Chapter 3: Processes-Concept

Prof. Li-Pin Chang

National Chiao Tung University

Chapter 3: Processes-Concept

- Process Concepts
- Process Scheduling
- Operations on Processes
- Inter-process Communication
- Examples of IPC Systems

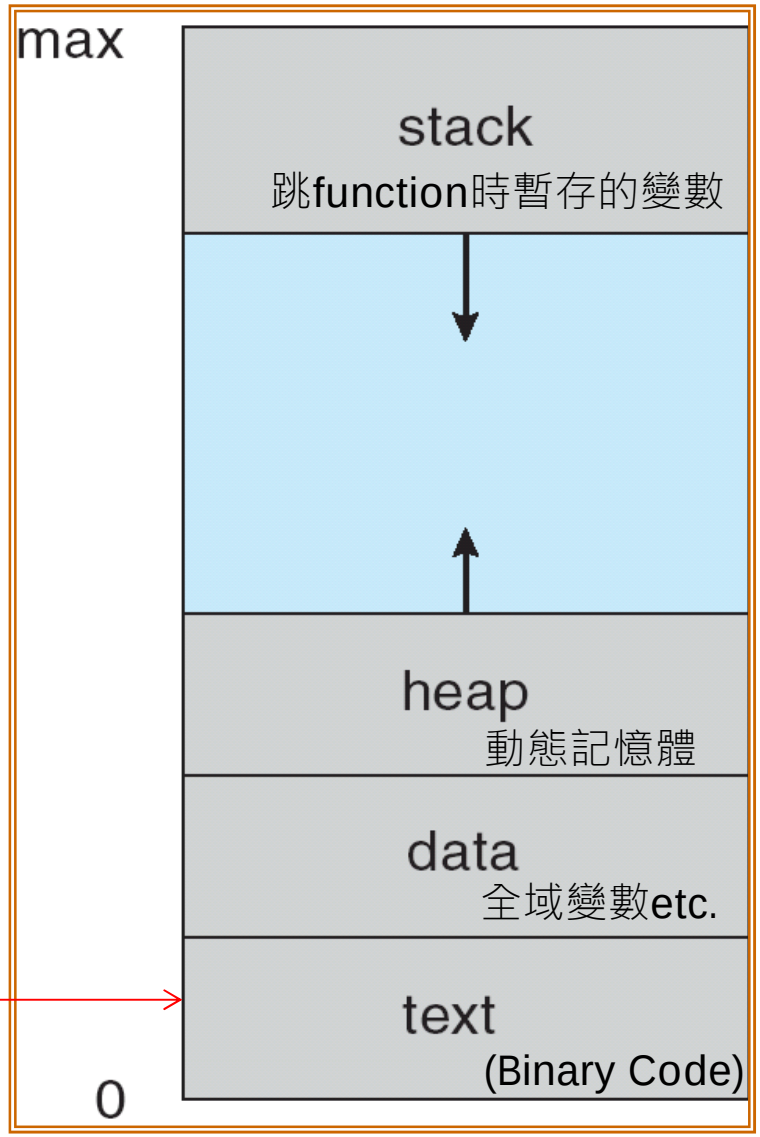
PROCESS CONCEPTS

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
  - Textbook uses the terms job, task, and process almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
  - Process: active, program: passive
- A process includes:
  - Text section
  - program counter
  - stack
  - data section
    - BSS and variables with initial values
  - Heap

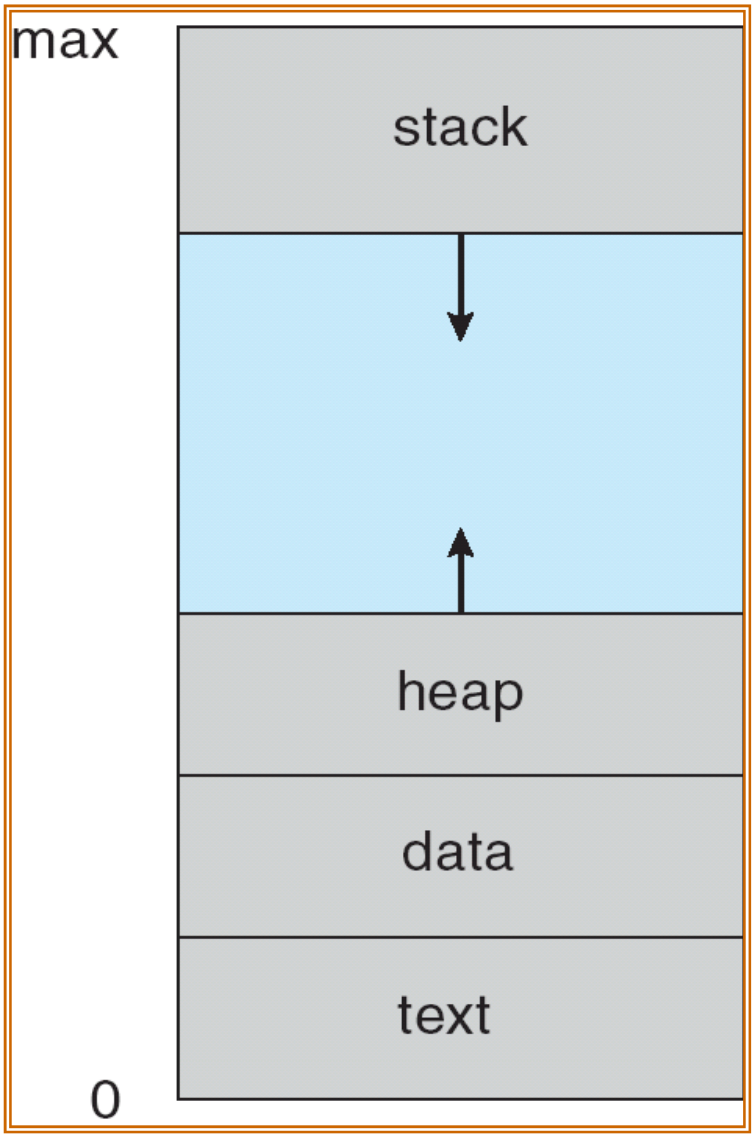
Process in Memory

邏輯上的記憶體空間



Where the variables below are allocated from?

