OS HW2

Part1

> Operating System Concept Chapter 2 Exercises: 2.9, 2.10, 2.12. 2.17

2.9

The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories, and discuss how they differ.

Answer:

For that question, the desciption of two categories and the difference between them are listed below.

Category 1:

For this category, the keyword is "*process isolation*". To be more specifically, this kind of services provided by an operating system is to enforce protection between different processes running concurrently in the system. For example, processes A are only allowed to access the memory locations that are associated with its own address spaces, but can not access the memory locations associated with process B. What's more, processes are not allowed to corrupt files associated with other users, as if this kind of operation is leagal, my lead to confliction. Accessing devices directly without operating system intervention are also unallowed.

Category 2:

For this category, the keyword is "*resource abstraction*". To be more specifically, The second class of services provided by an operating system is to provide new functionality that is not supported directly by the underlying hardware. As we all know, process and virtual memory as well as file system are important basic abstract concept in an operating system.

Difference:

Macroscopically speaking, the first category of the services and functions is in order to avoid the confliction between different processes which can access or modify the content of memory or file. Without it, if another process can arbitrarily access and modify the content of memory or file which is associated with itself, then will likely cause error in an excuting process. While the second category of the services and functions is work for solving the the difficult in the implement of "*process isolation*". With this "*resource abstraction*" service, the process can hold the resource more independently and avoid many confliction.

2.10

Describe three general methods for passing parameters to the operating system.

Answer:

Three general methods are passing by register, passing by the block or table in the memory and passing by push/pop operation on the stack.

2.12

What are the advantages and disadvantages of using the same systemcall interface of manipulating both files and devices?

Answer:

Each device can be accessed as though it was a file in the file system. Since most of the kernel deals with devices through this file interface, it is relatively easy to add a new device driver by implementing the hardware-specific code to support this abstract file interface. Therefore, this benefits the development of both user program code, which can be written to access devices and files in the same manner, and devicedriver code, which can be written to support a well-defined API. The disadvantage with using the same interface is that it might be difficult to capture the functionality of certain devices within the context of the file access API, thereby resulting in either a loss of functionality or a loss of performance. Some of this could be overcome by the use of the ioctl operation that provides a general-purpose interface for processes to invoke operations on devices.

2.17

Why is the separation of mechanism and policy desirable?

Answer:

Mechanism and policy must be separate to ensure that systems are easy to modify. No two system installations are the same, so each installation may want to tune the operating system to suit its needs. With mechanism and policy separate, the policy may be changed at will while the mechanism stays unchanged. This arrangement provides a more flexible system.

Part2

Operating System Concept Chapter 3 Exercises: 3.1

3.1

Using the program shown below, explain what the output will be at *LINE A*.

```c
#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;
    }
}
```

Answer:

The result is still 5 as the child updates its copy of value. When control returns to the parent, its value remains at 5. The reason is that the resource shared between the parent process and the child process is only the code space, the other recouces including stack and global variable are totally copied by the child process. I will draw illustration to explian it.

