

Operating System Homework 3

繳交期限: 4/16 17:00 前繳至 ED817

手寫題：

- 3.5 Describe the actions taken by a kernel to context-switch between processes.
- 3.9 Describe the differences among short-term, medium-term, and long-term scheduling.
- 3.10 Including the initial parent process, how many processes are created by the program shown in Figure 3.28?

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

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.28 How many processes are created?

- 3.11 Using the program in Figure 3.29, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

```

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

int main()
{
    pid_t pid, pid1;

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

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        pid1 = getpid();
        printf("child: pid = %d",pid); /* A */
        printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
        pid1 = getpid();
        printf("parent: pid = %d",pid); /* C */
        printf("parent: pid1 = %d",pid1); /* D */
        wait(NULL);
    }

    return 0;
}

```

Figure 3.29 What are the pid values?

- 3.12 Using the program shown in Figure 3.30, explain what the output will be at Line A.

```

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

int value = 5;

int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE A */
        return 0;
    }
}

```

Figure 3.30 What output will be at Line A?

程式題：

- 3.13 The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, Formally, it can be expressed as:

$$\begin{aligned} fib_0 &= 0 \\ fib_1 &= 1 \\ fib_n &= fib_{n-1} + fib_{n-2} \end{aligned}$$

Write a C program using the `fork()` system call that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line. For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child

process. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

Input:

請於 **command line** 上輸入參數 – Number of sequence

1. Number of sequence: 指定輸出前幾項數列
2. 測試時會給予不同參數，必須有 **error checking** 機制，避免收取不合法參數
3. 測試時，不會要求輸出超過 30 項

Output:

在 process 起始與結束分別印出相關資訊，輸出格式如下，請參考範例圖片：

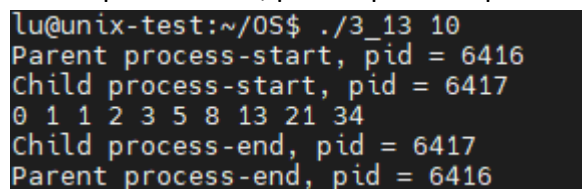
Parent process-start, pid = <parent's pid>

Child process-start, pid = <child's pid>

<Fibonacci sequence>

Child process-end, pid = <child's pid>

Parent process-end, pid = <parent's pid>



```
lu@unix-test:~/OS$ ./3_13 10
Parent process-start, pid = 6416
Child process-start, pid = 6417
0 1 1 2 3 5 8 13 21 34
Child process-end, pid = 6417
Parent process-end, pid = 6416
```

Rules:

1. 請確保你的.c/.cpp 檔能被 gcc/g++編譯 (不能的話則扣 5 分)
2. 請確保你的程式能 run 在 linux 的環境上 (不能的話則扣 5 分)
3. 請將上傳至 E3 的檔案命名為 hw3_13_學號.c/.cpp (不符合規定扣 2 分)

- 3.17 In Exercise 3.13, the child process must output the Fibonacci sequence, since the parent and child have their own copies of the data. Another approach to designing this program is to establish a shared-memory segment between the parent and child processes. This technique allows the child to write the contents of the Fibonacci sequence to the shared-memory segment and has the parent output the sequence when the child completes. Because the memory is shared, any changes the child makes will be reflected in the parent process as well.

This program will be structured using POSIX shared memory as described in Section 3.5.1. The program first requires creating the data structure for the shared-memory segment. This is most easily accomplished using a struct. This data structure will contain two items: (1) a fixed-sized array of size `MAX_SEQUENCE` that will hold the Fibonacci values and (2) the size of the sequence the child process is to generate—`sequence_size`, where `sequence_size ≤ MAX_SEQUENCE`. These items can be represented in a struct as follows:

```
#define MAX_SEQUENCE 10

typedef struct {
    long fib_sequence[MAX_SEQUENCE];
    int sequence_size;
} shared_data;
```

The parent process will progress through the following steps:

- a. Accept the parameter passed on the command line and perform error checking to ensure that the parameter is $\leq \text{MAX_SEQUENCE}$.
- b. Create a shared-memory segment of size `shared_data`.
- c. Attach the shared-memory segment to its address space.
- d. Set the value of `sequence_size` to the parameter on the command line.
- e. Fork the child process and invoke the `wait()` system call to wait for the child to finish.
- f. Output the value of the Fibonacci sequence in the shared-memory segment.
- g. Detach and remove the shared-memory segment.

Because the child process is a copy of the parent, the shared-memory region will be attached to the child's address space as well as the parent's. The child process will then write the Fibonacci sequence to shared memory and finally will detach the segment.

One issue of concern with cooperating processes involves synchronization issues. In this exercise, the parent and child processes must be synchronized so that the parent does not output the Fibonacci sequence until the child finishes generating the sequence. These two processes will be synchronized using the `wait()` system call; the parent process will invoke `wait()`, which will cause it to be suspended until the child process exits.