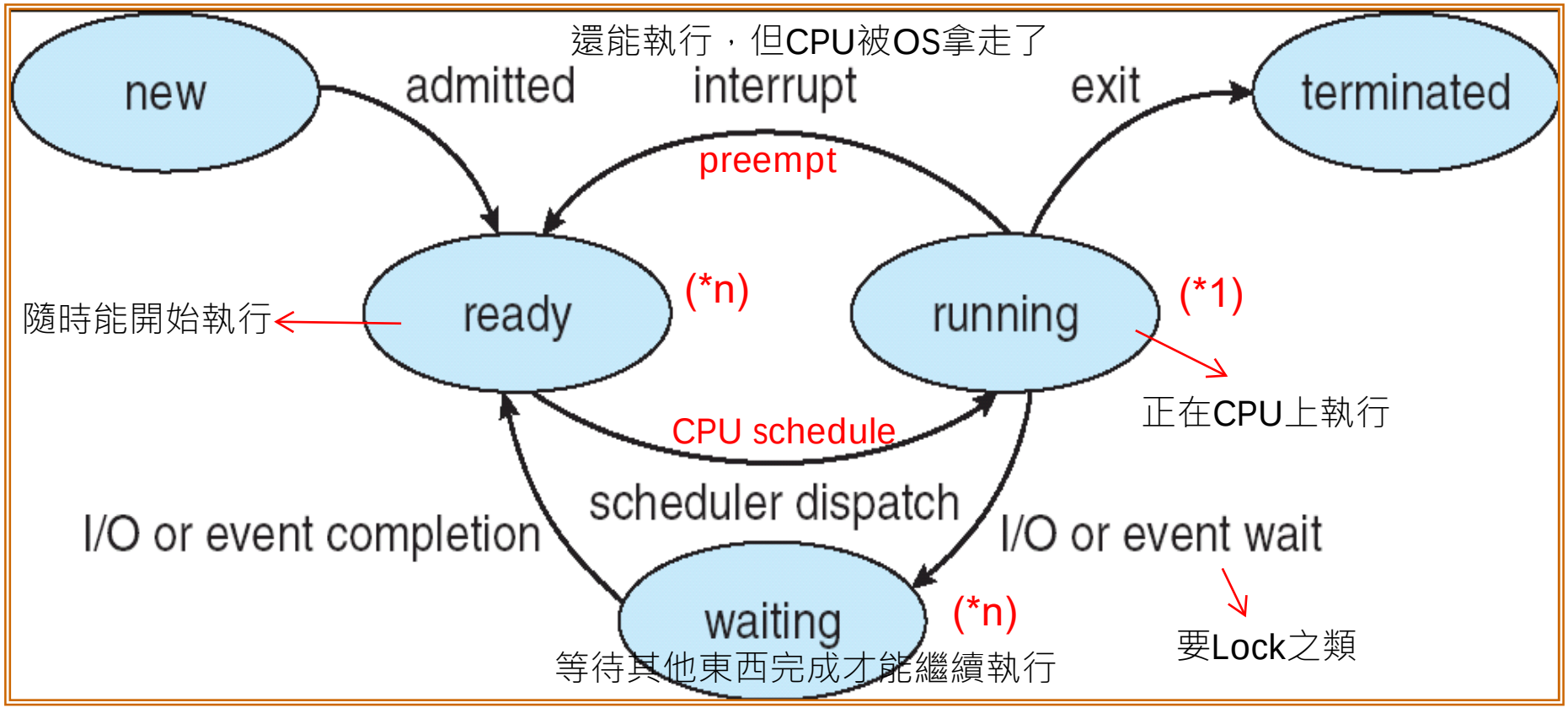
```
int i; (data)

                (stack)
int foo(int x)
{               (stack)
        int *y;

        i=0;
        y=(int *)malloc(100);
}                   (heap)
```



Process State

- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - ready: The process is waiting to be assigned to a processor
  - waiting: The process is waiting for some event to occur
  - terminated: The process has finished execution

Diagram of Process State



注意：waiting中的process不能直接跳回running

- A running process voluntarily leaves the running state
- Running → waiting: the running process requests a system service that can not be immediately fulfilled
  - Triggered by traps (calling the kernel for I/O service)

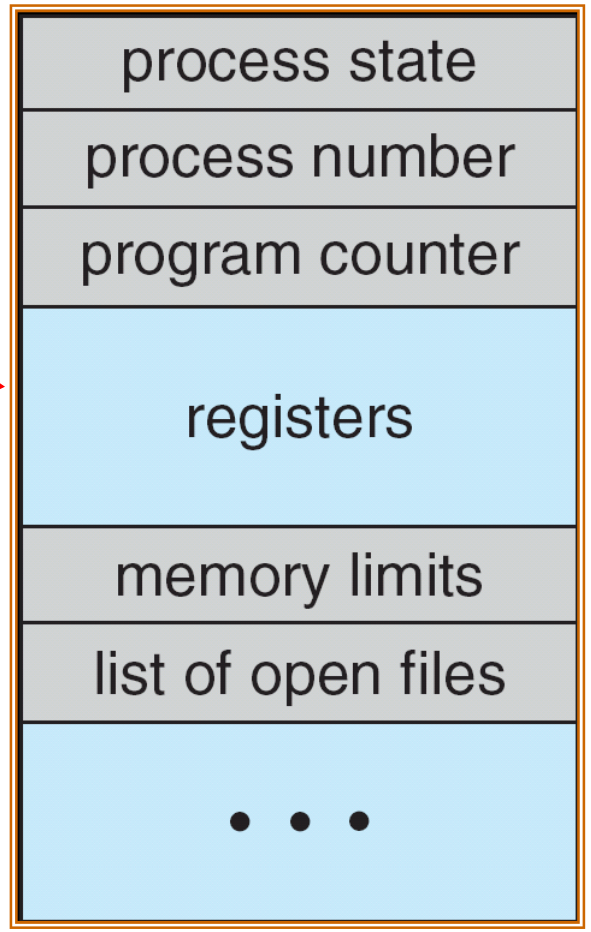
- When a running process involuntarily leaves the running state
- Running → ready: the running process runs out its time quota under time sharing.
  - Triggered by timer interrupts
- Running → ready, case 2: IO interrupts make a high-priority process ready and the running process is preempted by the high-priority process
  - Triggered by I/O interrupts

- Hardware interrupts can trigger which one(s) of the following transitions?
  1. Running → ready 由於timer、被high priority process搶走CPU
  2. Running → waiting
  3. Waiting → ready

- What is the state transition of
  - Starting an synchronous I/O
  - Process resume
  - Process suspend?

Process Control Block (PCB)

- Information associated with each process
  - Process state
  - CPU registers values
  - CPU scheduling info (e.g., priority)
  - Memory-management information (e.g., segment table and page-table base register)
  - Accounting info (e.g., disk quota…)
  - I/O status info (e.g., opened files)