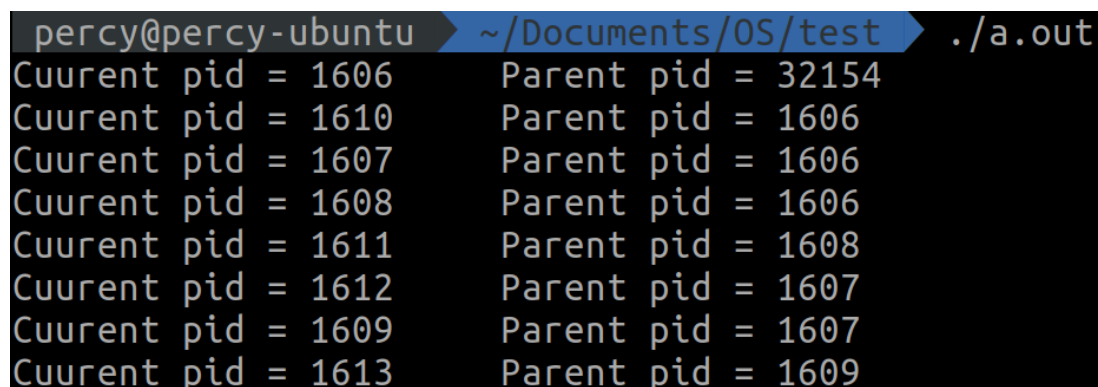*Illustration:*

Part3

3.2
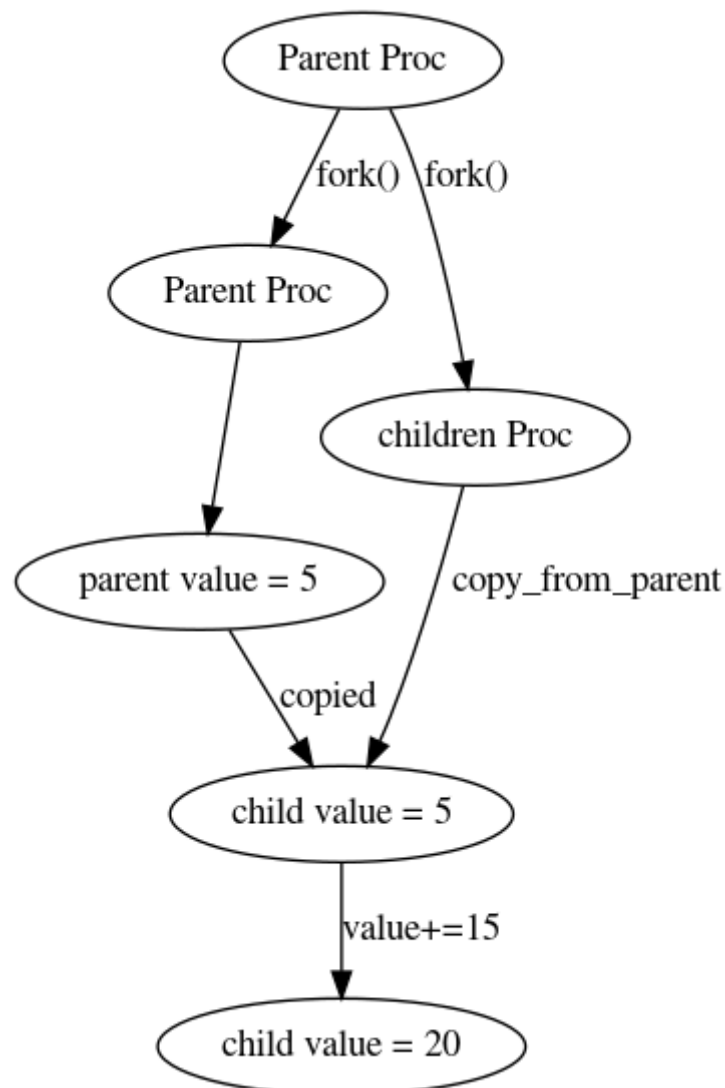
Including the inital parent process, how many processed are created by the program shown below?

```c
#include <stdio.h>
#include <unistd.h>
int main()
{
    /* fork a child process */
    fork();
    /* fork another child process */
    fork();
    /* and fork another */
    fork();
    return 0;
}
```

And inorder to make the process creating situation clearer, I modified the code.

```c
#include <stdio.h>
#include <unistd.h>
int main()
{
    /* fork a child process */
    fork();
    /* fork another child process */
    fork();
    /* and fork another */
    fork();
    printf("Cuurent pid = %d", getpid());
    printf("Parent pid = %d", getppid());
    return 0;
}
```
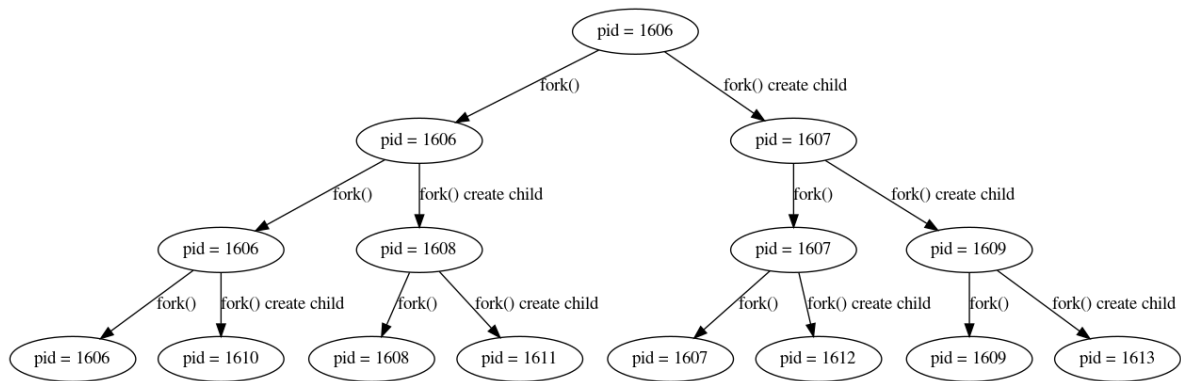
Answer:

```
percy@percy-ubuntu  ~/Documents/OS/test  ./a.out
Cuurent pid = 1606      Parent pid = 32154
Cuurent pid = 1610      Parent pid = 1606
Cuurent pid = 1607      Parent pid = 1606
Cuurent pid = 1608      Parent pid = 1606
Cuurent pid = 1611      Parent pid = 1608
Cuurent pid = 1612      Parent pid = 1607
Cuurent pid = 1609      Parent pid = 1607
Cuurent pid = 1613      Parent pid = 1609
```

According to the result shown in the sreenshot, we can know that there are 8 processes created. I will draw flow chart to explian it.

