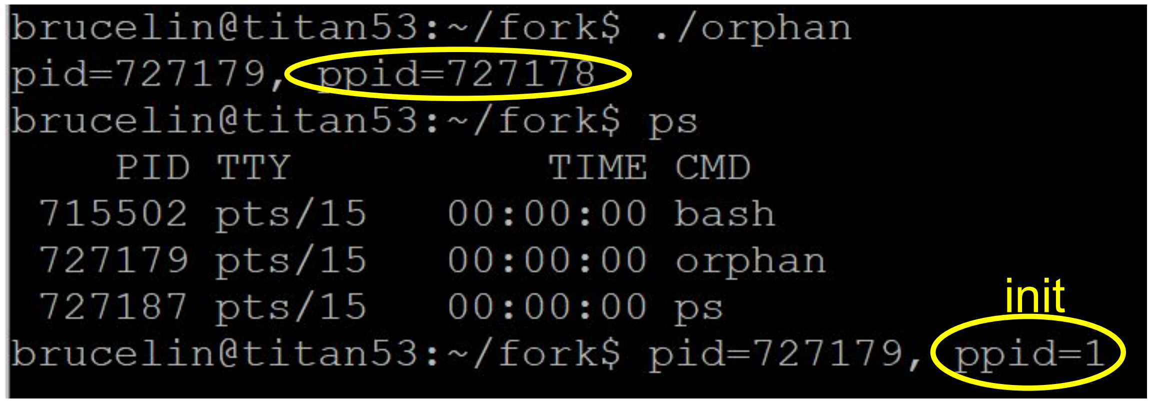


```
brucelin@brucelin-VirtualBox:~/linux_programming/ch03$ ./orphan
in parent process
brucelin@brucelin-VirtualBox:~/linux_programming/ch03$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
brucelin 28662 28638  0  三 09 pts/1  00:00:00 bash
brucelin 32156   991  0 17:38 pts/1    00:00:00 ./orphan
brucelin 32157 28662  0 17:38 pts/1    00:00:00 ps -f
brucelin@brucelin-VirtualBox:~/linux_programming/ch03$ in child process
```

- 當一個父行程沒有等待子行程結束就自行結束，而其子行程尚未結束，這個子行程就是所謂孤兒行程 (orphan process)。
- 這個例子中，父行程很快就結束，但子行程仍在執行。

Example of orphan process

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
	int i;
	pid_t pid;

	pid=fork();

	if(pid==0){
		printf("pid=%d, ppid=%d\n", getpid(), getppid());
		sleep(50);
		printf("pid=%d, ppid=%d\n", getpid(), getppid());
	}
	else{
		exit(0);
	}

}
```



```
brucelin@titan53:~/fork$ ./orphan
pid=727179, ppid=727178
brucelin@titan53:~/fork$ ps
    PID TTY          TIME CMD
 715502 pts/15   00:00:00 bash
 727179 pts/15   00:00:00 orphan
 727187 pts/15   00:00:00 ps
brucelin@titan53:~/fork$ pid=727179, ppid=1
```

Quiz 1

- What is the result?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
int value = 5;
int main(){
        pid_t pid;
        pid = fork();

        if(pid == 0){
                value += 15;
        }
        else{
                wait(NULL);
                printf("Parent: value: %d\n", value);
        }
        return 0;
}
```

Quiz 1

- What is the result?
- Answer is 5 as the child and parent processes each have their own copy of value.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
int value = 5;
int main(){
        pid_t pid;
        pid = fork();

        if(pid == 0){
                value += 15;
        }
        else{
                wait(NULL);
                printf("Parent: value: %d\n", value);
        }
        return 0;
}
```

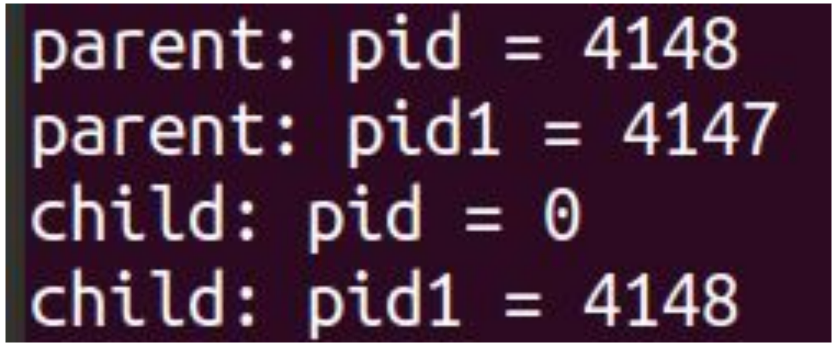
Quiz 2

- What are the values of lines A, B, C, and D?

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

int main(){
        pid_t pid, pid1;
        pid = fork();
        if(pid<0){
                fprintf(stderr, "Fork failed");
                return 1;
        }
        else if(pid == 0){
                pid1 = getpid();
                printf("child: pid = %d\n", pid);//Line A
                printf("child: pid1 = %d\n", pid1);//Line B
        }
        else{
                pid1 = getpid();
                printf("parent: pid = %d\n", pid);//Line C
                printf("parent: pid1 = %d\n", pid1);//Line D
        wait(NULL);
        }
```

Quiz 2

- What are the values of lines A, B, C, and D?
- ANS:



```
parent: pid = 4148
parent: pid1 = 4147
child: pid = 0
child: pid1 = 4148
```

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

int main(){
        pid_t pid, pid1;
        pid = fork();
        if(pid<0){
                fprintf(stderr, "Fork failed");
                return 1;
        }
        else if(pid == 0){
                pid1 = getpid();
                printf("child: pid = %d\n", pid);//Line A
                printf("child: pid1 = %d\n", pid1);//Line B
        }
        else{
                pid1 = getpid();
                printf("parent: pid = %d\n", pid);//Line C
                printf("parent: pid1 = %d\n", pid1);//Line D
        wait(NULL);
        }
```

Quiz 3

- How many processes are created?

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

int main(){
        int i;
        for(i=0; i<4; i++){
            fork();
        }

        return 0;

}
```

Quiz 3

- How many processes are created?
- ANS: 16

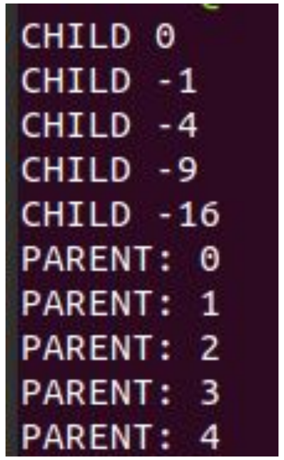
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main(){
        int i;
        for(i=0; i<4; i++){
              fork();
        }

        return 0;

}
```

Quiz 4

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#define SIZE 5
int nums[SIZE] = {0,1,2,3,4};
int main()
{
        int i;
        pid_t pid;
        pid = fork();
        if (pid == 0) {
                for (i = 0; i < SIZE; i++) {
                        nums[i] *= -i;
                        printf("CHILD %d\n",nums[i]); /* LINE X */
                }
        }
        else if (pid > 0) {
                wait(NULL);
                for (i = 0; i < SIZE; i++)
                        printf("PARENT: %d\n",nums[i]); /* LINE Y */
        }
        return 0;
}
```

Quiz 4

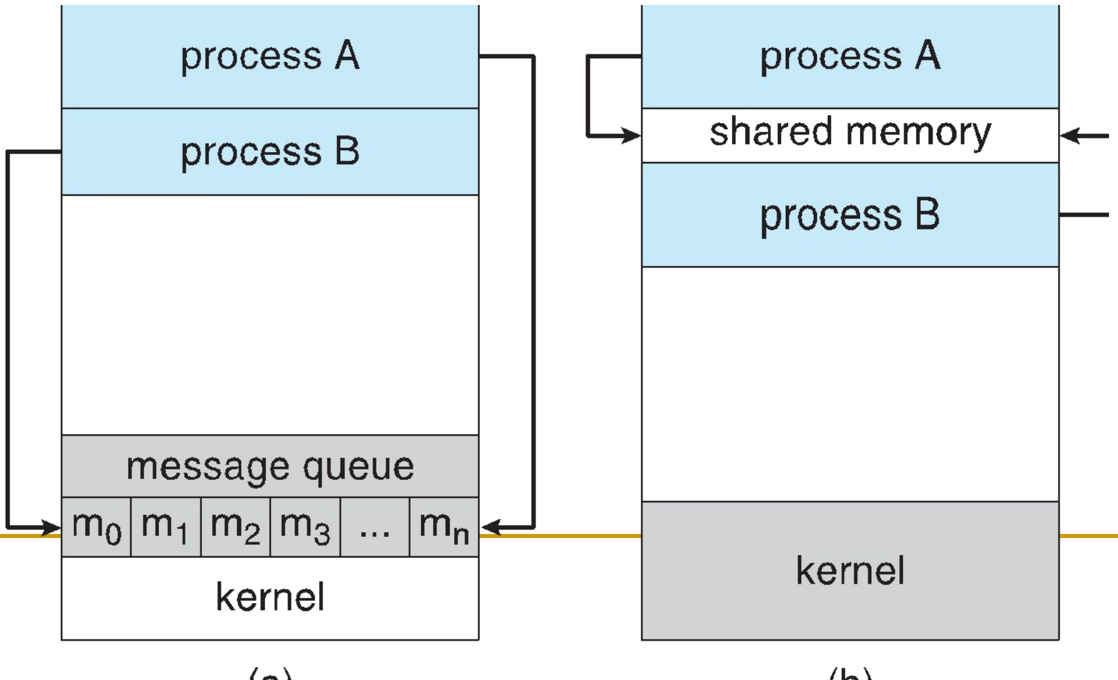
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#define SIZE 5
int nums[SIZE] = {0,1,2,3,4};
int main()
{
        int i;
        pid_t pid;
        pid = fork();
        if (pid == 0) {
                for (i = 0; i < SIZE; i++) {
                        nums[i] *= -i;
                        printf("CHILD %d\n",nums[i]); /* LINE X */
                }
        }
        else if (pid > 0) {
                wait(NULL);
                for (i = 0; i < SIZE; i++)
                        printf("PARENT: %d\n",nums[i]); /* LINE Y */
        }
        return 0;
}
```



```
CHILD 0
CHILD -1
CHILD -4
CHILD -9
CHILD -16
PARENT: 0
PARENT: 1
PARENT: 2
PARENT: 3
PARENT: 4
```

3.4 行程間通訊

- 行程合作(process cooperation)的理由
  - **資訊共享**: 數個使用者可能對相同的一項資訊(例如, 公用檔案)有興 趣, 因此須提供一個環境允許使用者能同時使用這些資源。
  - **加速運算**: 如果希望某一特定工作執行快一點, 則可以分成一些子工作, 每一個子工作都可以和其它子工作平行地執行。
  - **模組化**: 希望以模組的方式來建立系統, 把系統功能分配到數個行程。
  - **方便性**:即使是單一使用者也可能同時執行數項工作。



Producer process using POSIX shared-memory API

- POSIX shared-memory is organized using memory-mapped files(記憶體映射檔案)
- Create shared memory object using shm_open()
- gcc producer.c –o producer -lrt

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

int main()
{
        const int SIZE = 4096;
        const char *name = "OS";
        const char *message0= “Hello";
        const char *message1= “World!";
        int shm_fd;
        void *ptr;
        /*建立shared memory object的名稱並設定權限為-rw-rw-rw- ,
            成功的話回傳一個integer file descriptor*/
        shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

        /* 設定 shared memory 大小為4096 bytes */
        ftruncate(shm_fd, SIZE);

        /* 建立memory-mapped file包含這個shared memory object */
        ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

        sprintf(ptr,"%s", message0);
        ptr += strlen(message0);
        sprintf(ptr,"%s", message1);
        ptr += strlen(message1);

        return 0;
}
```

Consumer process

```
int main()
{
        const char *name = "OS";
        const int SIZE = 4096;

        int shm_fd;
        void *ptr;
        int i;

        /* 開啟shared memory segment，並設定為read-only*/
        shm_fd = shm_open(name, O_RDONLY, 0666);
```

Consumer process

```
        /* 建立memory-mapped file包含這個shared memory object */
    ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

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

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

    return 0;
}
```

3.4.1共用記憶體系統Shared-Memory Solution

- 為了闡述合作行程的觀念，讓我們來看 "生產者-消費者"的問題。生產者(producer)行程產生資訊，消費者(consumer)行程消耗掉這些資訊。

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
      . . .
} item;

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

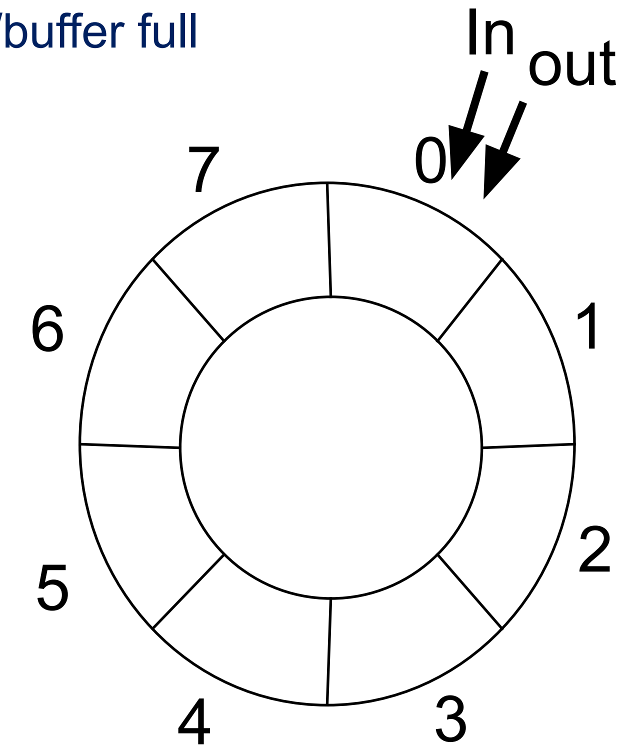
```
//producer
item nextProduced;
while (true) {
          /* Produce an item in nextProduced*/
         while (((in + 1) % BUFFER SIZE count)  == out);
                        /* do nothing -- no free buffers */
        buffer[in] = nextProduced;
          in = (in + 1) % BUFFER SIZE;
}
```

```
//consumer
item nextConsumed;
while (true) {
        while (in == out) ; // do nothing -- nothing to
      consume

        nextConsumed = buffer[out];
         out = (out + 1) % BUFFER SIZE;
}
```