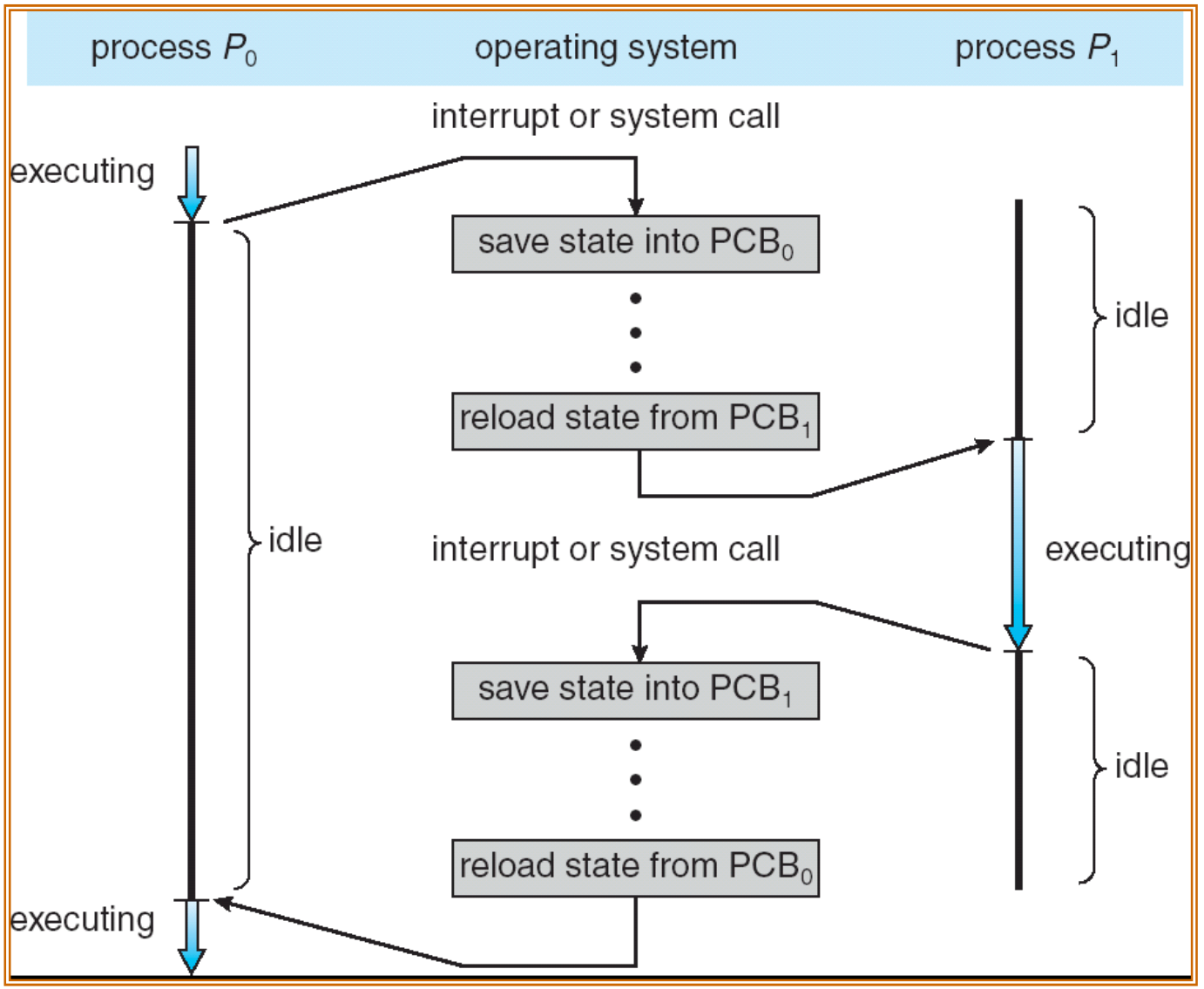
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is an overhead; the system does no useful work while switching
- Time dependent on hardware
  - Roughly 2000 ns/cxtsw on Intel 5150 (2.66 GHz)
  - And the subsequent costs of pipeline stall and cache pollution

還不算pipe line和cache的損失

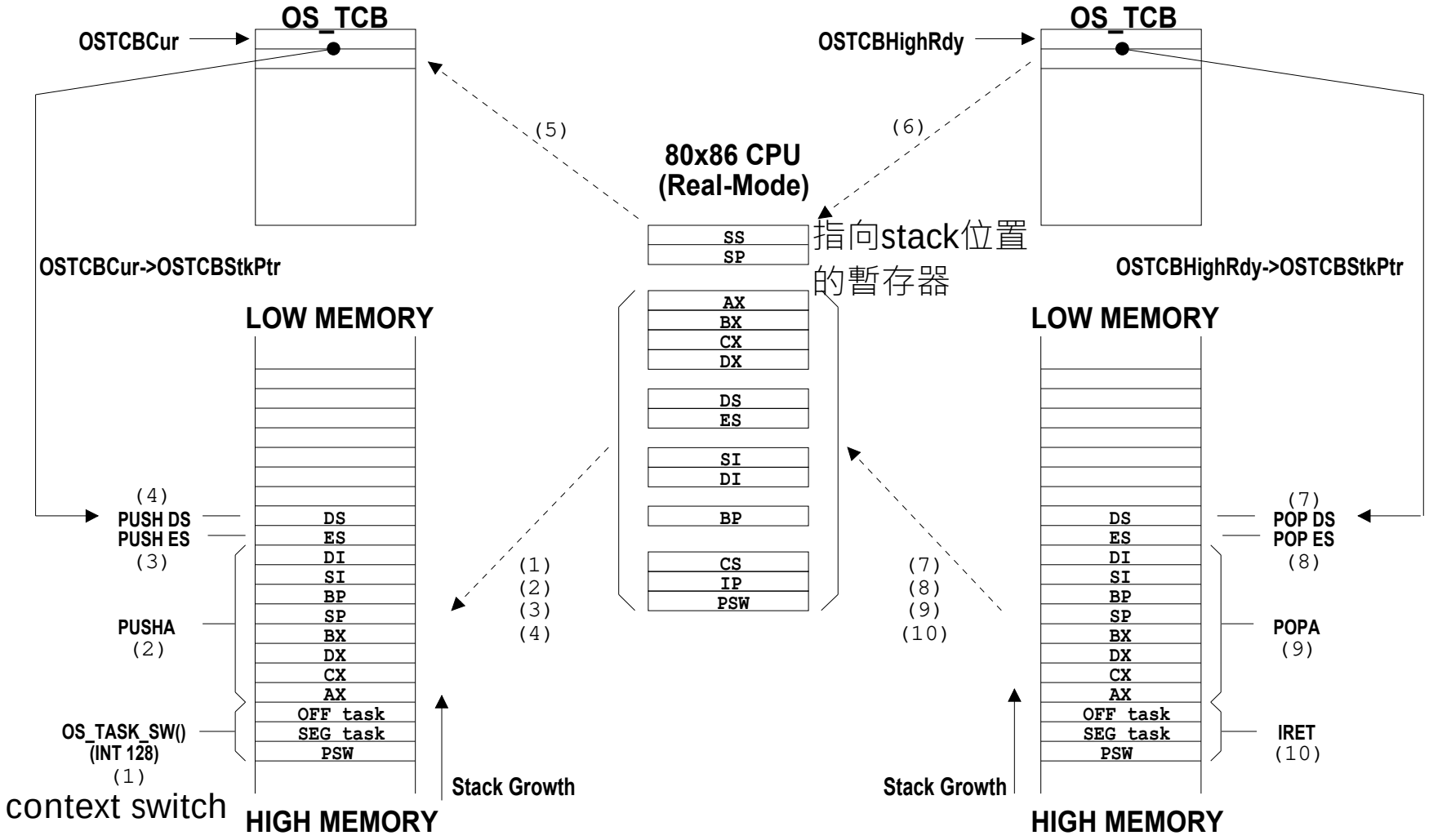
http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

CPU Switch From Process to Process



Example: Context Switch in uC/OS-2

```
;*********************************************************************************************************
*
;                                PERFORM A CONTEXT SWITCH (From task level)
;                                          void OSCtxSw(void)
;
; Note(s): 1) Upon entry,
;             OSTCBCur     points to the OS_TCB of the task to suspend
;             OSTCBHighRdy points to the OS_TCB of the task to resume
;
;          2) The stack frame of the task to suspend looks as follows:
;
;                SP -> OFFSET  of task to suspend    (Low memory)
;                      SEGMENT of task to suspend
;                      PSW     of task to suspend    (High memory)
;
;          3) The stack frame of the task to resume looks as follows:
;
;                OSTCBHighRdy->OSTCBStkPtr --> DS                              (Low memory)
;                                              ES
;                                              DI
;                                              SI
;                                              BP
;                                              SP
;                                              BX
;                                              DX
;                                              CX
;                                              AX
;                                              OFFSET  of task code address
;                                              SEGMENT of task code address
;                                              Flags to load in PSW             (High memory)
;*********************************************************************************************************
*
```