Part 4

| Operating System Concept Chapter 4 Exercises: 4.8 4.9. 4.10 4.17 4.19

4.8

Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

Answer:

Example 1:

We can easily realize that multithreading's advantage is in the seperating a complex task to small tasks and then try to complish them at the same time with several threads. So if the program is sequential, task seperating and excute at the same time become impossible. For the program can excute correctly, obviously, single-threaded solution will have a better performance. A program that need to print strings in order, net salary, the progarm will have a better performance when it is single threaded. The code and time cost are listed below.

Multithread program:

```c
// File: thread.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

void *fun_1()
{
    printf("fun_1 ");
}

void *fun_2()
{
    printf("fun_2\n");
}
int main()
{
    for (int i =0 ;i<10000;i++){
    pthread_t pid1;
    pthread_t pid2;
```

```
    if(pthread_create(&pid1,NULL,fun_1,NULL))
    {
        return -1;
    }
    if(pthread_create(&pid2,NULL,fun_2,NULL))
    {
        return -1;
    }
    pthread_join(pid1, NULL);
    pthread_join(pid2, NULL);}
    return 0;
}
```

Single thread program:

```
// singleThread.c
#include <stdio.h>
int main(){
    for (int i =0;i<10000;i++)
    printf("fun_1 fun_2\n");
    return 0;
}
```

Time cost:

Example 2:

If a program need closely monitor its own working space such as open files, environment variables, and current working directory, using multi-thread machnisam obviously will affect the implement of these function which can be an important demand. The typical example is a "shell" program such as the C-shell or Korn shell.

4.9

Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

Answer:

Simply, we can divided these situation into two categories.

The first situation is that the program's mission can be divided to many seperated and independent tasks, so multiple kernel threads can claculate them at the same time. On a multi-processor system, it can obviously bring the progress on program running speed.

The other situation is that when a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful lmanner. while a single-threaded process will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multithreaded solution would perform better even on a single-processor system as it can avoid many error solving and recovering time.

4.10

Which of the following components of program state are shared across threads in a multithreaded process?
    a. Register values
    b. Heap memory
    c. Global variables
    d. Stack memory

Answer:

The components of program state are shared: b. Heap memory & c. Global variables
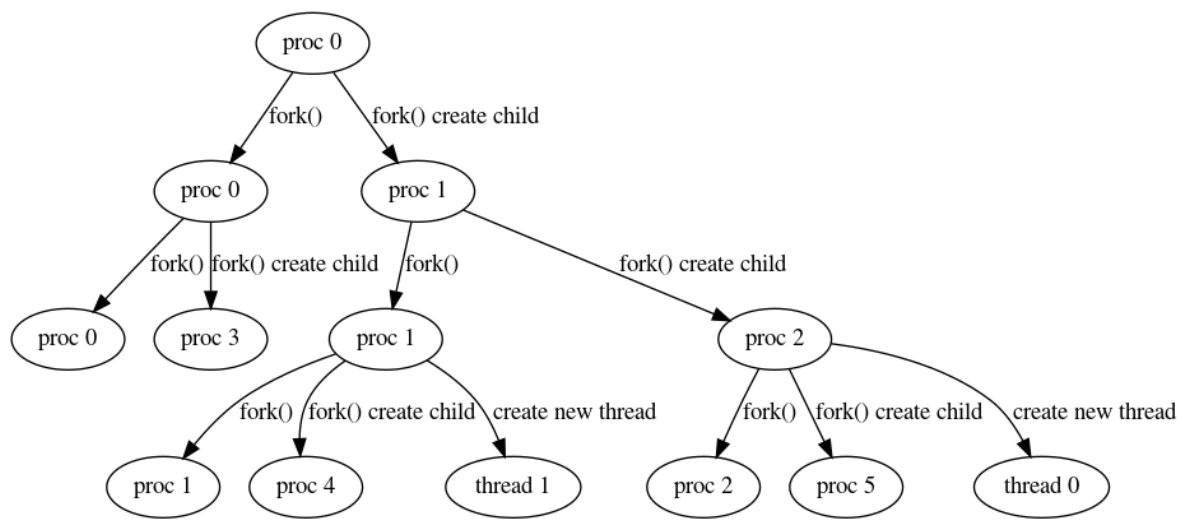
4.17

Consider the following code segment:

```
pid t pid;
pid = fork();
if (pid == 0) { /* child process */
    fork();
    thread create( . . .);
}
fork();
```

    a. How many unique processes are created?
    b. How many unique threads are created?