◆**Solution is correct, but can only use BUFFER_SIZE-1 elements**

Circular queue

```
item nextProduced;
while (true) {
        /* Produce an item in nextProduced*/
        while (((in + 1) % BUFFER SIZE)  == out);  //buffer full
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER SIZE;
}


item nextConsumed;
while (true) {
        while (in == out) ; // buffer is empty
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER SIZE;
}
```



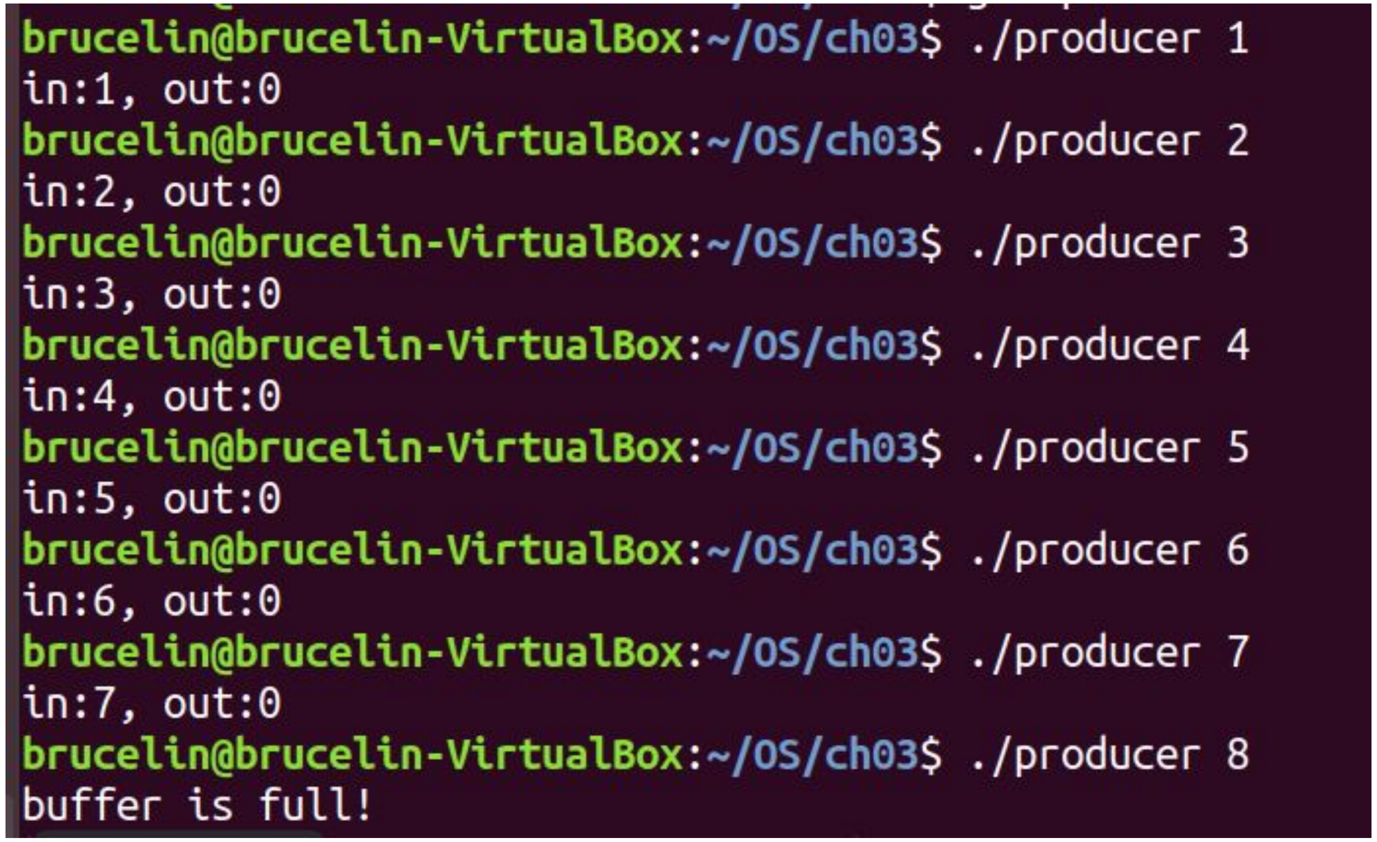
Producer using shared memory

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#define N 8

int main(int argc, char* argv[]){

        const int SIZE = N*sizeof(int);
        const char *shm_buffer_name = "shm_buffer";
        const char *shm_in_name = "shm_in";
        const char *shm_out_name = "shm_out";

        if(argc!=2){
                printf("using command: %s [integer]", argv[0]);
                exit(1);
        }

        int shm_fd, shm_in, shm_out;
        static int *buffer, *in, *out;
```

```
    /* create the shared memory segment */
shm_fd = shm_open(shm_buffer_name, O_CREAT | O_RDWR, 0666);
if (shm_fd == -1) {
    printf("shared memory failed\n");
    exit(-1);
}

shm_in = shm_open(shm_in_name, O_CREAT | O_RDWR, 0666);
if (shm_in == -1) {
    printf("shared memory failed\n");
    exit(-1);
}

shm_out = shm_open(shm_out_name, O_CREAT | O_RDWR, 0666);
if (shm_out == -1) {
    printf("shared memory failed\n");
    exit(-1);
}

/* configure the size of the shared memory segment */
ftruncate(shm_fd, SIZE);
ftruncate(shm_in, sizeof(int));
ftruncate(shm_out, sizeof(int));
```

```
            /* now map the shared memory segment in the address space of the process */
        buffer = (int *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
        if (buffer == MAP_FAILED) {
                printf("Map buffer failed\n");
                return -1;
        }

        in = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_in, 0);
        if (in == MAP_FAILED) {
                printf("Map in failed\n");
                return -1;
        }
        out = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_out, 0);
        if (out == MAP_FAILED) {
                printf(" Map out failed\n");
                return -1;
        }

        while((*in+1)%N==*out){
                printf("buffer is full!\n");
                exit(1);
        }

        buffer[*in] = atoi(argv[1]);
        printf("Produce buffer[%d]: %d\n", *in, buffer[*in]);
        *in = (*in + 1)%N;

        printf("Next in:%d, out:%d\n", *in, *out);
        return 0;
}
```

Example

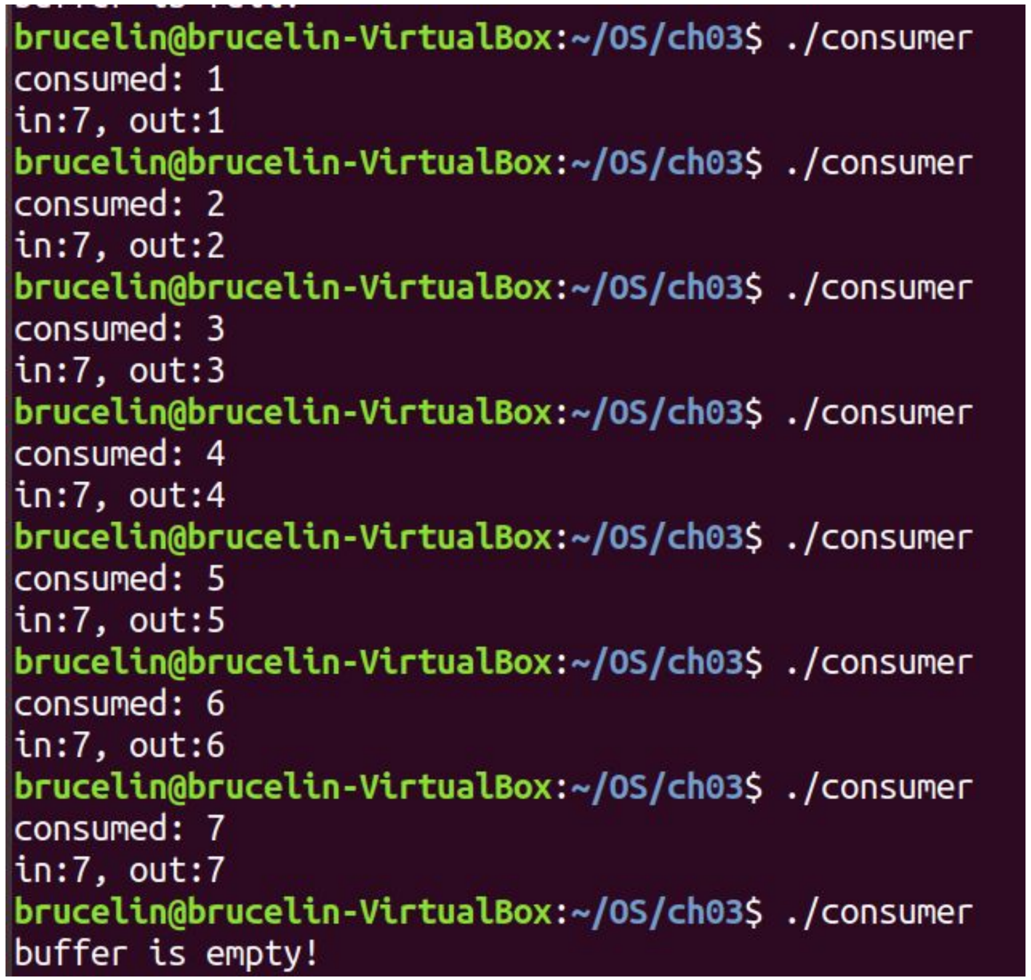


```
bruceIin@bruceIin-VirtualBox:~/OS/ch03$ ./producer 1
in:1, out:0
bruceIin@bruceIin-VirtualBox:~/OS/ch03$ ./producer 2
in:2, out:0
bruceIin@bruceIin-VirtualBox:~/OS/ch03$ ./producer 3
in:3, out:0
bruceIin@bruceIin-VirtualBox:~/OS/ch03$ ./producer 4
in:4, out:0
bruceIin@bruceIin-VirtualBox:~/OS/ch03$ ./producer 5
in:5, out:0
bruceIin@bruceIin-VirtualBox:~/OS/ch03$ ./producer 6
in:6, out:0
bruceIin@bruceIin-VirtualBox:~/OS/ch03$ ./producer 7
in:7, out:0
bruceIin@bruceIin-VirtualBox:~/OS/ch03$ ./producer 8
buffer is full!
```

Consumer using shared memory

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#define N 8

int main()
{
        const int SIZE = 8*sizeof(int);
        const char *shm_buffer_name = "shm_buffer";
        const char *shm_in_name = "shm_in";
        const char *shm_out_name = "shm_out";
        int shm_fd, shm_in, shm_out;
        int *buffer, *in, *out;


        int i;
```

```
/* open the shared memory segment */
        shm_fd = shm_open(shm_buffer_name, O_RDONLY, 0666);
        if (shm_fd == -1) {
                printf("shared memory failed\n");
                exit(-1);
        }

        shm_in = shm_open(shm_in_name, O_CREAT | O_RDWR, 0666);
        if (shm_in == -1) {
                printf("shared memory failed\n");
                exit(-1);
        }
        shm_out = shm_open(shm_out_name, O_CREAT | O_RDWR, 0666);
        if (shm_out == -1) {
                printf("shared memory failed\n");
                exit(-1);
        }

        /* configure the size of the shared memory segment */
        ftruncate(shm_fd, SIZE);
        ftruncate(shm_in, sizeof(int));
        ftruncate(shm_out, sizeof(int));
        /* now map the shared memory segment in the address space of the process */
    buffer = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    if (buffer == MAP_FAILED) {
            printf("Map failed\n");
            exit(-1);
    }
    in = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_in, 0);
    if (in == MAP_FAILED) {
            printf("Map in failed\n");
            return -1;
    }
    out = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_out, 0);
    if (out == MAP_FAILED) {
            printf(" Map out failed\n");
            return -1;
    }

    while(*in==*out){
            printf("buffer is empty!\n");
            exit(1);
    }
    printf("consumed buffer[%d]: %d\n", *out, buffer[*out]);
    *out = (*out + 1) % N;
            printf("in:%d, next out:%d\n", *in, *out);

    return 0;
}
```

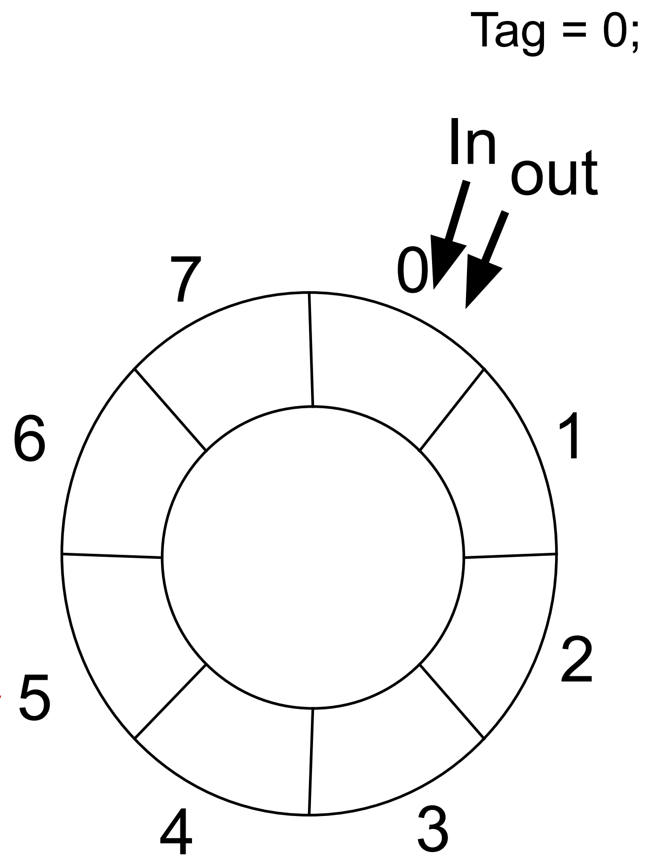


```
brucelin@brucelin-VirtualBox:~/OS/ch03$ ./consumer
consumed: 1
in:7, out:1
brucelin@brucelin-VirtualBox:~/OS/ch03$ ./consumer
consumed: 2
in:7, out:2
brucelin@brucelin-VirtualBox:~/OS/ch03$ ./consumer
consumed: 3
in:7, out:3
brucelin@brucelin-VirtualBox:~/OS/ch03$ ./consumer
consumed: 4
in:7, out:4
brucelin@brucelin-VirtualBox:~/OS/ch03$ ./consumer
consumed: 5
in:7, out:5
brucelin@brucelin-VirtualBox:~/OS/ch03$ ./consumer
consumed: 6
in:7, out:6
brucelin@brucelin-VirtualBox:~/OS/ch03$ ./consumer
consumed: 7
in:7, out:7
brucelin@brucelin-VirtualBox:~/OS/ch03$ ./consumer
buffer is empty!
```