OS_TCB
OSTCBCur
OSTCBCur->OSTCBStkPtr
LOW MEMORY
(4)
PUSH DS
PUSH ES
(3)
PUSHA
(2)
OS_TASK_SW()
(INT 128)
(1)
context switch
DS
ES
DI
SI
BP
SP
BX
DX
CX
AX
OFF task
SEG task
PSW
HIGH MEMORY
Stack Growth
(5)
(1)
(2)
(3)
(4)
80x86 CPU
(Real-Mode)
SS
SP
指向stack位置
的暫存器
AX
BX
CX
DX
DS
ES
SI
DI
BP
CS
IP
PSW
(6)
(7)
(8)
(9)
(10)
OS_TCB
OSTCBHighRdy
OSTCBHighRdy->OSTCBStkPtr
LOW MEMORY
(7)
POP DS
POP ES
(8)
POPA
(9)
IRET
(10)
DS
ES
DI
SI
BP
SP
BX
DX
CX
AX
OFF task
SEG task
PSW
Stack Growth
HIGH MEMORY

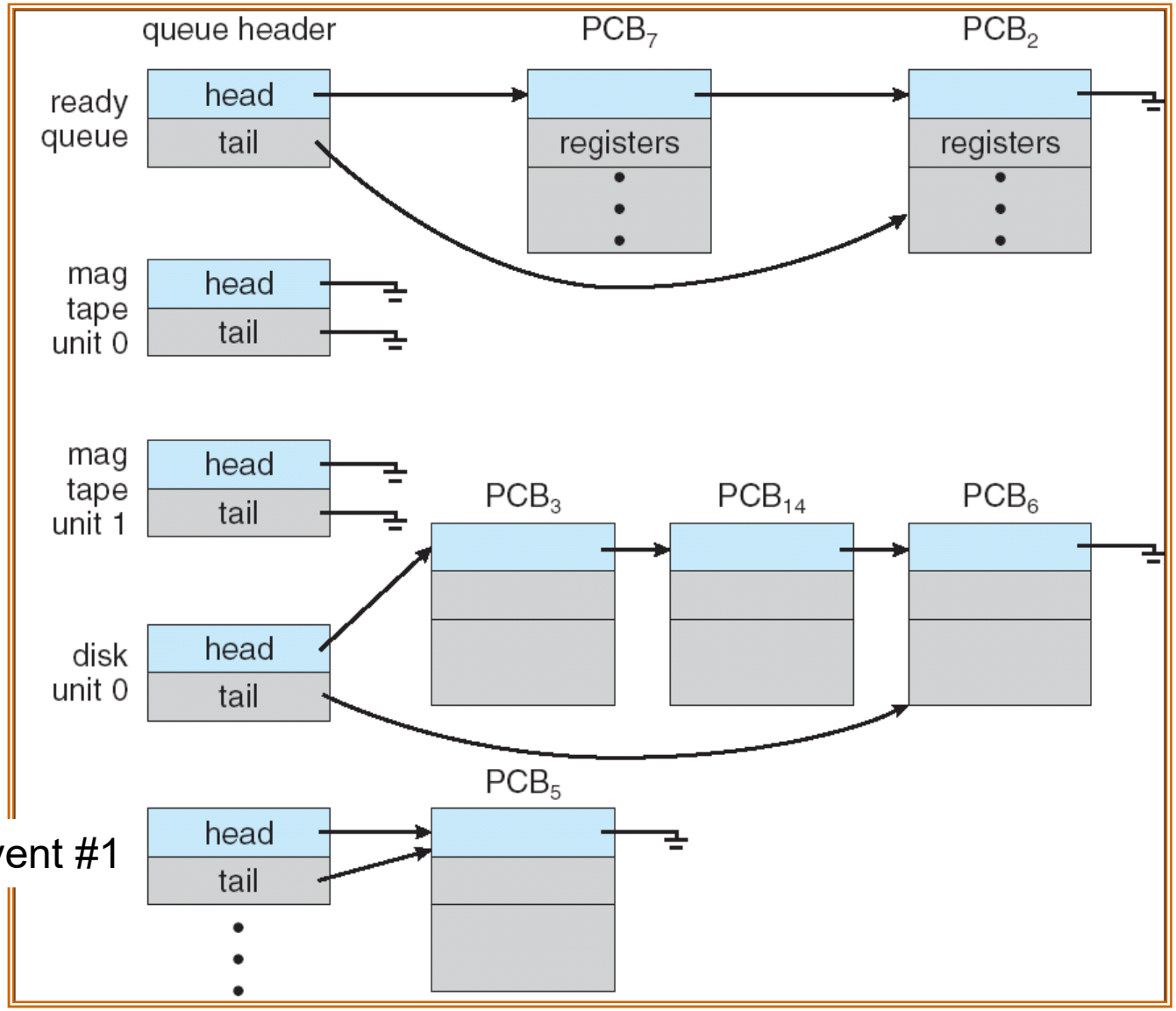
```
_OSCtxSw    PROC   FAR
;
            PUSHA                                   ; Save current task's context
            PUSH   ES                               ;
            PUSH   DS                               ;
;
            MOV    AX, SEG _OSTCBCur                ; Reload DS in case it was altered
            MOV    DS, AX                           ;
;
            LES    BX, DWORD PTR DS:_OSTCBCur       ; OSTCBCur->OSTCBStkPtr = SS:SP
            MOV    ES:[BX+2], SS                    ;
            MOV    ES:[BX+0], SP                    ;
;
            CALL   FAR PTR _OSTaskSwHook            ; Call user defined task switch hook
;
            MOV    AX, WORD PTR DS:_OSTCBHighRdy+2  ; OSTCBCur = OSTCBHighRdy
            MOV    DX, WORD PTR DS:_OSTCBHighRdy    ;
            MOV    WORD PTR DS:_OSTCBCur+2, AX      ;
            MOV    WORD PTR DS:_OSTCBCur, DX        ;
;
            MOV    AL, BYTE PTR DS:_OSPrioHighRdy   ; OSPrioCur = OSPrioHighRdy
            MOV    BYTE PTR DS:_OSPrioCur, AL       ;
;
            LES    BX, DWORD PTR DS:_OSTCBHighRdy   ; SS:SP = OSTCBHighRdy->OSTCBStkPtr
            MOV    SS, ES:[BX+2]                    ;
            MOV    SP, ES:[BX]                      ;
;
            POP    DS                               ; Load new task's context
            POP    ES                               ;
            POPA                                    ;
;
            IRET                                    ; Return to new task
;
_OSCtxSw    ENDP
```