Answer:

Actually, there should be 8 threads and 6 processes.

Here's the diagrams to make it clear:

4.19

The program shown in Figure 4.23 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

```c
#include <pthread.h>
#include <stdio.h>
int value = 0;
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;
    pid = fork();
    if (pid == 0) { /* child process */
        pthread_attr_init(&attr);
        pthread_create(&tid,&attr,runner,NULL);
        pthread_join(tid,NULL);
        printf("CHILD: value = %d",value); /* LINE C */
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE P */
    }
}
void *runner(void *param) {
    value = 5;
    pthread exit(0);
}
```

Answer:

```
percy@percy-ubuntu   ~/Documents/OS/test   ./a.out
CHILD: value = 5PARENT: value = 0%
```

That is because the child process created by `fork()` will not shared the resource except code, while the thread create using the same address space and code, so the value change in the created thread will affect the child process's result but will not affect the parent process.

I will use graph to explian clearer:
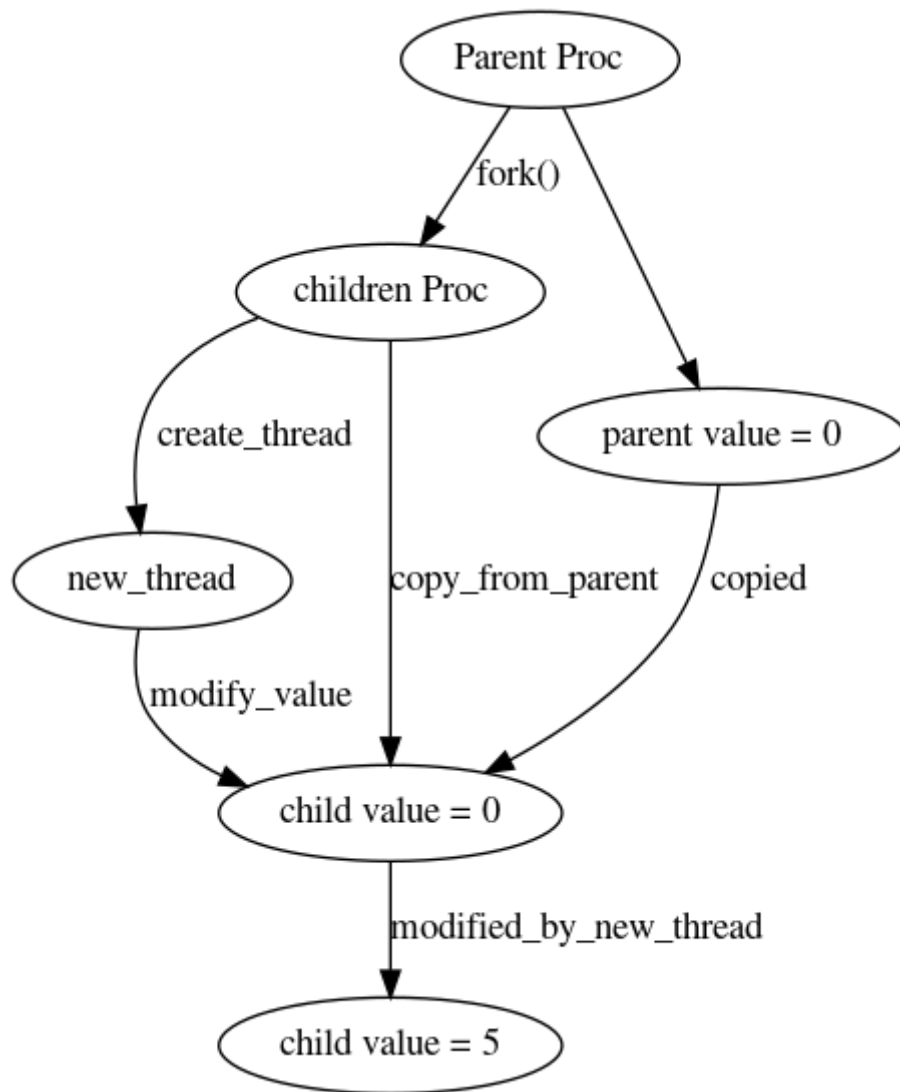
Part 5

> Compile and run the clone.c program twice, with 'vm' and without 'vm' as the argument, respectively. Take the screenshots of the running results and explain why the output is different .
>
> Hint: Understand the meaning of the CLONE_VM flag of the clone system call.