Homework

```
item nextProduced;
tag = 0;
while (true) {
        /* Produce an item in nextProduced*/
         while (in  == out && tag==1);  //buffer full
        buffer[in] = nextProduced;
         in = (in + 1) % BUFFER SIZE;
       if(in==out) tag = 1;
}
```

```
item nextConsumed;
while (true) {
        while (in == out && tag==0) ; // buffer is empty
        nextConsumed = buffer[out];
         out = (out + 1) % BUFFER SIZE;
         if(out==in) tag = 0;
}
```



Writing command line program

- Main function has two arguments, int argc, char*agrv[]
- int main(int argc, char* argv[])
- argc is the number of arguments
- argv is the array string arguments
- ./add 1 2
- argc = 5
- argv[0]: add
- argv[1]:1
- argv[2]:2

3.4.2 訊息傳遞系統Message Passing

- 訊息傳遞提供了允許行程互相溝通和彼此同步而不需要共享相同的位址空間。
- 訊息傳遞設施提供至少兩種操作:send (訊息) 和receive (訊息)。
- 如果兩個行程 P 與 Q 要互相聯繫, 則它們必須互相傳送與接收訊息。為了使它們可這樣做, 因此在它們間必須存在一個通訊鏈。

3.4.2.1 命名

- **直接聯繫 (direct communication)**方法中，每一個要傳送或接收訊息的行程必須先確定聯繫接收者或傳送者的名稱。在這個體系之中，send 與 receive的基本運算定義如下:
  - send (P, message)傳送一個訊息(message)至行程P。
  - receive (Q, message)自行程Q接收一個訊息(message)。

- **間接式聯繫 (indirect communication)**之中，需藉著信箱(mailbox，也叫作埠port)來傳送與接收訊息。這種 send 與 receive 的基本運算之定義如下:
  - send (A, message) 將訊息 (message)傳送至信箱A。
  - receive (A, message) 自信箱A接收一個訊息 (message)。

3.4.2.2 同步化

訊息傳遞可以是等待(blocking)或非等待(nonblocking)，也稱為同步 (synchronous)和非同步 (asynchronous)。

- 等待傳送 (blocking send):傳送行程等待著，直到接收行程或信箱接收訊息。
- 非等待傳送 (nonblocking send):傳送行程送出訊息，即重新操作。
- 等待接收 (blocking receive):接收者等待，直到有效訊息出現。
- 非等待接收 (nonblocking receive):接收者收到有效訊息或無效資料。

POSIX

- **POSIX**是IEEE是IEEE為要在各種UNIX是IEEE為要在各種UNIX作業系統是IEEE為要在各種UNIX作業系統上執行的軟體，而定義API是IEEE為要在各種UNIX作業系統上執行的軟體，而定義API的一系列互相關聯的標準的總稱，其正式稱呼為IEEE 1003，而國際標準名稱為ISO是IEEE為要在各種UNIX作業系統上執行的軟體，而定義API的一系列互相關聯的標準的總稱，其正式稱呼為IEEE 1003，而國際標準名稱為ISO／IEC 9945。
- 此標準源於一個大約開始於1985年的項目。POSIX這個名稱是由理察此標準源於一個大約開始於1985年的項目。POSIX這個名稱是由理察·此標準源於一個大約開始於1985年的項目。POSIX這個名稱是由理察·斯托曼應IEEE的要求而提議的一個易於記憶的名稱。它基本上是**Portable Operating System Interface**（可移植作業系統介面）的縮寫，而**X**則表明其對Unix API的傳承。

- Linux基本上逐步實作了POSIX相容，但並沒有參加正式的POSIX認證。

- 微軟微軟的Windows NT至少部分實作了POSIX相容。

Quiz 5 using shared memory

- Parent process shares data with child process using shared memory

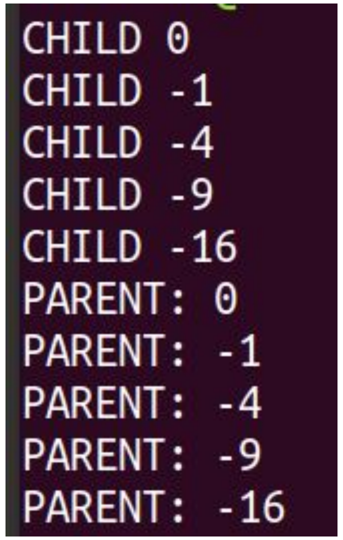
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
static int *glob_var;
int main(void)
{
        glob_var = mmap(NULL, sizeof (int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
        *glob_var = 1;
        if (fork() == 0) {
        *glob_var = 5;
        exit(EXIT_SUCCESS);
         } else {
        wait(NULL);
        printf("%d\n", *glob_var);
        munmap(glob_var, sizeof *glob_var);
         }
         return 0;
}
```

Quiz 6 using shared memory

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/wait.h>
#define SIZE 5

int main()
{
        int i;
        pid_t pid;
        int *nums;
        nums = mmap(NULL, SIZE*sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED  |MAP_ANONYMOUS, -1, 0);
        for(i=0; i<SIZE; i++){
                nums[i] = i;
        }
        pid = fork();
        if (pid == 0) {
                for (i = 0; i < SIZE; i++) {
                        nums[i] *= -i;
                        printf("CHILD %d\n",nums[i]); /* LINE X */
                }
        }
        else if (pid > 0) {
                wait(NULL);
                for (i = 0; i < SIZE; i++)
                        printf("PARENT: %d\n",nums[i]); /* LINE Y */
        }
        return 0;
}
```

Quiz 6 using shared memory

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
#define SIZE 5

int main()
{
        int i;
        pid_t pid;
        int *nums;
        nums = mmap(NULL, SIZE*sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED  |MAP_ANONYMOUS, -1, 0);
        for(i=0; i<SIZE; i++){
                nums[i] = i;
        }
        pid = fork();
        if (pid == 0) {
                for (i = 0; i < SIZE; i++) {
                        nums[i] *= -i;
                        printf("CHILD %d\n",nums[i]); /* LINE X */
                }
        }
        else if (pid > 0) {
                wait(NULL);
                for (i = 0; i < SIZE; i++)
                        printf("PARENT: %d\n",nums[i]); /* LINE Y */
        }
        return 0;
}
```



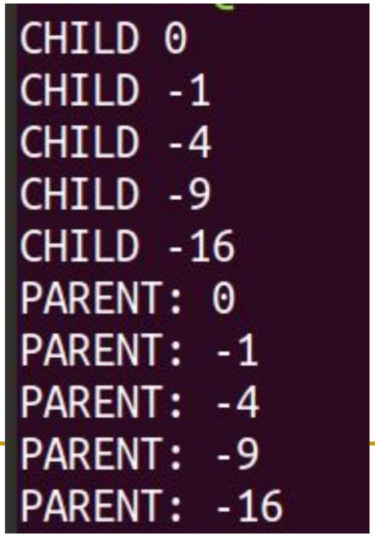
```
CHILD 0
CHILD -1
CHILD -4
CHILD -9
CHILD -16
PARENT: 0
PARENT: -1
PARENT: -4
PARENT: -9
PARENT: -16
```

Quiz 7: POSIX shared memory

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <fcntl.h>

#define SIZE 5

int main()
{
        int i;
        pid_t pid;
        int shm_fd;//shared memory file descriptor
        int* ptr;//pointer to shared memory object
        const char* name = "sharedMem";//name of shared memory
        const int SHM_SIZE = SIZE * sizeof(int);//size of shared memory
        /*Create the shared memory object*/
        shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

        /*Configure the size of the shared memory*/
        ftruncate(shm_fd, SHM_SIZE);

        /*memory map the shared memory object*/
        ptr = (int*)mmap(0, SHM_SIZE, PROT_WRITE | PROT_READ, MAP_SHARED, shm_fd, 0);

        for(i=0; i < SIZE; i++){
            ptr[i] = i;
        }
```

```
    pid = fork();
    if (pid == 0) {
        for (i = 0; i < SIZE; i++) {
            ptr[i] *= -i;
            printf("CHILD %d\n",ptr[i]); /* LINE X */
        }
    }
    else if (pid > 0) {
        wait(NULL);
        for (i = 0; i < SIZE; i++)
            printf("PARENT: %d\n",ptr[i]); /* LINE Y */
    }

    return 0;
}
```

```
        pid = fork();
        if (pid == 0) {
                for (i = 0; i < SIZE; i++) {
                        ptr[i] *= -i;
                        printf("CHILD %d\n",ptr[i]); /* LINE X */
                }
        }
        else if (pid > 0) {
                wait(NULL);
                for (i = 0; i < SIZE; i++)
                        printf("PARENT: %d\n",ptr[i]); /* LINE Y */
        }

        return 0;
}
```



```
CHILD 0
CHILD -1
CHILD -4
CHILD -9
CHILD -16
PARENT: 0
PARENT: -1
PARENT: -4
PARENT: -9
PARENT: -16
```

3.5.2 IPC system: Mach(國際發音：[mʌk])

- 由Carnegie Mellon大學所發展出的Mach作業系統，為典型的**訊息式(Message passing)**的作業系統範例。Mach核心支援多元任務的產生和刪除，這些任務類似於行程，但具有多重的控制執行緒。

- Mach中大部份的通訊(包括大部份系統呼叫和所有任務之間的資訊)都是由**訊息(messages)**完成。訊息都是由**信箱(mailboxes)**來傳送及接收。

- Mach稱信箱(mailboxes)為**埠(port)。**

- 只需要三個system calls, msg_send(), msg_receive(), and msg_rpc()

Mach

- 每一個task需要兩個信箱mailboxes, 稱為Kernel mailbox and Notify mailbox
- port_allocate()用來建立新的信箱
- 萬一信箱已經滿了, 執行緒有以下四種處理方式
  - 無限的等待直到信箱有空間
  - 最多等待n milliseconds
  - 不等待直接返回
  - 透過OS暫存訊息, 當信箱有空間, 再將暫存的訊息給發送者發送給信箱
- Major problem: double copying of message
- Solution: virtual-memory-management

3.5.3 Windows

- Windows作業系統是現代設計的一個例子，它採用模組化設計以便增進功能和降低製作新特性所需的時間。
- Windows 提供支援多元操作環境(multiple operating environment)或稱為子系統(subsystem)。
- 應用程式透過訊息傳遞與子系統(subsystem)溝通。這些應用程式可以視為Windows子系統的委託人(client)。

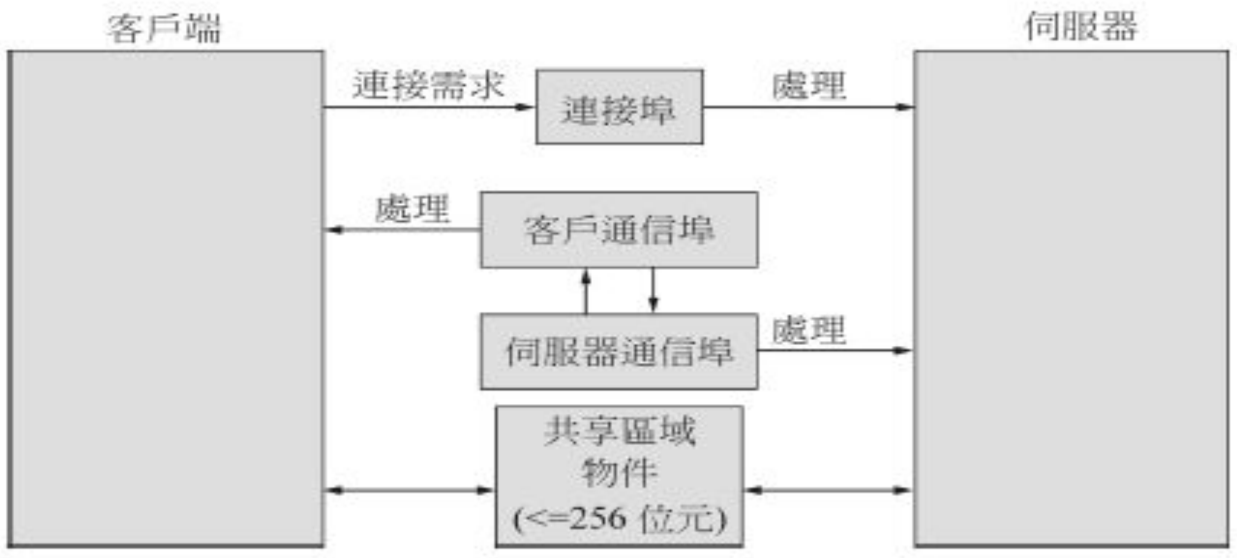


圖 3.17 Windows XP 中的區域程序呼叫

Windows

- Windows透過Advanced local procedure call (ALPC)進行兩個行程間的訊息傳遞
- 使用port(類似信箱)來建立與維護兩個行程間的連結，一個為connection port，一個為communication port
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