PROCESS SCHEDULING

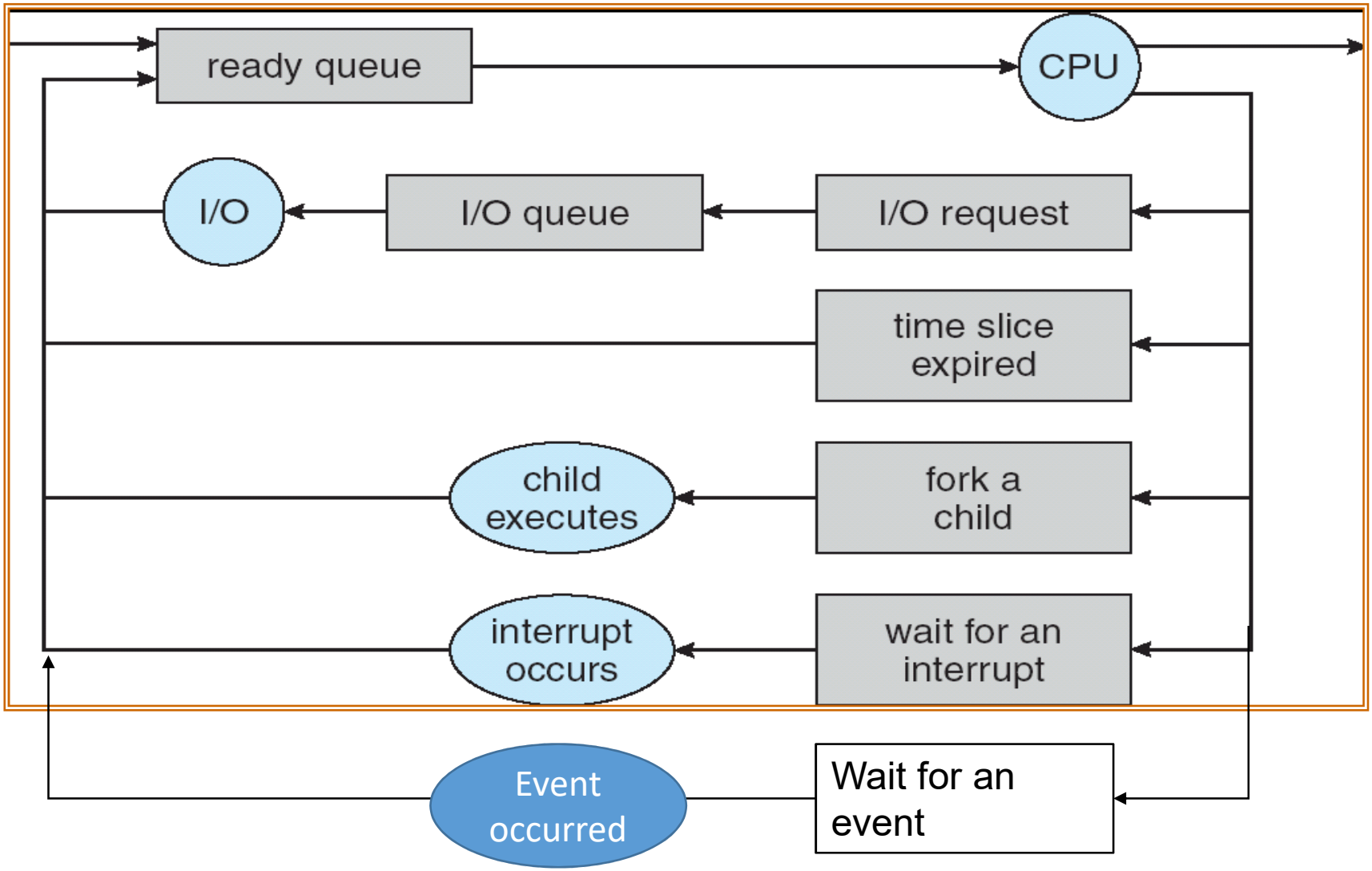
Process Scheduling Queues

- Ready queue – set of all processes residing in main memory, ready for execution
- Device queues – set of processes waiting for an I/O device
- Event queues – set of processes waiting for an event (e.g., semaphore)
- Processes migrate among the various queues

Various Process Queues



Representation of Process Scheduling



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

Processes' expire time is 5 mm

Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast) 通常要O(1) or O(logN)，linear time太慢
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow) 要以適合的比例將I/O-bound、CPU-bound 的processes混合來執行

主要用在server

- The long-term scheduler controls the degree of multiprogramming
- Processes can be described as either: 大部分時間都在等I/O完成 ex.editor, downlaoder
  - 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 大部分時間都花在CPU運算上 ex. 檔案轉換

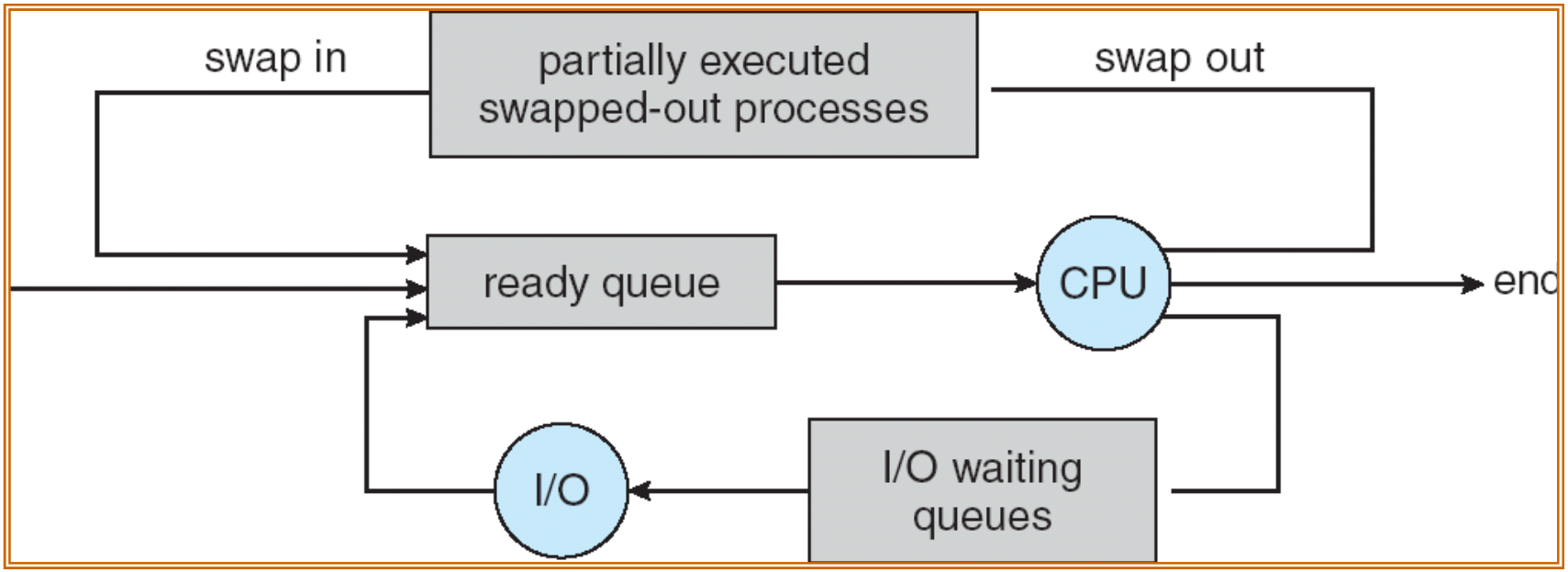
Long-Term Scheduler: Now and Then

- In batch-processing systems, the long-term scheduler is to make a good mix of I/O bound processes and CPU-bound processes
- Timesharing systems do not have long-term schedulers
  - The degree of multiprogramming is limited by physical limitation (e.g., RAM space)
  - The user will give up if the system cannot launch any more processes