Screenshot of the running results:


```
percy@percy-ubuntu   ~/Documents/OS/test   gcc clone.c -o clone
percy@percy-ubuntu   ~/Documents/OS/test   ./clone
In Child: child sees buf = "hello from parent"
In Child: child changes buf = "hello from child"
In Parent: parent sees buf = "hello from parent"
percy@percy-ubuntu   ~/Documents/OS/test   ./clone vm
In Child: child sees buf = "hello from parent"
In Child: child changes buf = "hello from child"
In Parent: parent sees buf = "hello from child"
percy@percy-ubuntu   ~/Documents/OS/test
```

Explaination:

Firstly, we need to know the meaning of CLONE_VM flag, so find the related document.

```
CLONE_VM (since Linux 2.0)
          If CLONE_VM is set, the calling process and the child process
          run in the same memory space.  In particular, memory writes
          performed by the calling process or by the child process are
          also visible in the other process.  Moreover, any memory
          mapping or unmapping performed with mmap(2) or munmap(2) by
          the child or calling process also affects the other process.

          If CLONE_VM is not set, the child process runs in a separate
          copy of the memory space of the calling process at the time of
          clone().  Memory writes or file mappings/unmappings performed
          by one of the processes do not affect the other, as with
          fork(2).
```

Through the declaration of `CLONE_VM`, we can known that if the flag is not set that clone() is similar to fork() and all memory space will be have a separate copy for the running of the child process. While if the flag is set, than the the child process will run in the same memory space with the parent process, which means that the variable changed in the child process will also affect the parent process.

For you can understand cleared, two illustrations will given below.

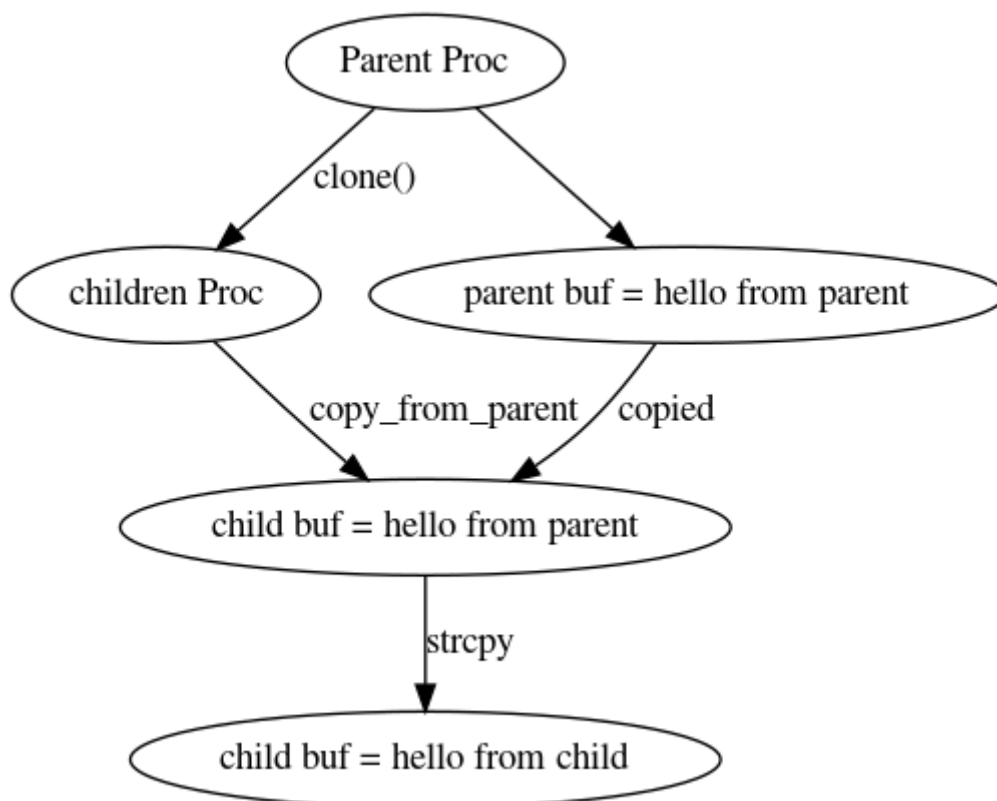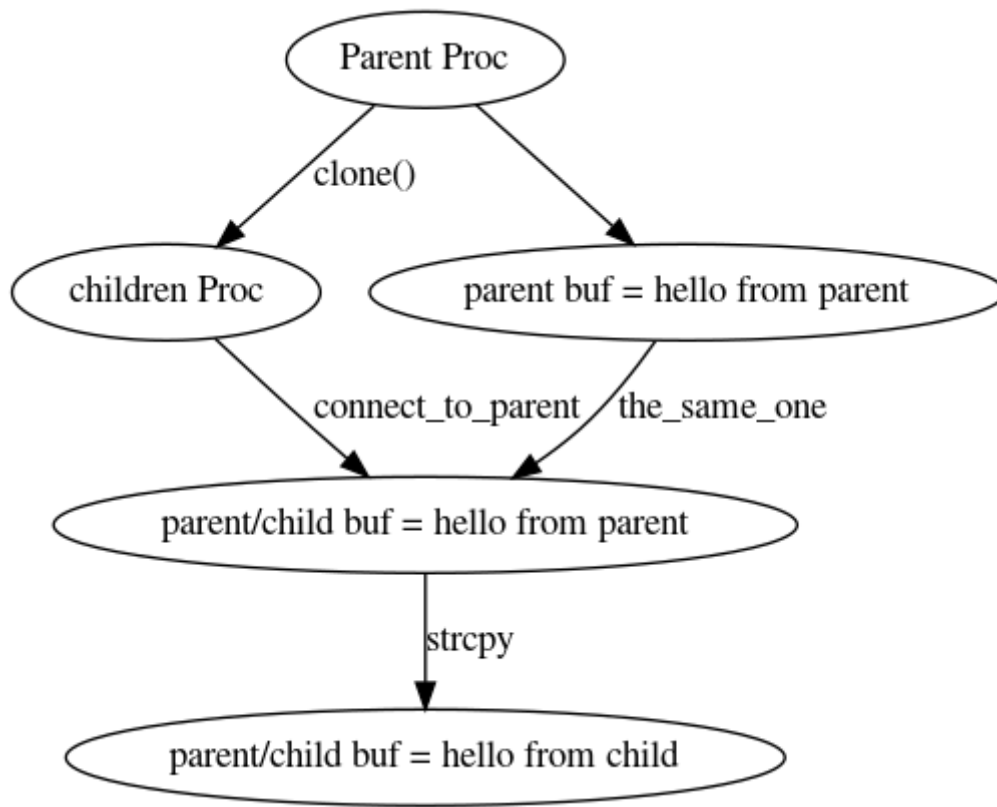*Illustration 1(flag is not set):*