3.6 客戶(client)-伺服器(Server)的通信

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

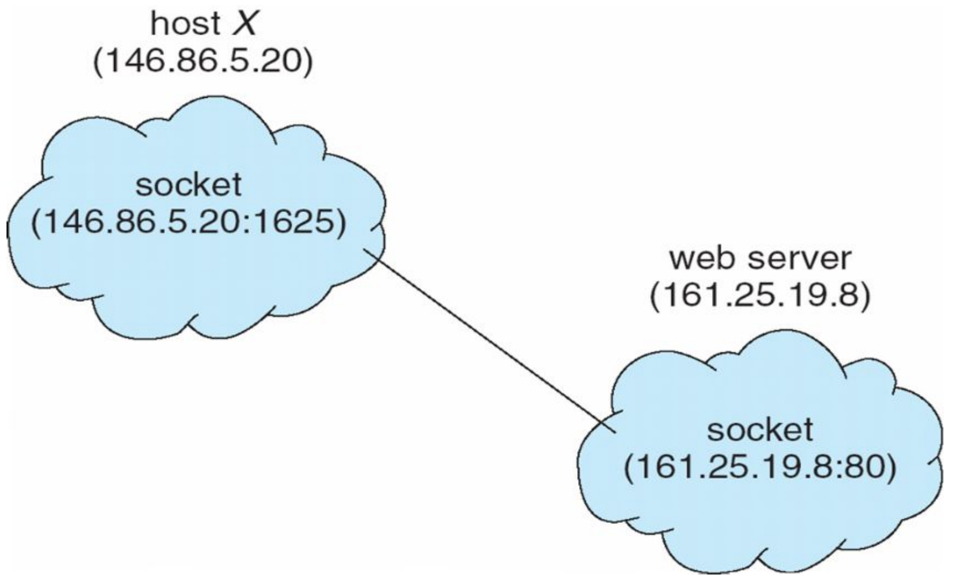
3.6.1 插座(Socket)

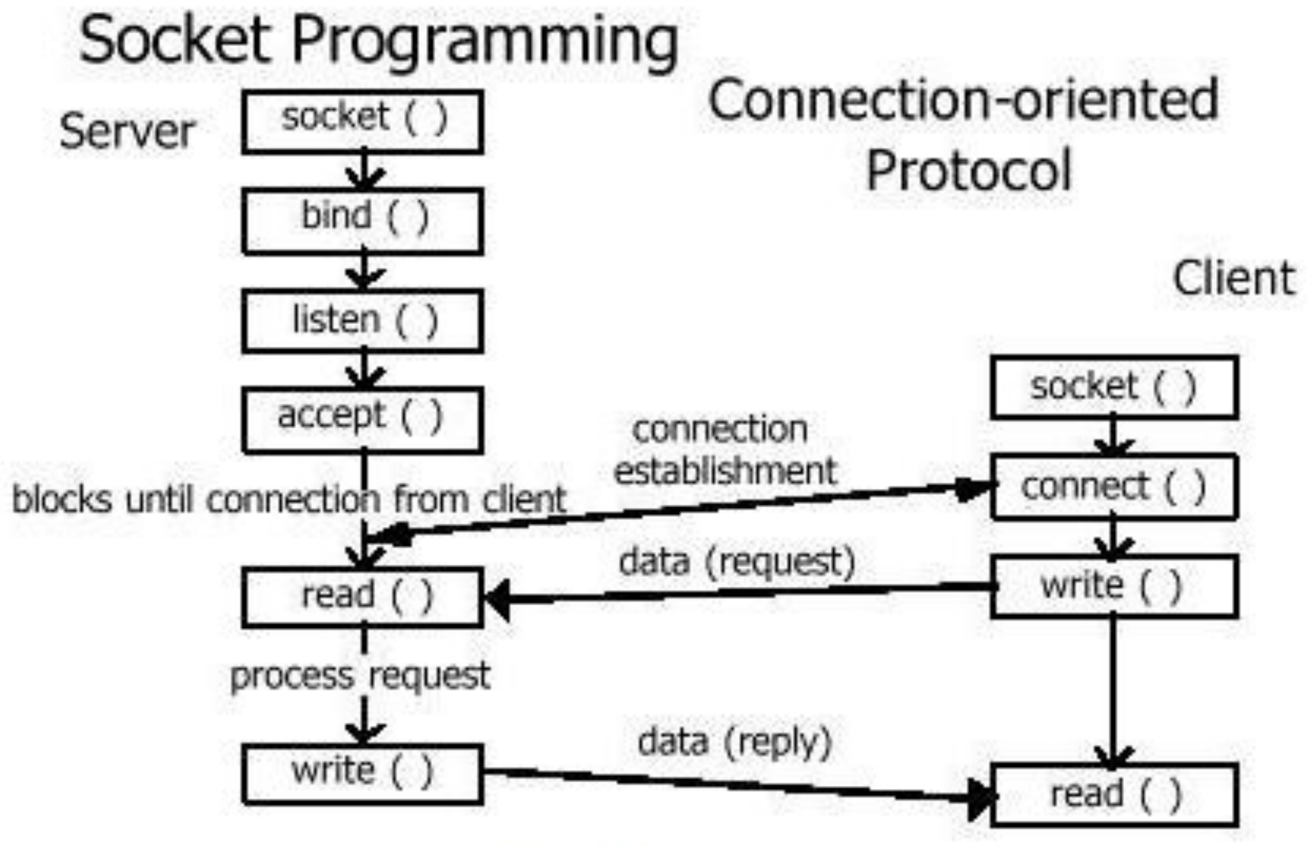
- 插座(Socket)定義成通信的終端。
- 一組行程使用一對插座 (雙方各一個)在網路上通信。
- 一個插座包含一個IP位址和一個埠號碼 (Port number)所組成。
  - The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**

- All ports below 1024 are ***well known***, used for standard services

- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication





Socket Programming

Sockets in Java

- Three types of sockets
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**
  - **MulticastSocket** class– data can be sent to multiple recipients

Sockets in Java

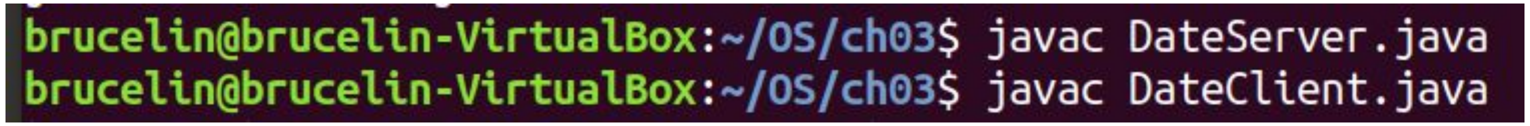
```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args)  {
      try {
            ServerSocket sock = new ServerSocket(6013);
            // now listen for connections
            while (true) {
               Socket client = sock.accept();
               // we have a connection
               PrintWriter pout = new PrintWriter(client.getOutputStream(), true);
               // write the Date to the socket
               pout.println(new java.util.Date().toString());
               // close the socket and resume listening for more connections
               client.close();
            }
      }
      catch (IOException ioe) {
            System.err.println(ioe);
      }
   }
}
```

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args)  {
        try {
      // this could be changed to an IP name or address other than the localhost
      Socket sock = new Socket("127.0.0.1", 6013);
      InputStream in = sock.getInputStream();
      BufferedReader bin = new BufferedReader(new InputStreamReader(in));

      String line;
      while( (line = bin.readLine()) != null)
              System.out.println(line);
              sock.close();
      }
      catch (IOException ioe) {
              System.err.println(ioe);
      }
    }
}
```

Java on Ubuntu

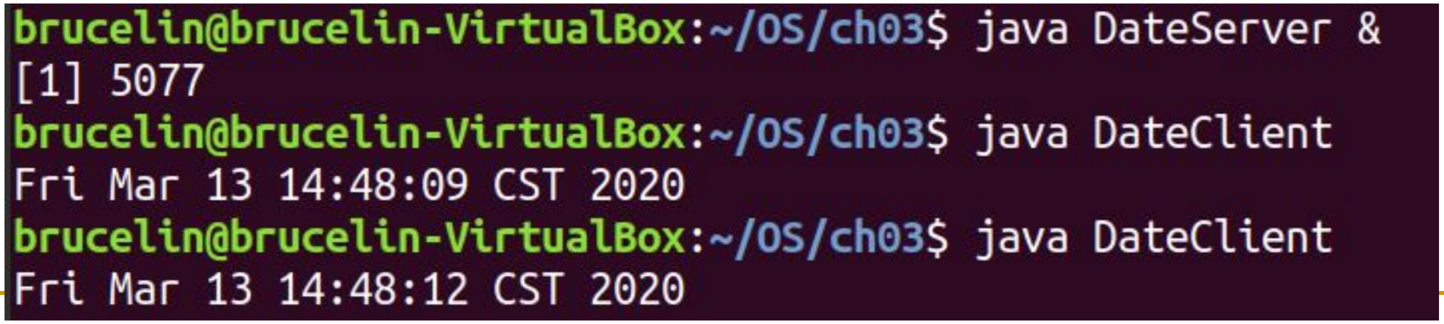
- **How do I set up the JAVA environment on Ubuntu?**
  $sudo apt install default-jdk

- **How do I compile a JAVA program on Ubuntu?**
  $javac program_name.java



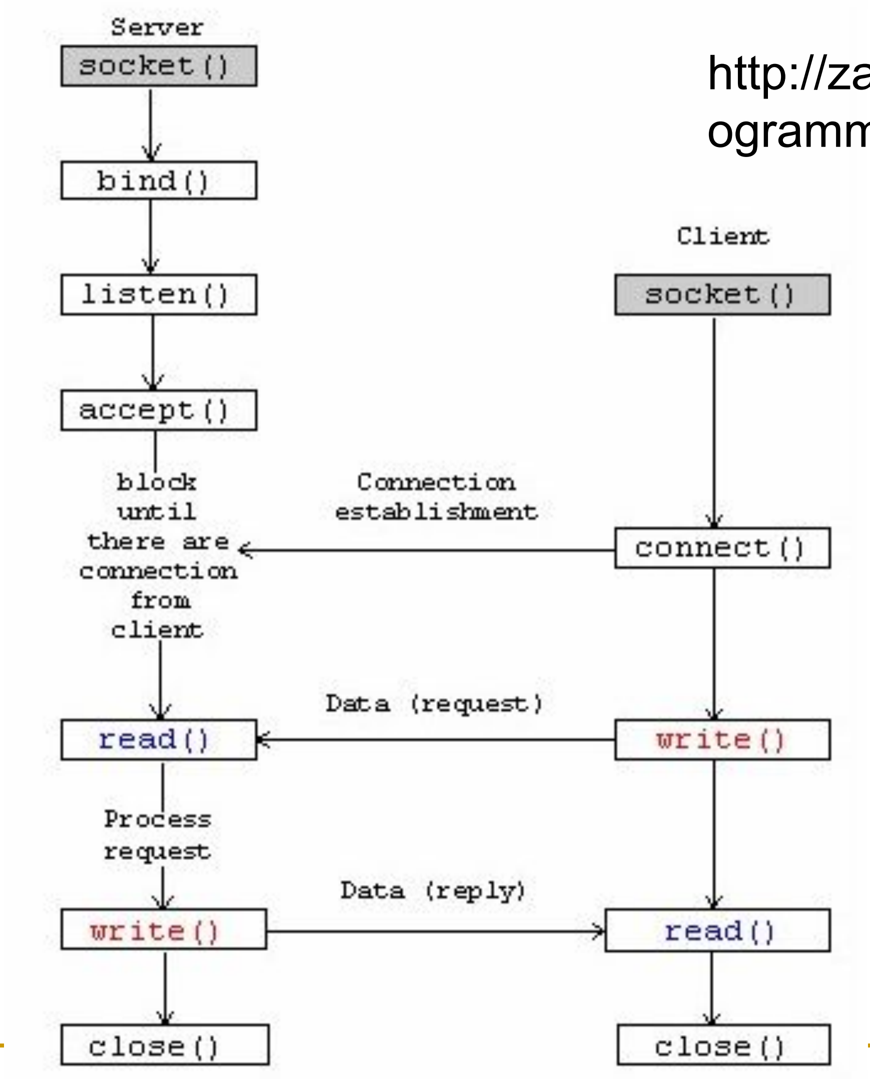
```
brucelin@brucelin-VirtualBox:~/OS/ch03$ javac DateServer.java
brucelin@brucelin-VirtualBox:~/OS/ch03$ javac DateClient.java
```

- **How do I execute a JAVA program on Ubuntu?**
  $java program_name



```
brucelin@brucelin-VirtualBox:~/OS/ch03$ java DateServer &
[1] 5077
brucelin@brucelin-VirtualBox:~/OS/ch03$ java DateClient
Fri Mar 13 14:48:09 CST 2020
brucelin@brucelin-VirtualBox:~/OS/ch03$ java DateClient
Fri Mar 13 14:48:12 CST 2020
```

TCP Socket on Linux



http://zake7749.github.io/2015/03/17/SocketProgramming/

Server example in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc , char *argv[])

{
    //socket的建立
    char inputBuffer[256] = {};
    char message[] = {"Hi,this is server.\n"};
    int sockfd = 0, forClientSockfd = 0;
    sockfd = socket(AF_INET , SOCK_STREAM , 0);

    if (sockfd == -1){
        printf("Fail to create a socket.");
    }
```

- AF_UNIX/AF_LOCAL：用在本機程序與程序間的傳輸，讓兩個程序共享一個檔案系統(file system)
- AF_INET , AF_INET6 ：讓兩台主機透過網路進行資料傳輸，AF_INET使用的是IPv4協定，而AF_INET6則是IPv6協定。

