Tutorial 2

CS2106: Introduction to Operating Systems

Outline

1. Recap
2. Tutorial Questions

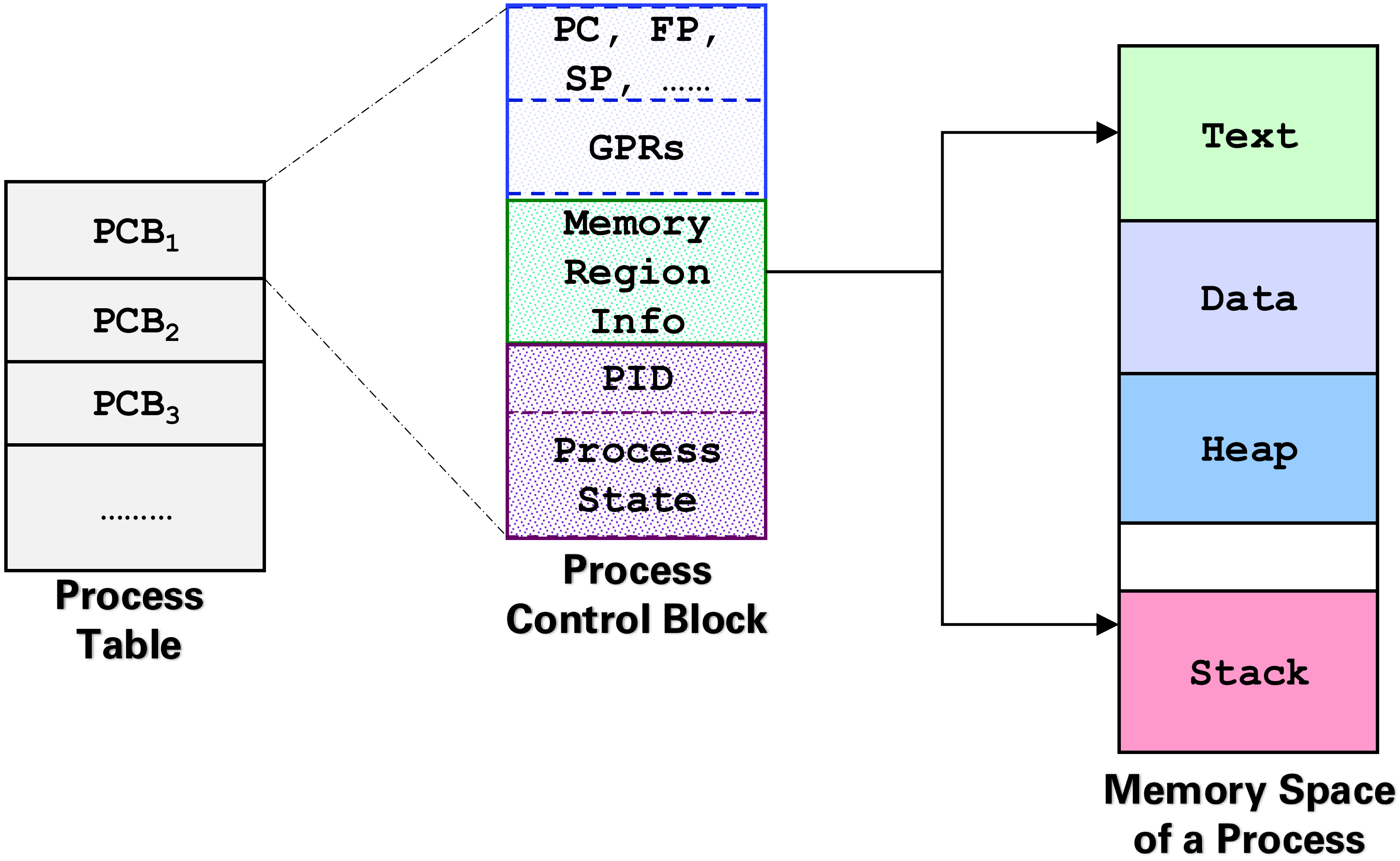
Objectives

- To understand the process behaviour based on the process abstraction model in Unix.
  - What happens during process creation?
  - What happens when we call **`fork()`**?
- Use **`fork()`** and **`execl()`** in to solve problems.