目的在於空出記憶體

Addition of Medium Term Scheduling

把太久沒用的processes寫入
到磁碟釋放記憶體



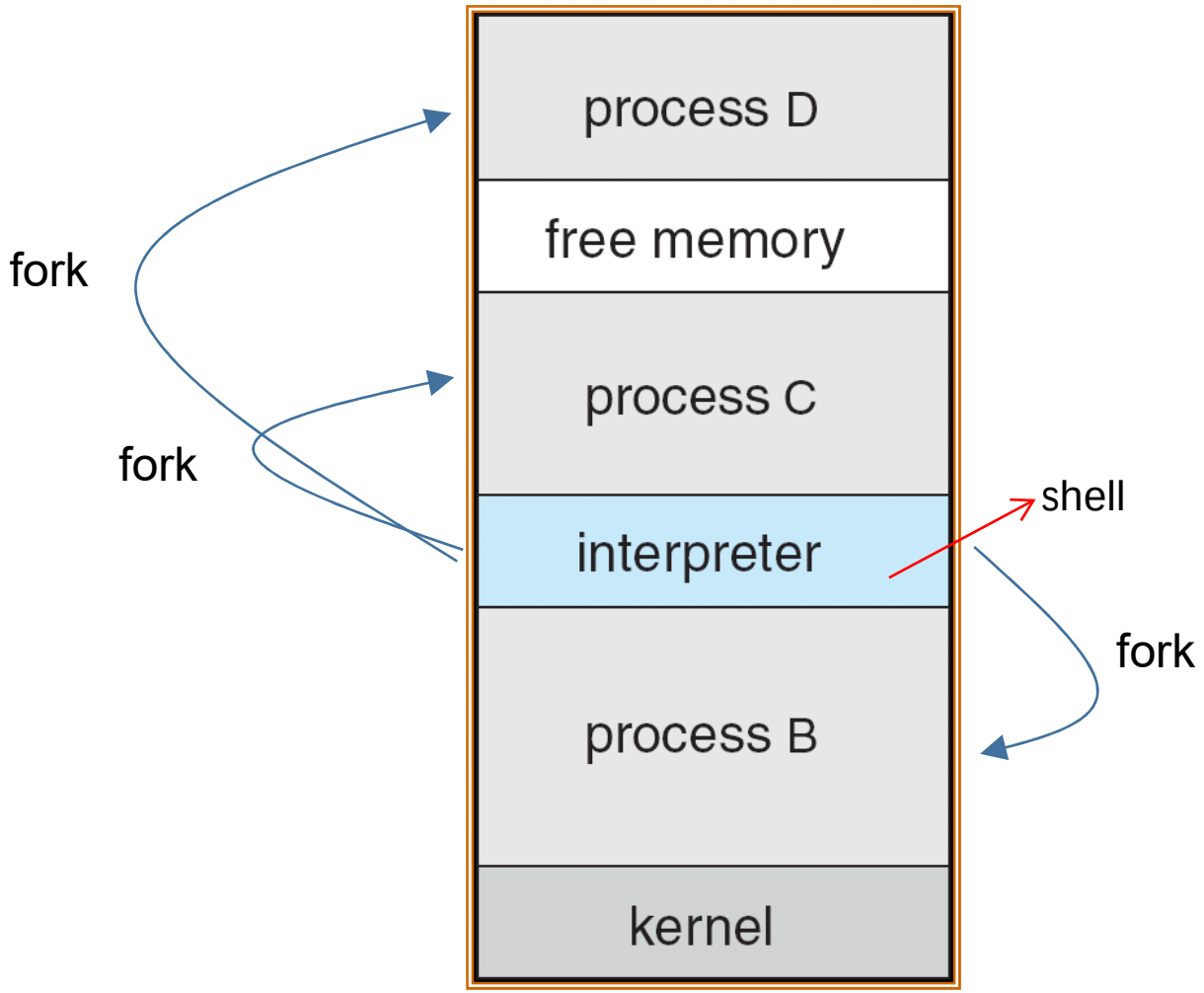
Swapping out: “saving” the memory image of a process to give memory space to new processes

Checklist

- Long term scheduler
- Short term scheduler
- Mid term scheduler
- I/O-bound and CPU-bound processes

OPERATIONS ON PROCESSES (CREATION & TERMINATION)

FreeBSD Program Execution (Physical View, No Paging)



Process Creation

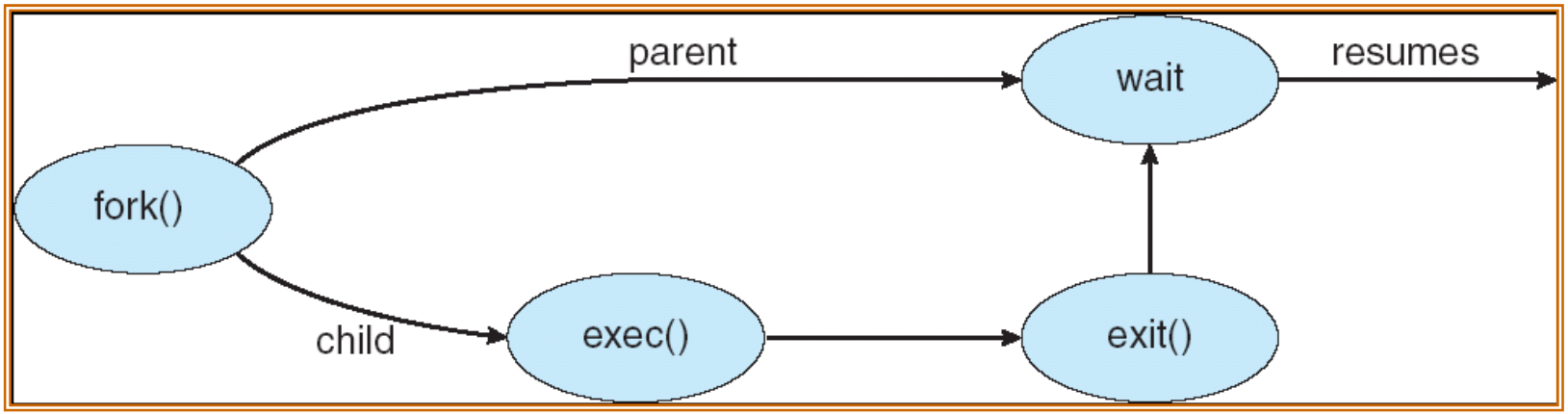
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute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

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process
  - exec system call used after a fork to replace the process' memory space with a new program

- Right after fork():
  - The child is an exact copy of the parent

Process Creation



C Program Forking Separate Process (in UNIX)

Parent and child see different return values!!!

```
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 */
            /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
   }
}
```

The child won't return here after exec()

The parent has child's pid so it can kill the child (if necessary)