- SOCK_STREAM：提供一個序列化的連接導向位元流，可以做位元流傳輸。對應的protocol為TCP。
- SOCK_DGRAM：提供的是一個一個的資料包(datagram)，對應的protocol為UDP

```
//socket的連線
   struct sockaddr_in serverInfo, clientInfo;
   int addrlen = sizeof(clientInfo);
   bzero(&serverInfo, sizeof(serverInfo));

   serverInfo.sin_family = PF_INET;
   serverInfo.sin_addr.s_addr = INADDR_ANY;
   serverInfo.sin_port = htons(8700);
   bind(sockfd, (struct sockaddr *)&serverInfo, sizeof(serverInfo));
   listen(sockfd,5);

   while(1){
       forClientSockfd = accept(sockfd, (struct sockaddr*) &clientInfo, &addrlen);
       send(forClientSockfd, message, sizeof(message), 0);
       recv(forClientSockfd, inputBuffer, sizeof(inputBuffer), 0);
       printf("Get:%s\n", inputBuffer);
   }
   return 0;
}
```

sockaddr_in為Ipv4結構

htons()就是Host TO Network Short integer的縮寫，它將本機端的字節序(endian)轉換成了網路端的字節序。

sockfd是socket的描述符，5代表最多能有5個人能連入server

Client example in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc , char *argv[])
{

    //socket的建立
    int sockfd = 0;
    sockfd = socket(AF_INET , SOCK_STREAM , 0);

    if (sockfd == -1){
        printf("Fail to create a socket.");
    }
//socket的連線
struct sockaddr_in info;
bzero(&info, sizeof(info));
info.sin_family = PF_INET;

//localhost test
info.sin_addr.s_addr = inet_addr("127.0.0.1");
info.sin_port = htons(8700);
int err = connect(sockfd, (struct sockaddr *)&info,sizeof(info));
if(err==-1){
    printf("Connection error");
}
//Send a message to server
char message[] = {"Hi there"};
char receiveMessage[100] = {};
send(sockfd, message, sizeof(message),0);
recv(sockfd, receiveMessage, sizeof(receiveMessage), 0);

printf("%s", receiveMessage);
printf("close Socket\n");
close(sockfd);
return 0;
}
```

sockaddr_in為Ipv4結構

iner_addr()把IP地址numbers-and-dots notation轉換為整數形式

htons()就是Host TO Network Short integer的縮寫，它將本機端的字節序(endian)轉換成了網路端的字節序。

3.6.2 遠程程序呼叫(RPC)

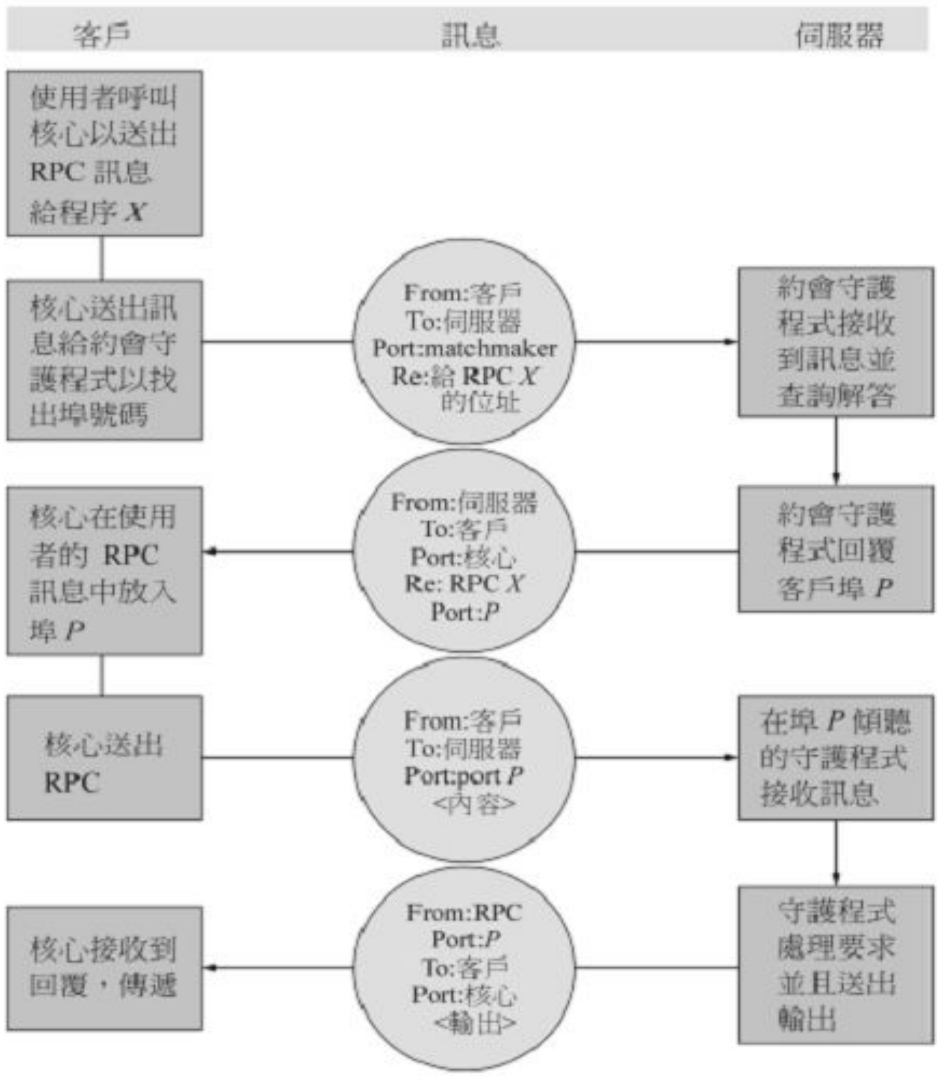
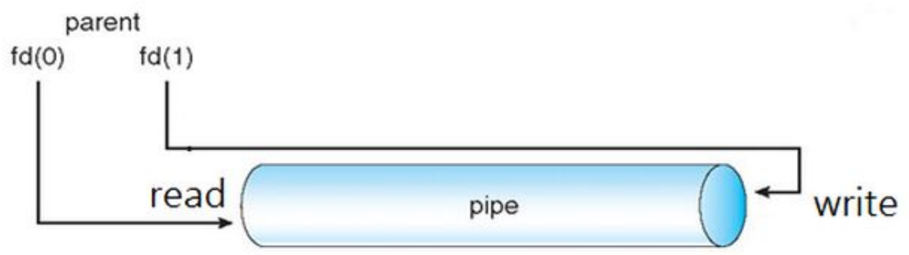
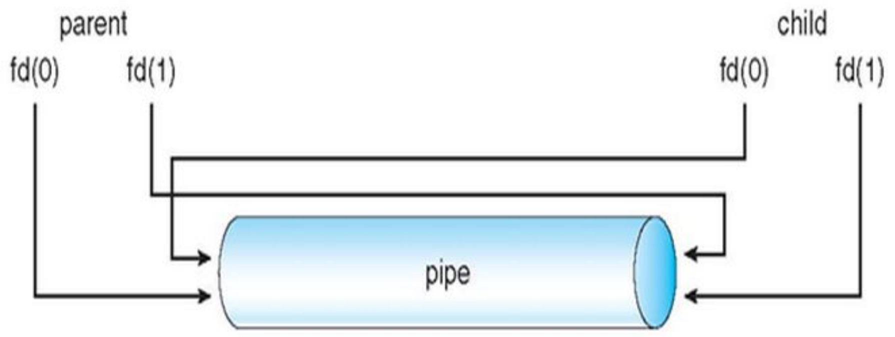

客戶
訊息
伺服器
使用者呼叫核心以送出RPC 訊息給程序 X
核心送出訊息給約會守護程式以找出埠號碼
From:客戶 To:伺服器 Port:matchmaker Re:給 RPC X 的位址
約會守護程式接收到訊息並查詢解答
約會守護程式回覆客戶埠 P
From:伺服器 To:客戶 Port:核心 Re: RPC X Port:P
核心在使用者的 RPC 訊息中放入埠 P
核心送出 RPC
From:客戶 To:伺服器 Port:port P <內容>
在埠 P 傾聽的守護程式接收訊息
守護程式處理要求並且送出輸出
From:RPC Port:P To:客戶 Port:核心 <輸出>
核心接收到回覆，傳遞

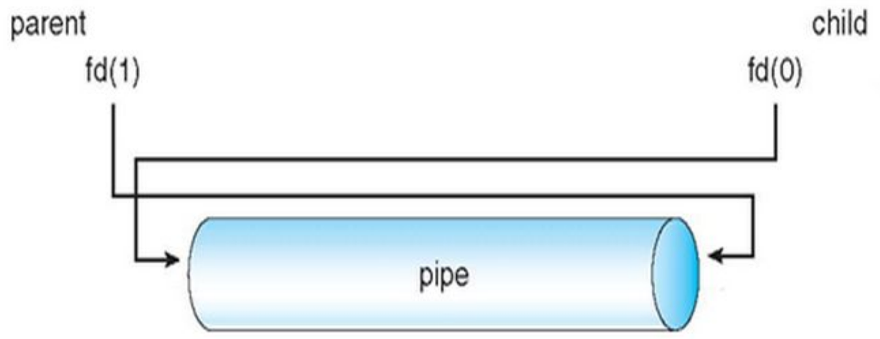
圖 3.21 遠程程序呼叫（RPC）的執行



(a) A parent process creates a pipe.



(b) The parent process forks a child process. Both processes have read (fd[0]) and write (fd[1]) terminals.

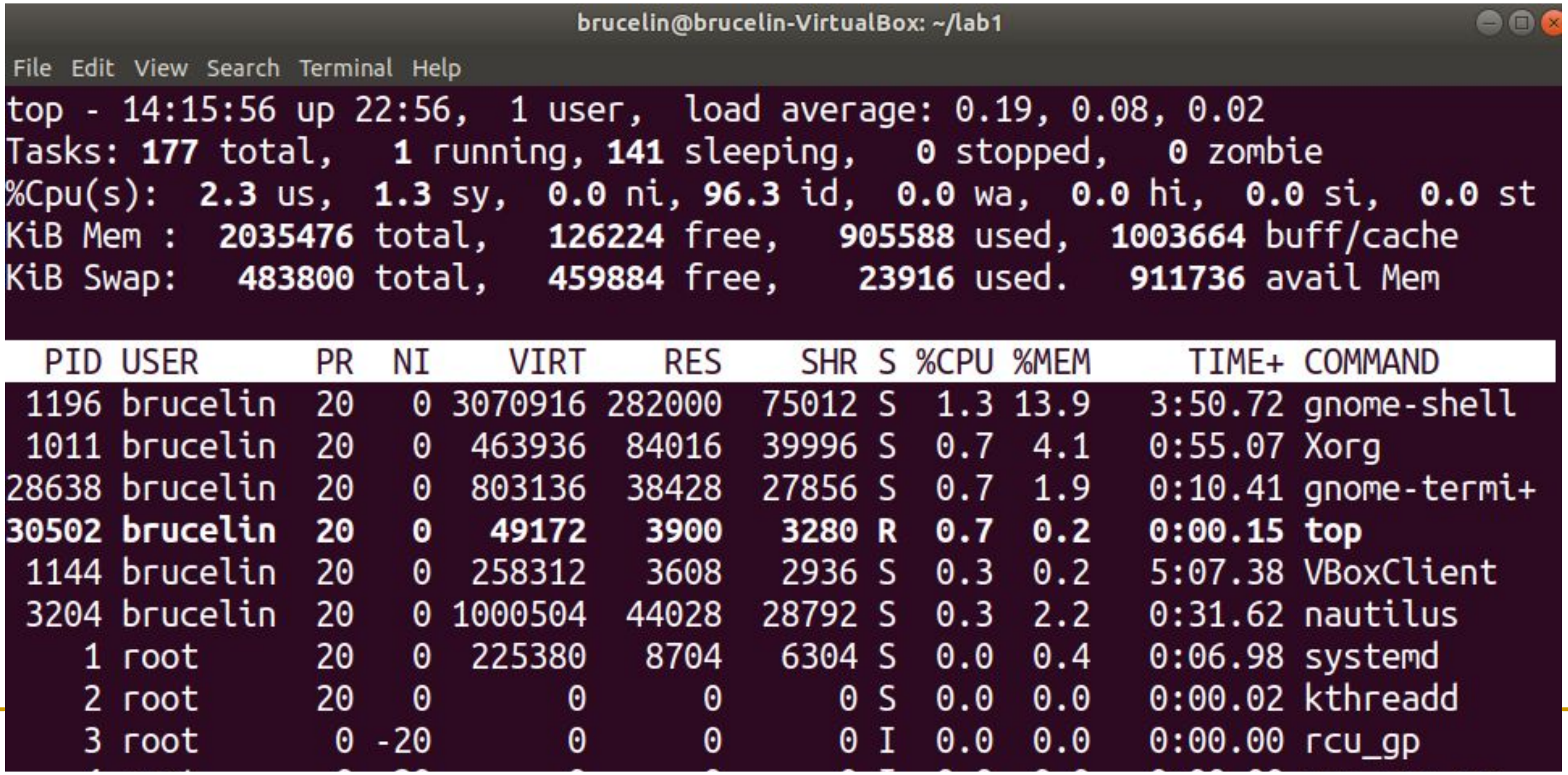


(c) The parent process closes the read terminal (fd[0]) while the child process closes the write terminal(fd[1]).

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.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];
        pid_t pid;
        int fd[2];

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

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

        if (pid < 0) {
          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("child read %s\n",read_msg);

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

        return 0;
}
```