Illustration 2(flag is set):

Part 6

| Operating System Concept Chapter 5 Exercises: 5.3 5.4 5.5 5.8

5.3

Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

| Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as *future-knowledge scheduling*.

Answer:

a) The result is 10.53.

First- Come, First-Served (FCFS) Scheduling

The processes arrive in the order: P_1, P_2, P_3. The Gantt Chart for the schedule is:
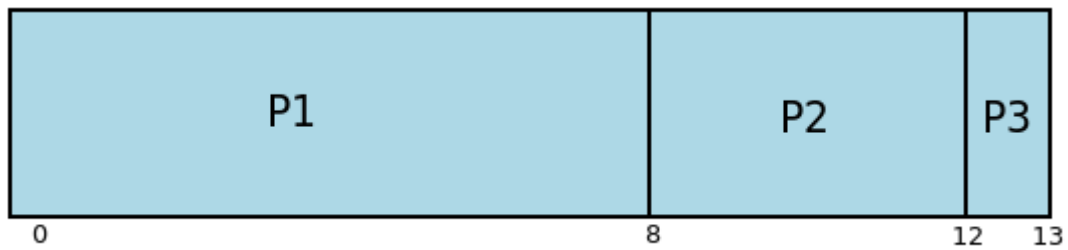
P1	P2	P3

0   8   12   13

Turnaround time = finish time – arrival time

Average Turnaround time = (total finish time - total arrival time)/3 = ((8+12+13) - (0+0.4+1))/3 = 10.53

b) The result is 9.53.

Shortest-Job-First (SJF) Scheduling (**Attention:** Non-preemptive mode)