Address Spaces of Parent and Child Processes

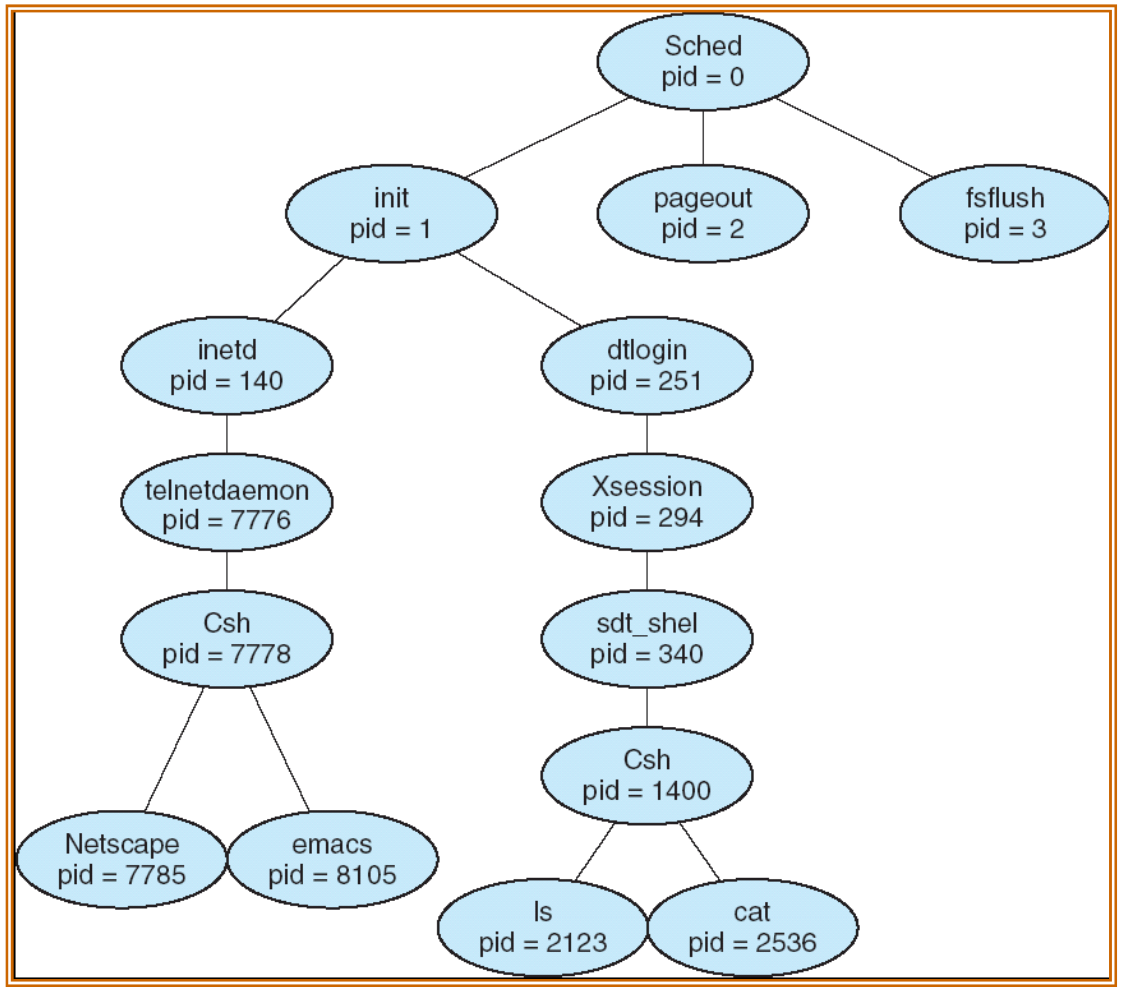
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int x=0;

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 */
                x++;
                exit(0);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("%d",x);
                exit(0);
        }
}
```

What is the output of this program?

parents 最後會印出0，因為child process是對"自己"的x加一

fork()會完全"複製"出一份process(所有資料，包括全域變數)

A tree of processes on a typical Solaris



Process Termination

- Process executes last statement and asks the operating system to delete it (exit) → C compiler會在main的右括號插入一個exit()
  - Output data from child to parent (via wait)
  - Process' resources are de-allocated by operating system
  - Synchronous termination 自行要求的termination
- Parent may terminate execution of its children (abort, kill)
  - Asynchronous termination
- If parent is exiting before children
  - Child processes become orphan proesses, and will be adopted by the grand-parent process or the "init" process

Orphan Processes and Zombie Processes

- An orphan process
  - A process whose parent process has been terminated
  - Will be adopted by process 0
  - Process 0 will wait (retrieve the return value) of an orphan
- A zombie (defunct) process 與orphan 不必然有關係
  - A process that has terminated but is not yet removed from the process table
  - Return value is not read by its parent

A Zombie Child Process

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

main(){

        if(fork()==0){
                // child process
                printf("child pid=%d\n", getpid());
                exit(0)
        }

        // parent process
        sleep(20);    // let the child print the message
        printf("parent pid=%d \n", getpid());
        exit(0);
}
```

After the child terminates, it becomes a zombie until being adopted and read by init.

vfork(): parent and child share most resources

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int x=0;