Process Management

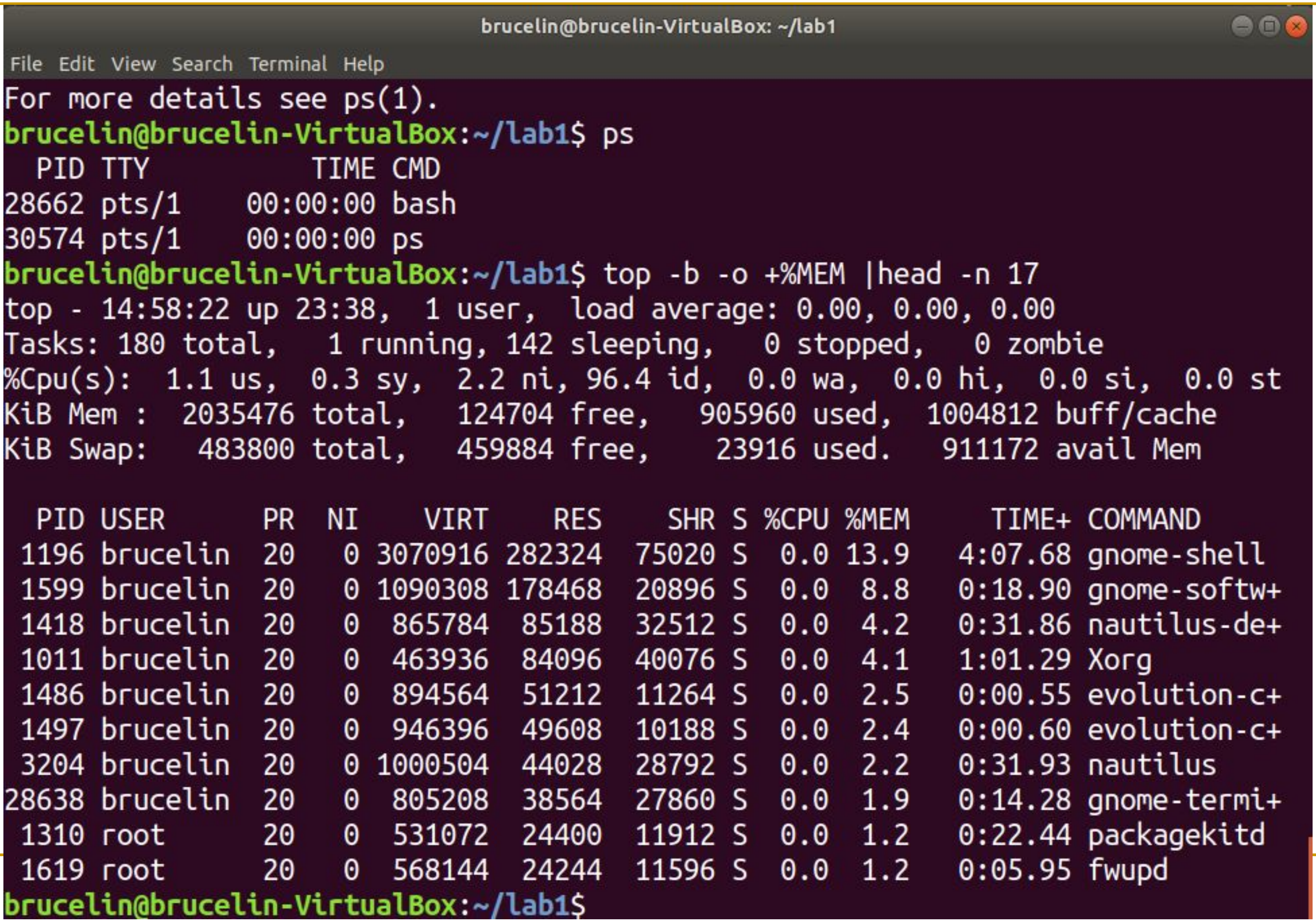
- Process is a unit of running program
- Each process has some information, like process ID, owner, priority, etc
- Example: Output of “top” command


brucelin@brucelin-VirtualBox: ~/lab1
File Edit View Search Terminal Help

```
top - 14:15:56 up 22:56,  1 user,  load average: 0.19, 0.08, 0.02
Tasks: 177 total,   1 running, 141 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.3 us,  1.3 sy,  0.0 ni, 96.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2035476 total,   126224 free,   905588 used,  1003664 buff/cache
KiB Swap:   483800 total,   459884 free,    23916 used.   911736 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1196 brucelin  20   0 3070916 282000  75012 S  1.3 13.9   3:50.72 gnome-shell
 1011 brucelin  20   0  463936  84016  39996 S  0.7  4.1   0:55.07 Xorg
28638 brucelin  20   0  803136  38428  27856 S  0.7  1.9   0:10.41 gnome-termi+
30502 brucelin  20   0   49172   3900   3280 R  0.7  0.2   0:00.15 top
 1144 brucelin  20   0  258312   3608   2936 S  0.3  0.2   5:07.38 VBoxClient
 3204 brucelin  20   0 1000504  44028  28792 S  0.3  2.2   0:31.62 nautilus
    1 root      20   0  225380   8704   6304 S  0.0  0.4   0:06.98 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.02 kthreadd
    3 root       0 -20       0      0      0 I  0.0  0.0   0:00.00 rcu_gp
```

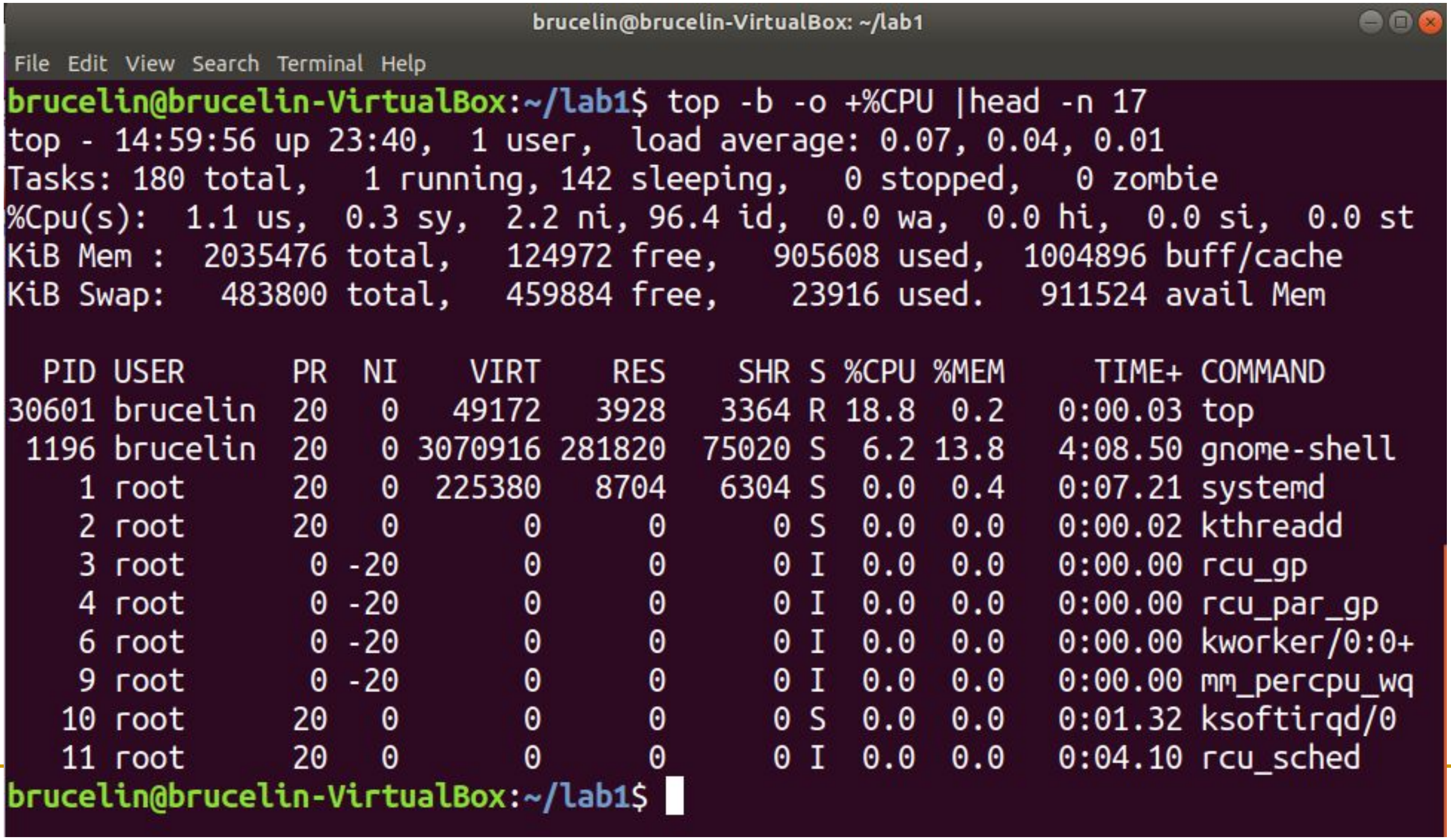
top

- 顯示即時的系統負載狀態
- 這行指令可將系統行程以記憶體的使用賴排序後，以 batch 模式輸出報表，並且只保留前 10 個最耗費記憶體的行程
  - top -b -o +%MEM | head -n 17
  - -b 參數是 batch 模式的意思
  - -o 參數則是設定以記憶體用量來排序行程
  - head -n 17 則是篩選 top 輸出的文字內容，只保留前 17 行，剩餘的內容則捨棄。


brucelin@brucelin-VirtualBox: ~/lab1
File Edit View Search Terminal Help

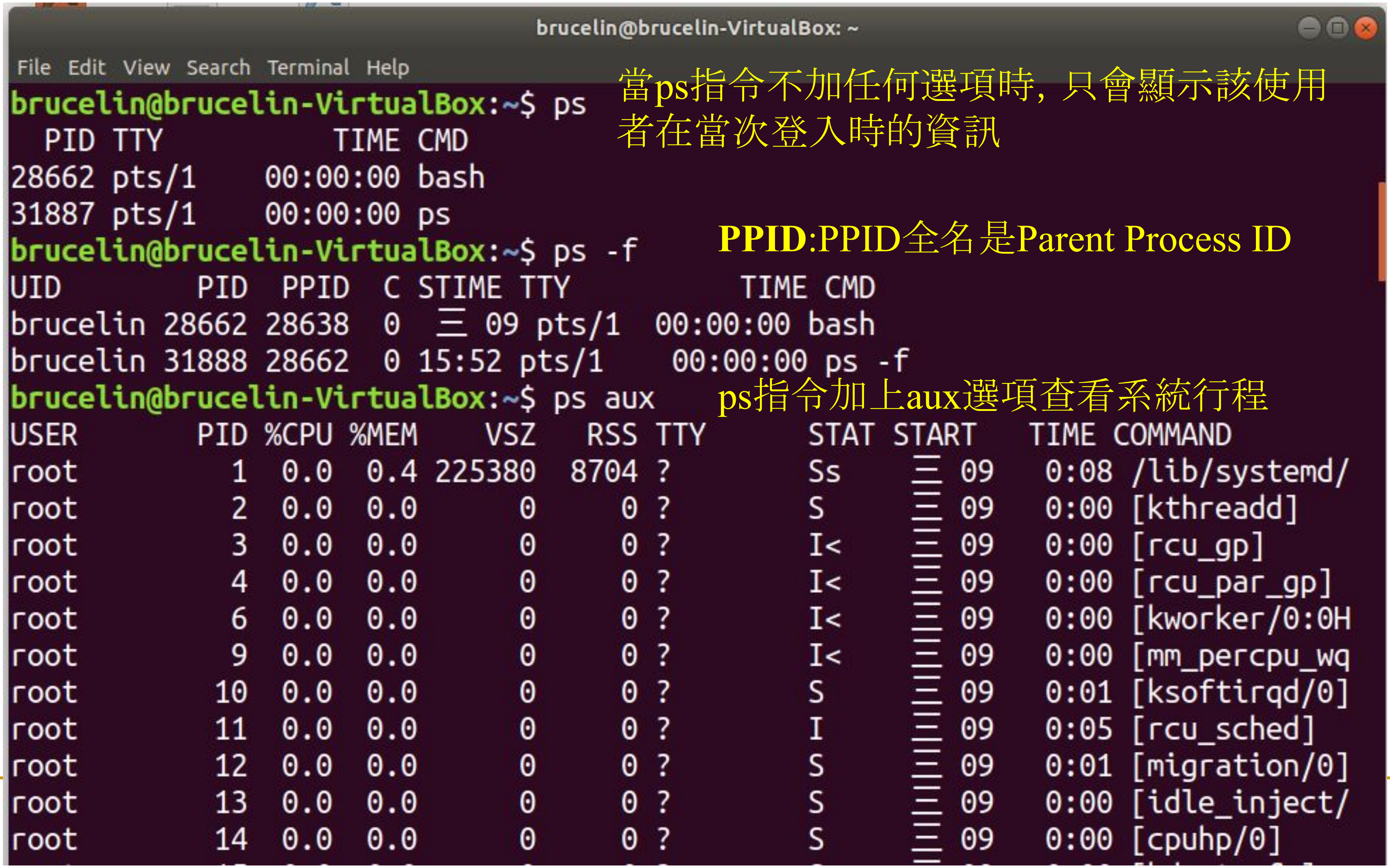
```
For more details see ps(1).
brucelin@brucelin-VirtualBox:~/lab1$ ps
  PID TTY          TIME CMD
28662 pts/1    00:00:00 bash
30574 pts/1    00:00:00 ps
brucelin@brucelin-VirtualBox:~/lab1$ top -b -o +%MEM |head -n 17
top - 14:58:22 up 23:38,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 180 total,   1 running, 142 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.1 us,  0.3 sy,  2.2 ni, 96.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2035476 total,   124704 free,   905960 used,  1004812 buff/cache
KiB Swap:   483800 total,   459884 free,    23916 used.   911172 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1196 brucelin  20   0 3070916 282324  75020 S  0.0 13.9   4:07.68 gnome-shell
 1599 brucelin  20   0 1090308 178468  20896 S  0.0  8.8   0:18.90 gnome-softw+
 1418 brucelin  20   0  865784  85188  32512 S  0.0  4.2   0:31.86 nautilus-de+
 1011 brucelin  20   0  463936  84096  40076 S  0.0  4.1   1:01.29 Xorg
 1486 brucelin  20   0  894564  51212  11264 S  0.0  2.5   0:00.55 evolution-c+
 1497 brucelin  20   0  946396  49608  10188 S  0.0  2.4   0:00.60 evolution-c+
 3204 brucelin  20   0 1000504  44028  28792 S  0.0  2.2   0:31.93 nautilus
28638 brucelin  20   0  805208  38564  27860 S  0.0  1.9   0:14.28 gnome-termi+
 1310 root      20   0  531072  24400  11912 S  0.0  1.2   0:22.44 packagekitd
 1619 root      20   0  568144  24244  11596 S  0.0  1.2   0:05.95 fwupd
brucelin@brucelin-VirtualBox:~/lab1$
```