The Gantt Chart for the schedule is:
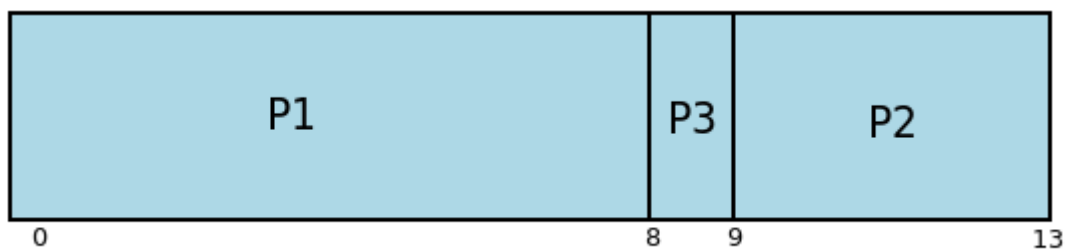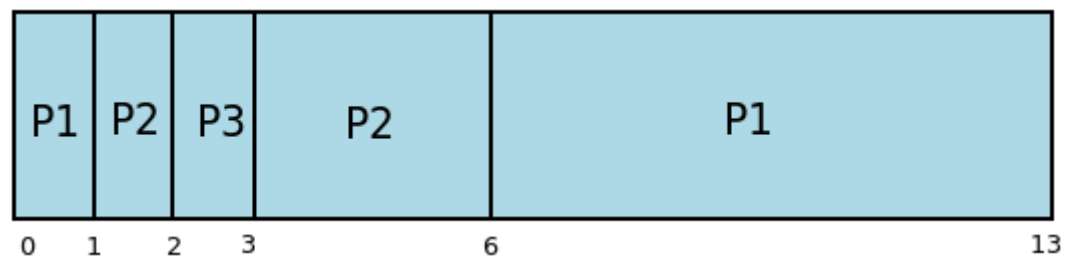
P1	P3	P2

0   8   9   13

Turnaround time = finish time – arrival time

Average Turnaround time = (total finish time - total arrival time)/3 = ((8+9+13) - (0+0.4+1))/3 = 9.53

c) The result is 6.86.

The Gantt Chart for the schedule is:

P1	P2	P3	P2	P1

0   1   2   3   6   13

Turnaround time = finish time – arrival time

Average Turnaround time = (total finish time - total arrival time)/3 = ((3+6+13) - (0+0.4+1))/3 = 6.86

5.4

Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

| Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?
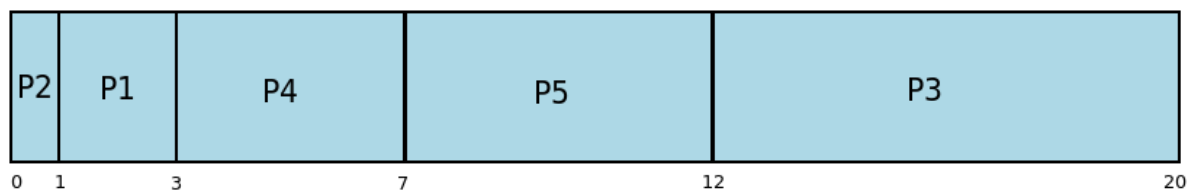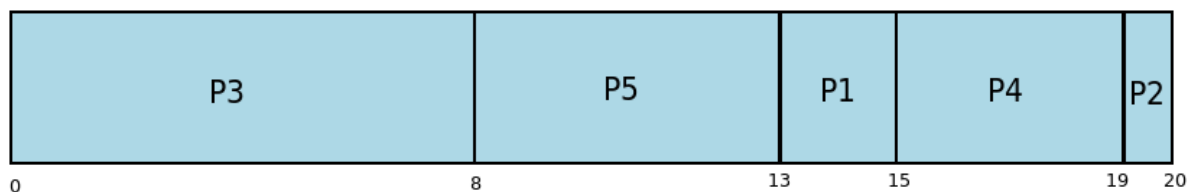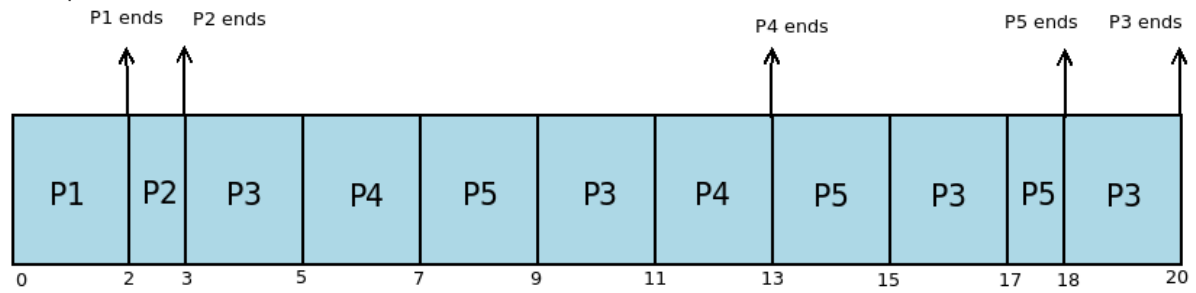
Answer:

a)

FCFS:

SJF:
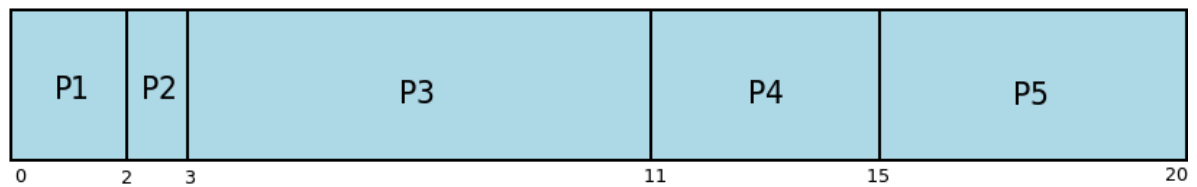
nonpreemptive priority(a **larger** priority number implies a **higher** priority):

RR (quantum = **2**):

b)

Turnaround time of each process for each of the scheduling algorithms:

Process	FCFS	SJF	priority	RR
P_1	2	3	15	2
P_2	3	1	20	3
P_3	11	20	8	20
P_4	15	7	19	13
P_5	20	12	13	18

c)

Waiting time of each process for each of the scheduling algorithms:

Process	FCFS	SJF	priority	RR
P_1	0	1	13	0
P_2	2	0	19	2
P_3	3	12	0	12
P_4	11	3	15	9
P_5	15	7	8	13

b)

Average waiting time (over all processes):

Algorithm	FCFS	SJF	priority	RR
Average waiting time	6.2	4.6	11	7.2

According to the table above, shortest-Job-First (SJF) Scheduling results in the minimum average waiting time.

5.5

The following processes are being scheduled using a preemptive, roundrobin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an **idle task** (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.

b. What is the turnaround time for each process?

c. What is the waiting time for each process?

d. What is the CPU utilization rate?

Answer:

a) Gantt chart:
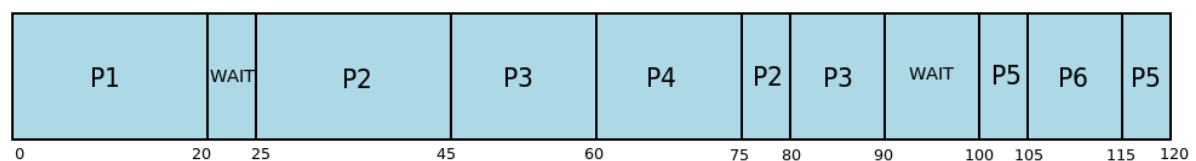
b) Turnaround time for each process(Tips: Turnaround time = finish time – arrival time):

Process	Turnaround time
P_1	20
P_2	55
P_3	60
P_4	15
P_5	20
P_6	10

c) Waiting time for each process(Tips: Waiting time = turnaround time – burst time):

Process	Waiting time
P_1	0
P_2	40
P_3	35
P_4	0
P_5	10
P_6	0

d) CPU utilization rate = (Total time - CPU waiting time) / Total time = (120 -10 -5)/120 = 87.5%

5.8

Suppose that a CPU scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

Answer:

Let me elaborate it with my reasons.

Firstly, as the CPU scheduling algorithm favors those processes that have used the least processor time in the recent past, the request from the I/O-bound programs do need relatively short CPU burst.

Secondly, the CPU resource will not always be occupaied by I/O-bound programs, the I/O-bound programs will relinquish the CPU relatively often to do their I/O, so the CPU-bound programs will not starve.

Part 7

> According to the ppt we have learnt in the class, explain the reason why the Mars Pathfinder had reset itself, and how to avoid that.

Answer:

The reason:

The detailed process is elaborated below.

"*The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus synchronized with mutual exclusion locks (mutexes). Other higher priority threads took precedence when necessary, including a very high priority bus management task, which also accessed the bus with mutexes. Unfortunately in this case, a long-running communications task, having higher priority than the meteorological task, but lower than the bus management task, prevented it from running.*

Soon, a watchdog timer noticed that the bus management task had not been executed for some time, concluded that something had gone wrong, and ordered a total system reset. (Engineers later confessed that system resets had occurred during pre-flight tests. They put these down to a hardware glitch and returned to focusing on the mission-critical landing software.)"   -- a quote form link

Breifly, the reason is that when a low priority process need to access the bus and it has got a mutex, but at this time a middle priority process which will run for a long time is ready, and make the low priority process pause, then a very high priority process need to wait for the mutex hold by the low priority process for a long time, because it needs to wait until the end of the long time middle priority process. And once the very high priority process wait to long time, it will be seen as a serious error in the system and start to reset operation.

The avoid approach:

We can adopte a *priority inheritance* strategy. When a high-priority process waits for a resource of a low-priority process, the low-priority process temporarily obtains a high priority. After the shared resource is released, the low-priority process will return to the original priority. This approach can perfectly solve the problem mentioned before.