Input:

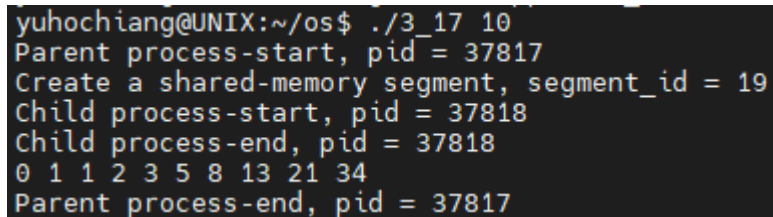
請於 **command line** 上輸入參數 – Number of sequence

1. Number of sequence: 指定輸出前幾項數列
2. 測試時會給予不同參數，必須有 **error checking** 機制，避免收取不合法參數
3. 測試時，不會要求輸出超過 10 項

Output:

在 process 起始與結束分別印出相關資訊，輸出格式如下，請參考範例圖片：

```
Parent process-start, pid = <parent's pid>
Create a shared-memory segment, segment_id = <segment id>
Child process-start, pid = <child's pid>
Child process-end, pid = <child's pid>
<Fibonacci sequence>
Parent process-end, pid = <parent's pid>
```



```
yuhochiang@UNIX:~/os$ ./3_17 10
Parent process-start, pid = 37817
Create a shared-memory segment, segment_id = 19
Child process-start, pid = 37818
Child process-end, pid = 37818
0 1 1 2 3 5 8 13 21 34
Parent process-end, pid = 37817
```

Rules:

1. 一定用 shared-memory 實作 (沒有的話則 0 分)
2. 請確保你的.c/.cpp 檔能被 gcc/g++編譯 (不能的話則扣 5 分)
3. 請確保你的程式能 run 在 linux 的環境上 (不能的話則扣 5 分)
4. 請將上傳至 E3 的檔案命名為 hw3_17_學號.c/.cpp (不符合規定扣 2 分)

3.5.1 An Example: POSIX Shared Memory

Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. Here, we explore the POSIX API for shared memory.

A process must first create a shared memory segment using the `shmget()` system call (`shmget()` is derived from SHared Memory GET). The following example illustrates the use of `shmget()`:

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

This first parameter specifies the key (or identifier) of the shared-memory segment. If this is set to `IPC_PRIVATE`, a new shared-memory segment is created. The second parameter specifies the size (in bytes) of the shared-memory segment. Finally, the third parameter identifies the mode, which indicates how the shared-memory segment is to be used—that is, for reading, writing, or both. By setting the mode to `S_IRUSR | S_IWUSR`, we are indicating that the owner may read or write to the shared-memory segment. A successful call to `shmget()` returns an integer identifier for the shared-memory segment. Other processes that want to use this region of shared memory must specify this identifier.

Processes that wish to access a shared-memory segment must attach it to their address space using the `shmat()` (SHared Memory ATtach) system call. The call to `shmat()` expects three parameters as well. The first is the integer identifier of the shared-memory segment being attached, and the second is a pointer location in memory indicating where the shared memory will be attached. If we pass a value of `NULL`, the operating system selects the location on the user's behalf. The third parameter identifies a flag that allows the shared-memory region to be attached in read-only or read-write mode; by passing a parameter of 0, we allow both reads and writes to the shared region. We attach a region of shared memory using `shmat()` as follows:

```
shared_memory = (char *) shmat(id, NULL, 0);
```

If successful, `shmat()` returns a pointer to the beginning location in memory where the shared-memory region has been attached.

Once the region of shared memory is attached to a process's address space, the process can access the shared memory as a routine memory access using the pointer returned from `shmat()`. In this example, `shmat()` returns a pointer to a character string. Thus, we could write to the shared-memory region as follows:

```
sprintf(shared_memory, "Writing to shared memory");
```

Other processes sharing this segment would see the updates to the shared-memory segment.

Typically, a process using an existing shared-memory segment first attaches the shared-memory region to its address space and then accesses (and possibly updates) the region of shared memory. When a process no longer requires access to the shared-memory segment, it detaches the segment from its address space. To detach a region of shared memory, the process can pass the pointer of the shared-memory region to the `shmdt()` system call, as follows:

```
shmdt(shared_memory);
```

Finally, a shared-memory segment can be removed from the system with the `shmctl()` system call, which is passed the identifier of the shared segment along with the flag `IPC_RMID`.

The program shown in Figure 3.16 illustrates the POSIX shared-memory API just discussed. This program creates a 4,096-byte shared-memory segment. Once the region of shared memory is attached, the process writes the message `Hi There!` to shared memory. After outputting the contents of the updated memory, it detaches and removes the shared-memory region. We provide further exercises using the POSIX shared-memory API in the programming exercises at the end of this chapter.