- 若要找出最耗費 CPU 資源的行程, 則改用 CPU 使用量來排序即可:
  - top -b -o +%CPU | head -n 17


brucelin@brucelin-VirtualBox: ~/lab1
File Edit View Search Terminal Help

```
brucelin@brucelin-VirtualBox:~/lab1$ top -b -o +%CPU |head -n 17
top - 14:59:56 up 23:40,  1 user,  load average: 0.07, 0.04, 0.01
Tasks: 180 total,   1 running, 142 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.1 us,  0.3 sy,  2.2 ni, 96.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2035476 total,   124972 free,   905608 used,  1004896 buff/cache
KiB Swap:   483800 total,   459884 free,    23916 used.   911524 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
30601 brucelin  20   0   49172   3928   3364 R 18.8  0.2   0:00.03 top
 1196 brucelin  20   0 3070916 281820  75020 S  6.2 13.8   4:08.50 gnome-shell
    1 root      20   0  225380   8704   6304 S  0.0  0.4   0:07.21 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.02 kthreadd
    3 root       0 -20       0      0      0 I  0.0  0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I  0.0  0.0   0:00.00 rcu_par_gp
    6 root       0 -20       0      0      0 I  0.0  0.0   0:00.00 kworker/0:0+
    9 root       0 -20       0      0      0 I  0.0  0.0   0:00.00 mm_percpu_wq
   10 root      20   0       0      0      0 S  0.0  0.0   0:01.32 ksoftirqd/0
   11 root      20   0       0      0      0 I  0.0  0.0   0:04.10 rcu_sched
brucelin@brucelin-VirtualBox:~/lab1$ 
```

ps(process status)



```
bruce1in@brucelin-VirtualBox:~$ ps
  PID TTY          TIME CMD
28662 pts/1    00:00:00 bash
31887 pts/1    00:00:00 ps
```

當ps指令不加任何選項時，只會顯示該使用者在當次登入時的資訊

```
brucelin@brucelin-VirtualBox:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
brucelin 28662 28638  0  三 09 pts/1   00:00:00 bash
brucelin 31888 28662  0 15:52 pts/1    00:00:00 ps -f
```

PPID:PPID全名是Parent Process ID

```
brucelin@brucelin-VirtualBox:~$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.4 225380  8704 ?        Ss    三 09   0:08 /lib/systemd/
root         2  0.0  0.0      0     0 ?        S     三 09   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        I<    三 09   0:00 [rcu_gp]
root         4  0.0  0.0      0     0 ?        I<    三 09   0:00 [rcu_par_gp]
root         6  0.0  0.0      0     0 ?        I<    三 09   0:00 [kworker/0:0H
root         9  0.0  0.0      0     0 ?        I<    三 09   0:00 [mm_percpu_wq
root        10  0.0  0.0      0     0 ?        S     三 09   0:01 [ksoftirqd/0]
root        11  0.0  0.0      0     0 ?        I     三 09   0:05 [rcu_sched]
root        12  0.0  0.0      0     0 ?        S     三 09   0:01 [migration/0]
root        13  0.0  0.0      0     0 ?        S     三 09   0:00 [idle_inject/
root        14  0.0  0.0      0     0 ?        S     三 09   0:00 [cpuhp/0]
```

ps指令加上aux選項查看系統行程

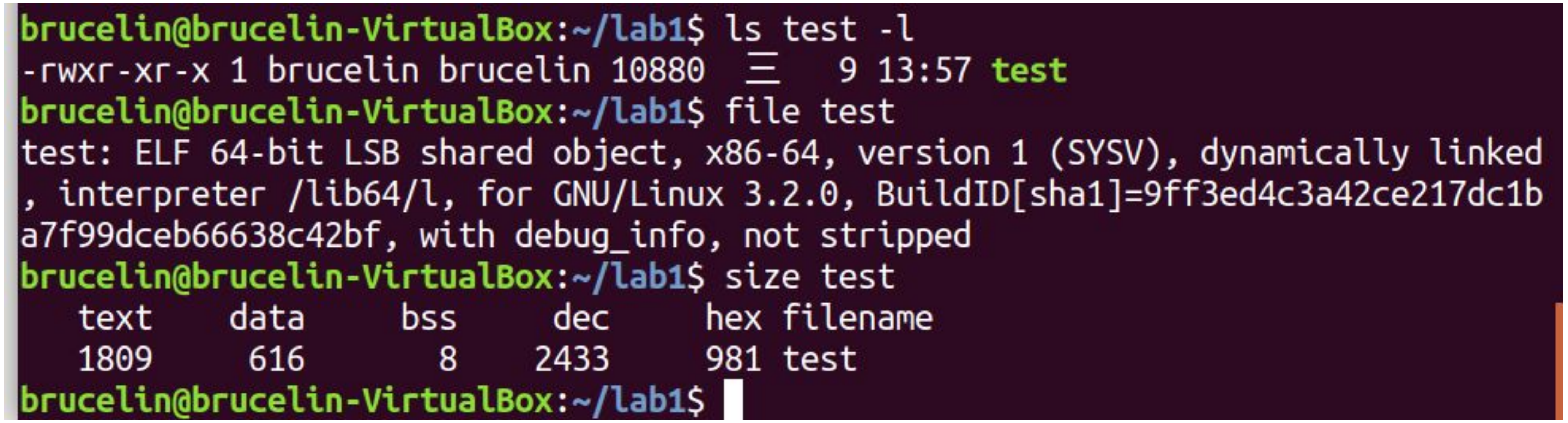
Process Management

- kill:能將目前運作的行程刪除，當kill指令送出訊號收到訊號的行程將依本身訊號值決定是否結束，能否結束還要看行程本身，若要強制結束可使用**-9**參數。

- killall:有時同一程式會同時執行好幾個行程，若使用kill一個一個刪實在太慢也太麻煩了，此時可以使用killall將相同名稱的行程一次刪除。
  語法:killall 行程名稱

Linux 可執行檔結構

- ELF是Linux下的可執行檔格式
- 未執行前的檔案包含文字區(text)、資料區(data)、未初始化資料區(bss)。
- 文字區 : 存放CPU執行的機器指令(machine instructions)、常數資料

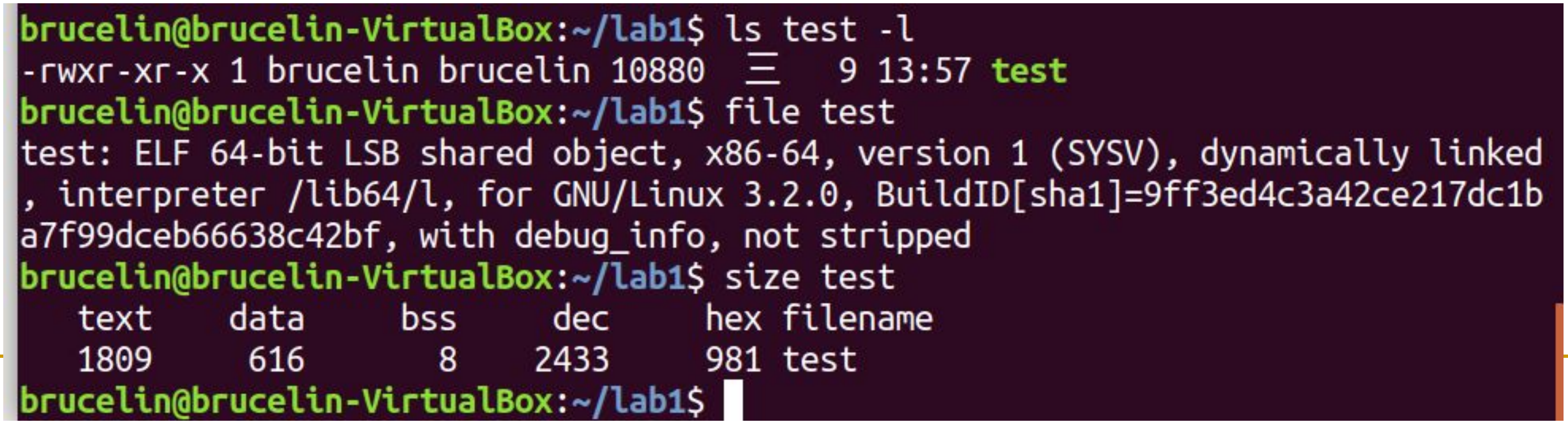


```
brucelin@brucelin-VirtualBox:~/lab1$ ls test -l
-rwxr-xr-x 1 brucelin brucelin 10880  三   9 13:57 test
brucelin@brucelin-VirtualBox:~/lab1$ file test
test: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked
, interpreter /lib64/l, for GNU/Linux 3.2.0, BuildID[sha1]=9ff3ed4c3a42ce217dc1b
a7f99dceb66638c42bf, with debug_info, not stripped
brucelin@brucelin-VirtualBox:~/lab1$ size test
   text    data     bss     dec     hex filename
   1809     616       8    2433     981 test
brucelin@brucelin-VirtualBox:~/lab1$ 
```

Linux 可執行檔結構

- 資料區又稱全域初始化資料區/靜態資料區(initialized data segment/static data segment):存放已初始化的全域變數與靜態變數(包含全域靜態變數與區域靜態變數)

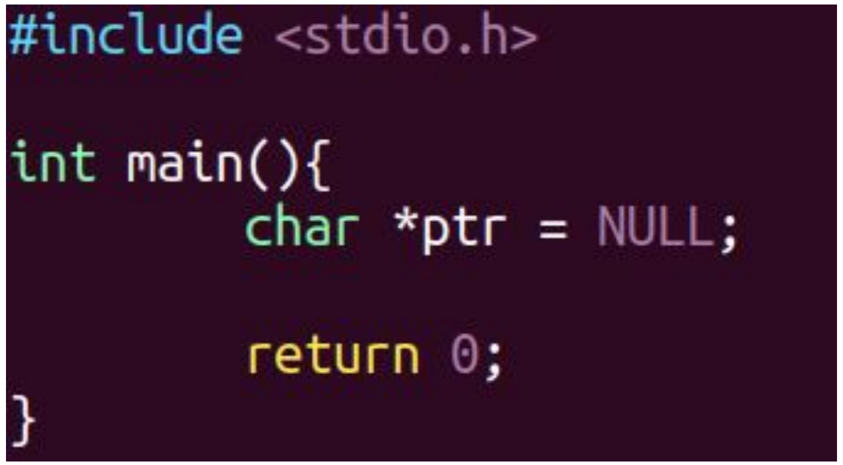
- 未初始化資料區(bss, Block Started by Symbol):存放未初始化的全域變數與的靜態變數。
  - BSS區的資料在程式開始之前被初始化為0或空指標(NULL)。



```
brucelin@brucelin-VirtualBox:~/lab1$ ls test -l
-rwxr-xr-x 1 brucelin brucelin 10880  三   9 13:57 test
brucelin@brucelin-VirtualBox:~/lab1$ file test
test: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked
, interpreter /lib64/l, for GNU/Linux 3.2.0, BuildID[sha1]=9ff3ed4c3a42ce217dc1b
a7f99dceb66638c42bf, with debug_info, not stripped
brucelin@brucelin-VirtualBox:~/lab1$ size test
   text    data     bss     dec     hex filename
   1809     616       8    2433     981 test
brucelin@brucelin-VirtualBox:~/lab1$ 
```

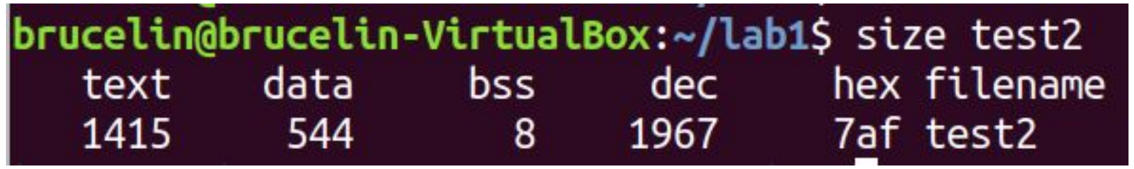
Example

- vim test2.c
- gcc test2.c -o test2

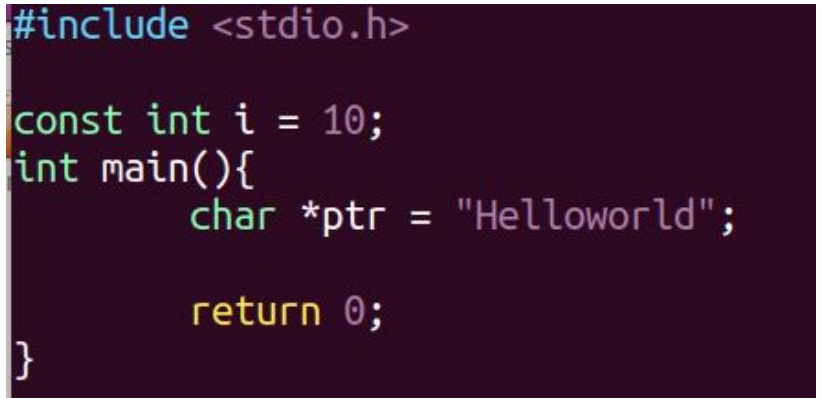


```
#include <stdio.h>

int main(){
        char *ptr = NULL;

        return 0;
}
```

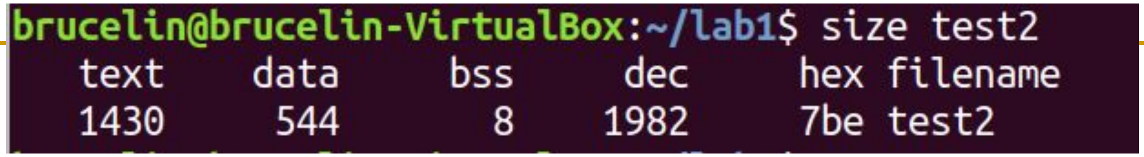
- size test2
- add const variable
- gcc test2.c -o test2
- size test2
- text區段增加15bytes，分別為
  - 4bytes const i
  - 11 bytes 常數字串



```
brucelin@brucelin-VirtualBox:~/lab1$ size test2
   text    data     bss     dec     hex filename
   1415     544       8    1967     7af test2
```



```
#include <stdio.h>

const int i = 10;
int main(){
        char *ptr = "Helloworld";

        return 0;
}
```