int main()
{
        pid_t  pid;
        /* fork another process */
        pid = vfork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                x++;
                _exit(0);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("%d",x);
                exit(0);
        }
}
```

直接寄生在parent的stack上，exec會直接另外找一塊

省去了copy一份的花費

What is the output of this program?

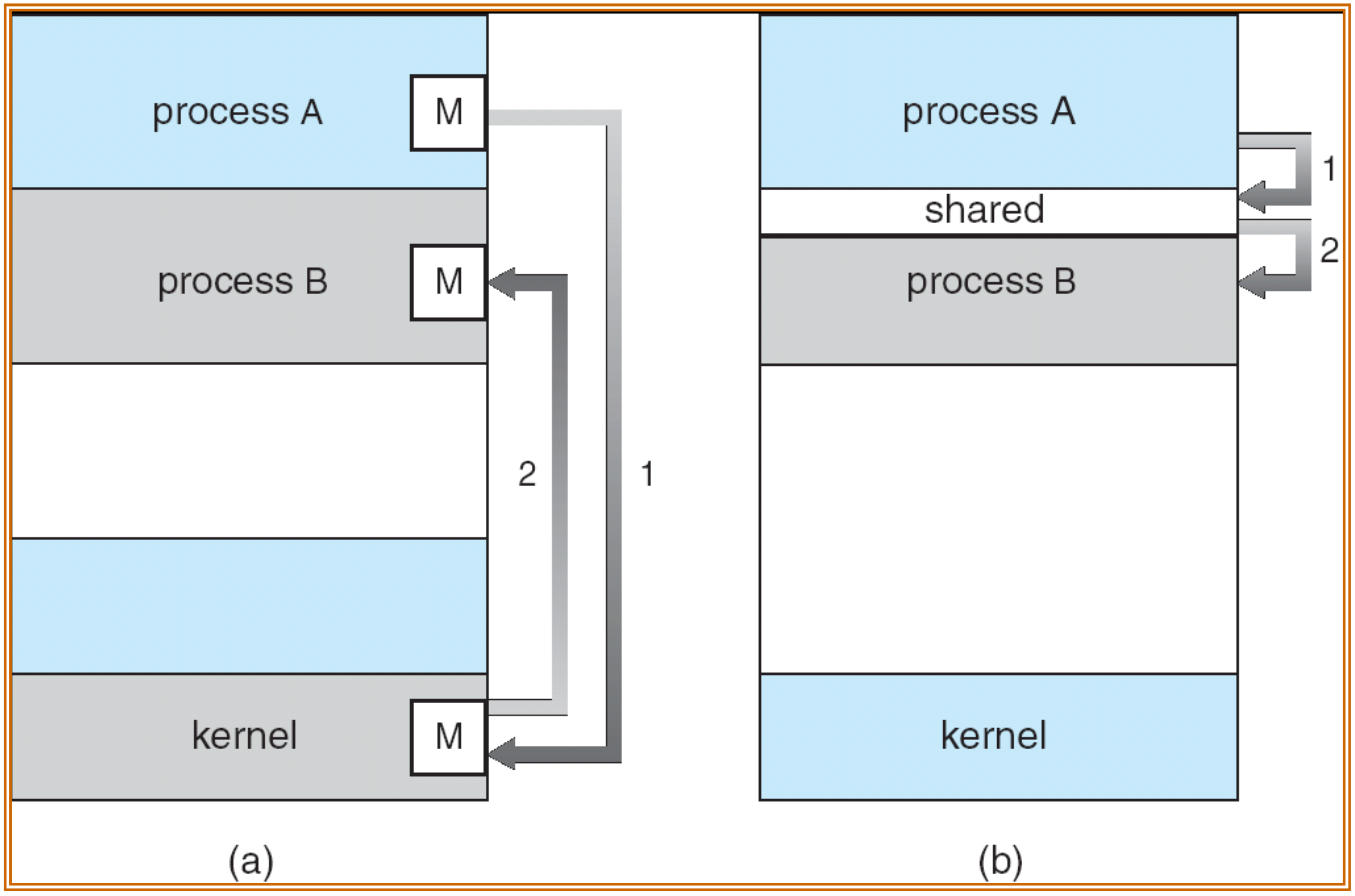
INTER-PROCESS COMMUNICATION

Cooperating Processes

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up (in SMT and/or SMP environments)
  - Modularity
  - Convenience

- In practice the most reason to use multiple processes is to handle events or to do things in background. For example:
  - To perform parallel computation
  - To handle user input while computation is being carried out
  - To defrag disk fragmentation in background

Communications Models



Message passing

有同步效果，對pipe處理時在一些情況下process會等在那
但成本還蠻高的

Shared memory

通常會在需要大量資料通訊時用
(可以直接顯示，很方便)

IPC- SHARED MEMORY

Shared Memory

- Linux offers the following system calls for shared memory management
  - shmget() – create a block of shared memory
  - shmat() – attach shared memory to the current process's address space 把shared momory嵌入到自己的記憶體空間(有poniter指向那塊physical memory
  - shmdt() – detach shared memory from the current process's address space
  - shmctl() – control shared memory
- Let assume that a piece of shared memory has been setup between two processes

Producer-Consumer Problem

- Paradigm for cooperating processes, a producer process produces information that is consumed by a consumer process
  - The two processes run concurrently
- Objective:
  - to synchronize a producer and a consumer via shared memory
- Issues:
  - The buffer size is limited – no overwriting and null reading

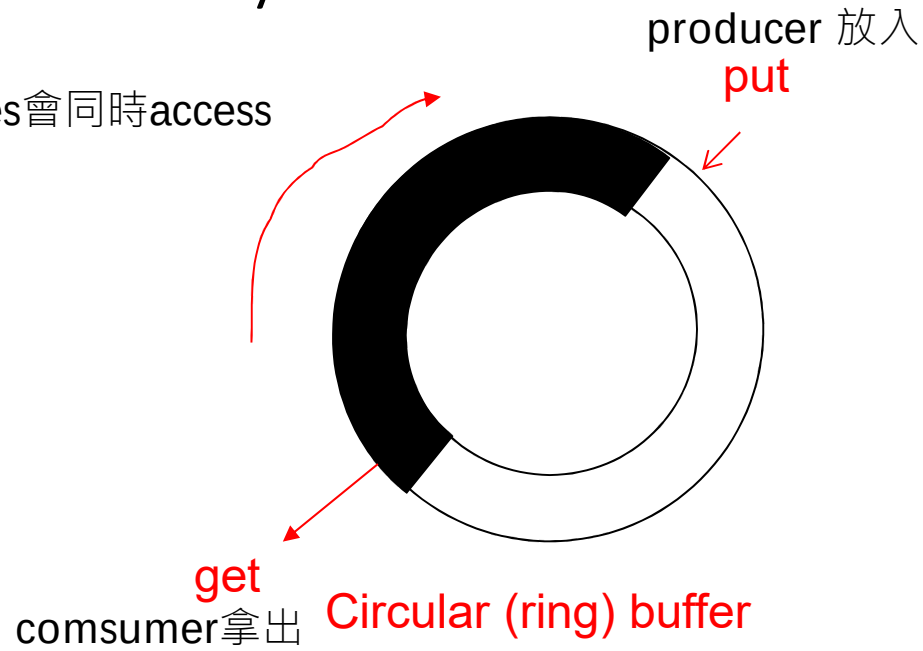
Lock-free的做法

Bounded-Buffer – Shared-Memory Solution

不可能用counter
→可能兩隻processes會同時access
→出現錯誤

- Shared data

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

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



- Solution is correct, but can only use BUFFER_SIZE-1 elements
- What are the conditions for buffer full and buffer empty?

put+1 == get　　put==get

Bounded-Buffer – Insert() Method

```
while (true) {

     /* Produce an item */
     while ((( (in + 1) % BUFFER SIZE count)  == out)
          ;   /* do nothing -- no free buffers */

     buffer[in] = item;
     in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Remove() Method

```
while (true) {

      while (in == out)
         ; // do nothing -- nothing to consume

      // remove an item from the buffer
      item = buffer[out];
      out = (out + 1) % BUFFER SIZE;
      return item;
}
```

Bounded Buffer Problem

- Data corruption?
- Performance issue?
- Why not to use a free-slot counter?
- How about multiple producers or consumers?

兩個Device溝通的時候會用這個
因為通常這兩是平行執行，不會互相干擾
→較沒有效能問題

IPC- MESSAGE PASSING

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message) – message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Example: Linux Pipe

- A basic mechanism for IPC
  - Widely used, e.g., “ls | more”
  - A process “ls”, a process “more”, and a pipe between them
- The system call pipe() creates a pipe
  - Receiver must close the output side, and receives from the input side
  - Sender must close the input side, and write to the output side
  - A pipe is created and configured by the parent process

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
```

```
        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

UNIX Signals

對象是process
中斷的對象是CPU

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled
- Analogy
  - Interrupts for CPU (async or sync)
  - Signals for processes (async or sync)
  - Interrupts and signals are not necessarily 1-1 mapping

Signal Handling C compiler會幫你加(compiler萬歲!)

- Synchronous signals
  - A signal that is delivered to the process caused the event
  - E.g., divide overflow and memory-access violations
- Asynchronous signals
  - A signal that is delivered to a process other than the signaling process
  - E.g., the kill signal
- Signal handlers
  - Default handlers
  - User-defined handlers (using signal() or sigaction())

UNIX Signal Example

- Synchronous signals
  - SIGSEGV : Memory protection fault
  - SIGFPE : Arithmetic fault, including divided by zero
- Asynchronous signals
  - SIGKILL : Kill a process
  - SIGSTOP : Suspend a process
  - SIGCHLD: ??? 跟zombie處理有關係

Handling SIGSEGV on your own

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void sigsegv_handler(int sig) {
        printf("Received segmentation violation (SIGSEGV). \n");
        exit(0);
}

int main() {
        int *null_pointer=(int *)NULL;
        signal(SIGSEGV,sigsegv_handler);

        printf("About to segfault:\n");
        *null_pointer=0;

        printf("Shouldn't be here!\n");
        return 1;
}
```

如果沒加這行會變無窮迴圈，因為handler跑完會跳回去原來的非法程式碼

Handling SIGSEGV on your own

```
void action(int sig, siginfo_t* siginfo, void* context)
{
    sig=sig; siginfo=siginfo;

    // get execution context
    mcontext_t* mcontext = &((ucontext_t*)context)->uc_mcontext;

    uint8_t* code = (uint8_t*)mcontext->gregs[REG_EIP];
    if (code[0] == 0x88 && code[1] == 0x10) { // mov %dl,(%eax)
        mcontext->gregs[REG_EIP] += 2; // skip it!
        return;
    }
}
main()
{
    ...
    sigaction(SIGSEGV, ...);
    ...
    for (int i = 0; i < 10; i++) { ((unsigned char*)0)[i] = i; }
}
```


http://stackoverflow.com/questions/6981702/how-to-write-a-segmentation-fault-handler-so-that-the-faulty-instruction-is-not

End of Chapter 3