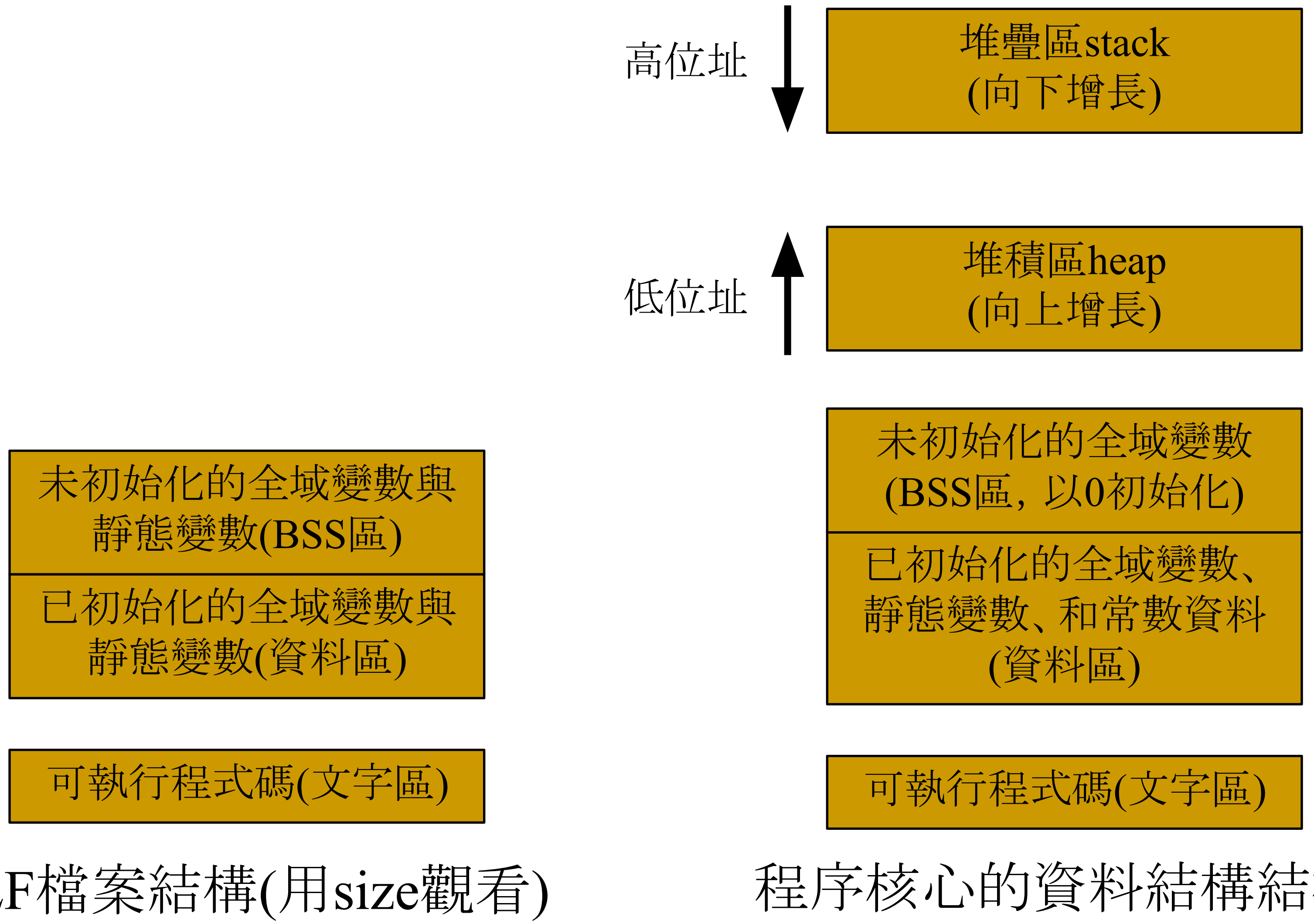


```
brucelin@brucelin-VirtualBox:~/lab1$ size test2
   text    data     bss     dec     hex filename
   1430     544       8    1982     7be test2
```

Linux 程序(Process)結構

- 在Linux系統下, 將ELF執行檔載入記憶體執行, 則將變成一個或多個程序(process)
- 產生多個程序的原因是載入的一個程序可以再建立其他新的程序。
- 所有的程序擁有各自獨立的環境與資源, 其環境是由目前的系統環境與其父程序(parent process)所組成

可執行檔與程序的儲存佈局



高位址
堆疊區stack (向下增長)
低位址
堆積區heap (向上增長)
未初始化的全域變數與靜態變數(BSS區)
已初始化的全域變數與靜態變數(資料區)
可執行程式碼(文字區)
未初始化的全域變數 (BSS區，以0初始化)
已初始化的全域變數、靜態變數、和常數資料 (資料區)
可執行程式碼(文字區)

ELF檔案結構(用size觀看)　　程序核心的資料結構結構

堆疊(stack)與堆積(heap)的區別

- 管理方式不同
  - 堆疊(stack)是由編譯器在程式執行時分配的空間，由作業系統維護(建立與釋放)。
  - 堆積(heap)則由malloc()函式分配的記憶體區塊，記憶體管理由程式人員手動控制，透過free()釋放。
- 空間大小不同
  - 堆疊向低位址擴展，為一連續的記憶體區域，堆疊的空間是固定的，一旦申請大小大於堆疊剩餘的空間，將出現堆疊溢位錯誤(stack overflow)。
  - 堆積向高位址擴展，他是不連續的記憶體區域。因為系統以鏈結表儲存空間記憶體位址，而鏈結表的讀取方向是從低位址倒高位址。

- 產生的碎片不同
  - 對於堆積而言, 頻繁的malloc/free(new/delete)會造成記憶體空間不連續, 進而導致大量的碎片。
  - 堆疊為連續的記憶體空間, 不會有碎片。
- 功能不同
  - 堆疊:存放區域變數(local variable)、函式參數(function/method parameter)、函數的返回位址(function/method return address)等資訊, 使運作過程可以順利取得所需的變數或函式所在地。
    - **可預測性外加後進先出的生存模式, 由系統自行 產生與回收其空間即可。**
  - 堆積:存放執行期間動態產生的資料, 由於為動態產生故結束點無法由系統來掌握, 故需使用者自行回收空間。

Stack overflow and heap overflow

- 當產生stack overflow一般是因為過多的函式呼叫(例如:遞迴太深)、或區域變數使用太多, 此時請試著將stack size調大一點, 另外檢查看看函式的呼叫跟變數的使用量。

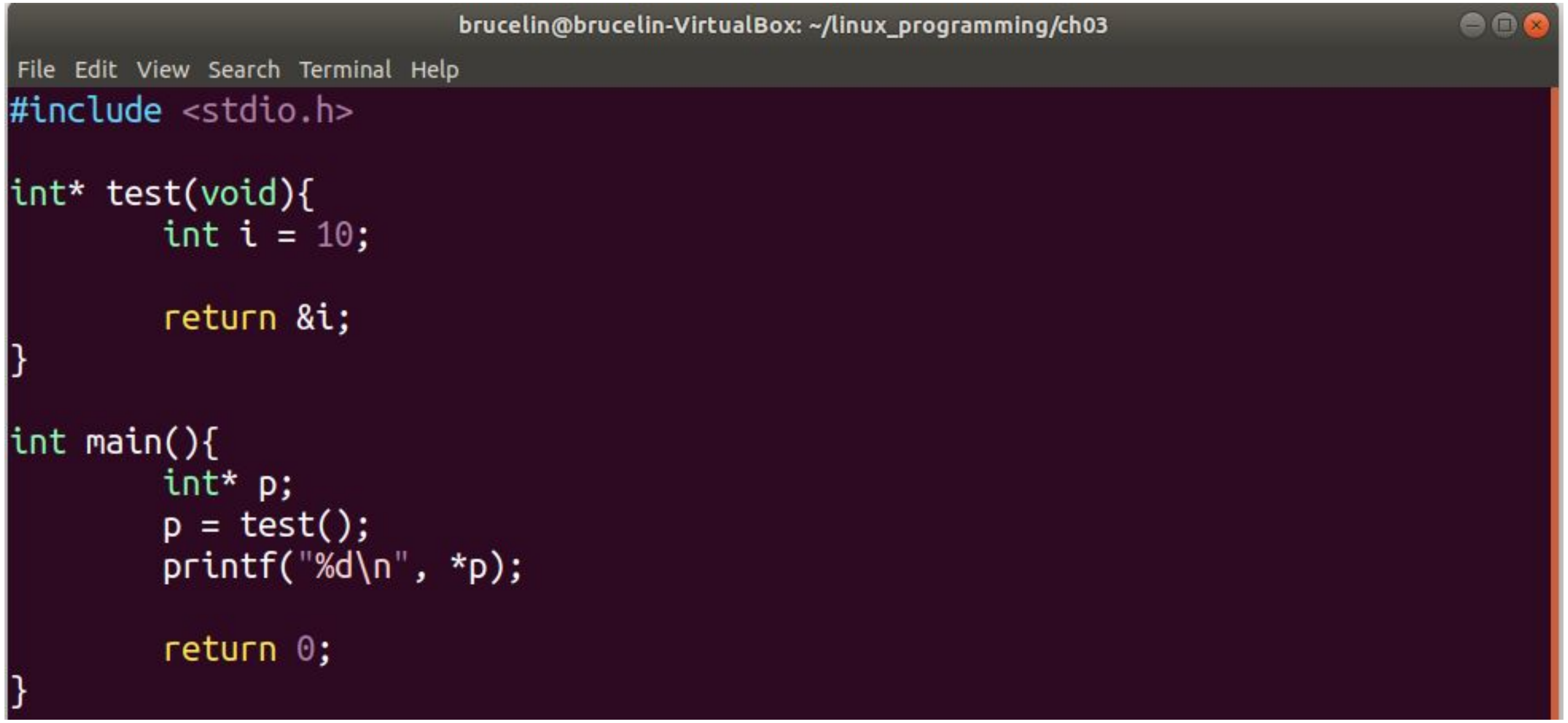
- 當發生heap overflow請檢查是否都有正確將heap space的資料回收(free/delete), 另外採行的動態配置是否合理, 不要過渡濫用而malloc(new)出無謂的空間。

Comparison of stack and heap

PARAMETER	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatic by compiler instructions.	Manual by programmer.
Cost	Less	More
Implementation	hard	Easy
Access time	faster	Slower
Main issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Flexibility	Fixed size	Resizing is possible

常見的記憶體錯誤

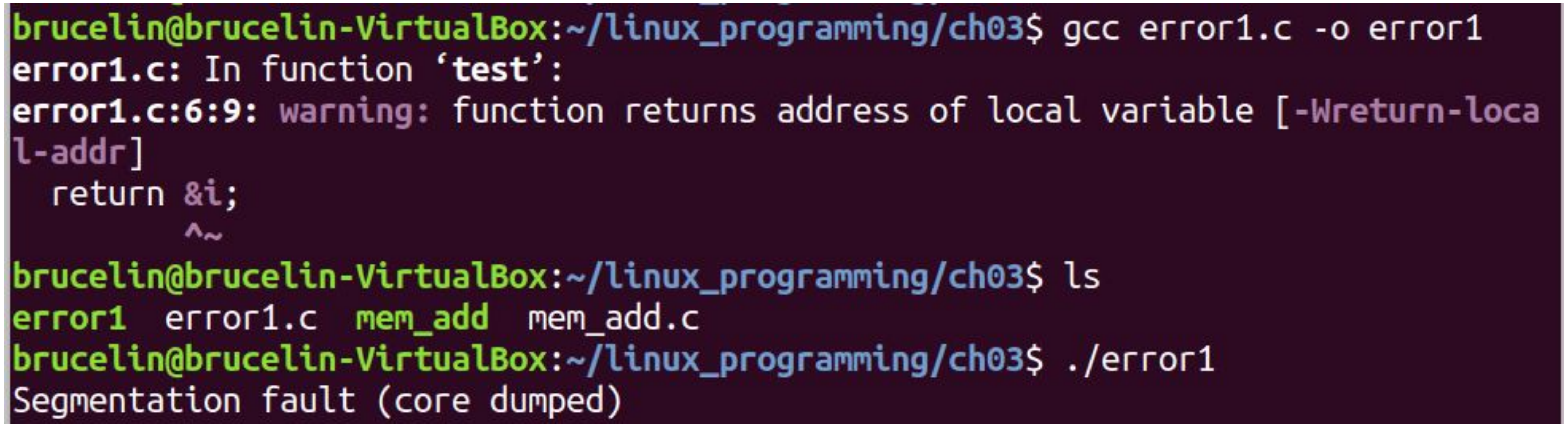
- 返回區域變數位址錯誤


brucelin@brucelin-VirtualBox: ~/linux_programming/ch03
File Edit View Search Terminal Help

```
#include <stdio.h>

int* test(void){
        int i = 10;

        return &i;
}

int main(){
        int* p;
        p = test();
        printf("%d\n", *p);

        return 0;
}
```

常見的記憶體錯誤

- 返回區域變數位址錯誤



```
brucelin@brucelin-VirtualBox:~/linux_programming/ch03$ gcc error1.c -o error1
error1.c: In function ‘test’:
error1.c:6:9: warning: function returns address of local variable [-Wreturn-loca
l-addr]
  return &i;
         ^~
brucelin@brucelin-VirtualBox:~/linux_programming/ch03$ ls
error1  error1.c  mem_add  mem_add.c
brucelin@brucelin-VirtualBox:~/linux_programming/ch03$ ./error1
Segmentation fault (core dumped)
```