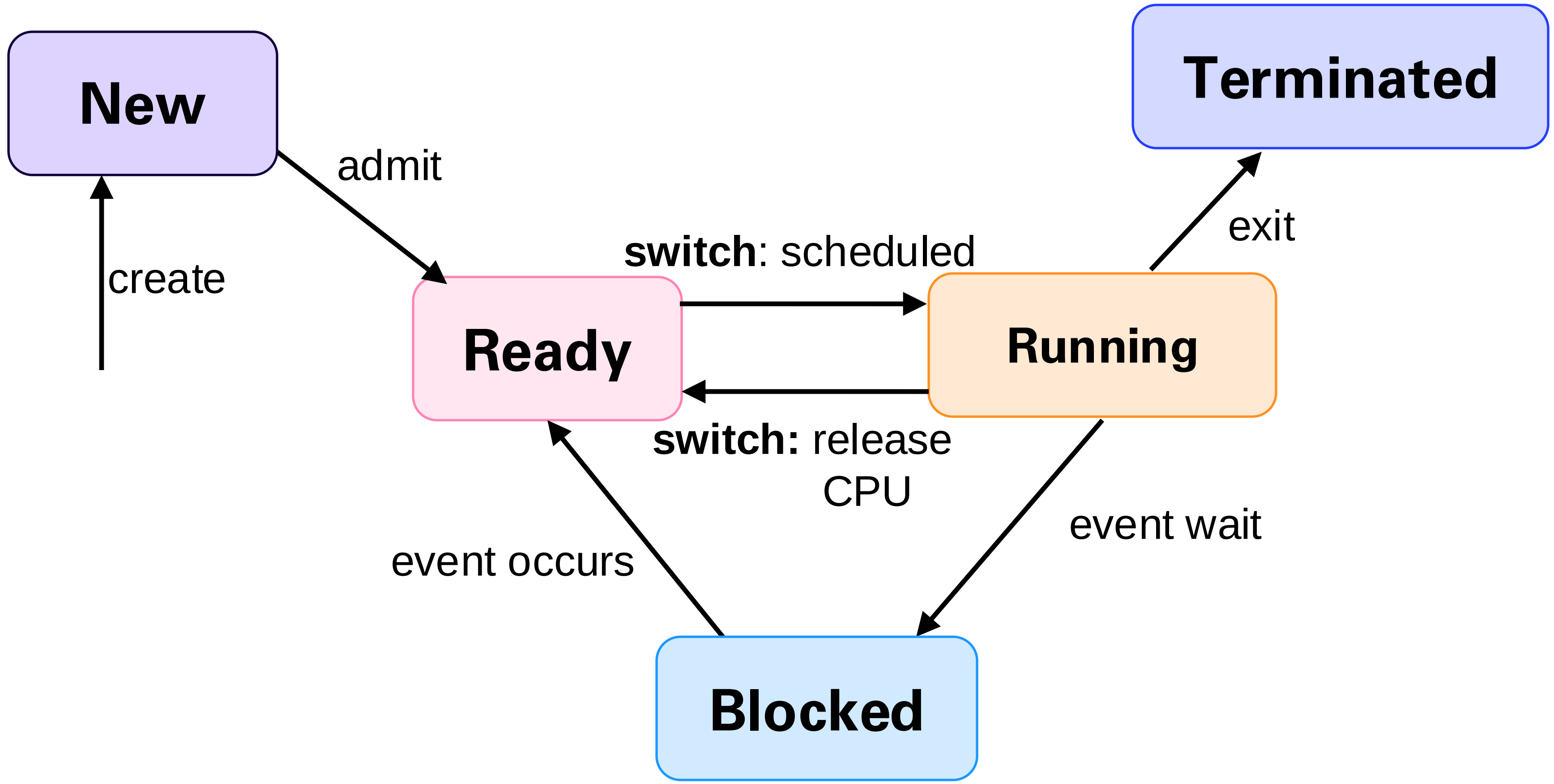
Recap

Lecture Contents

Topic Summary



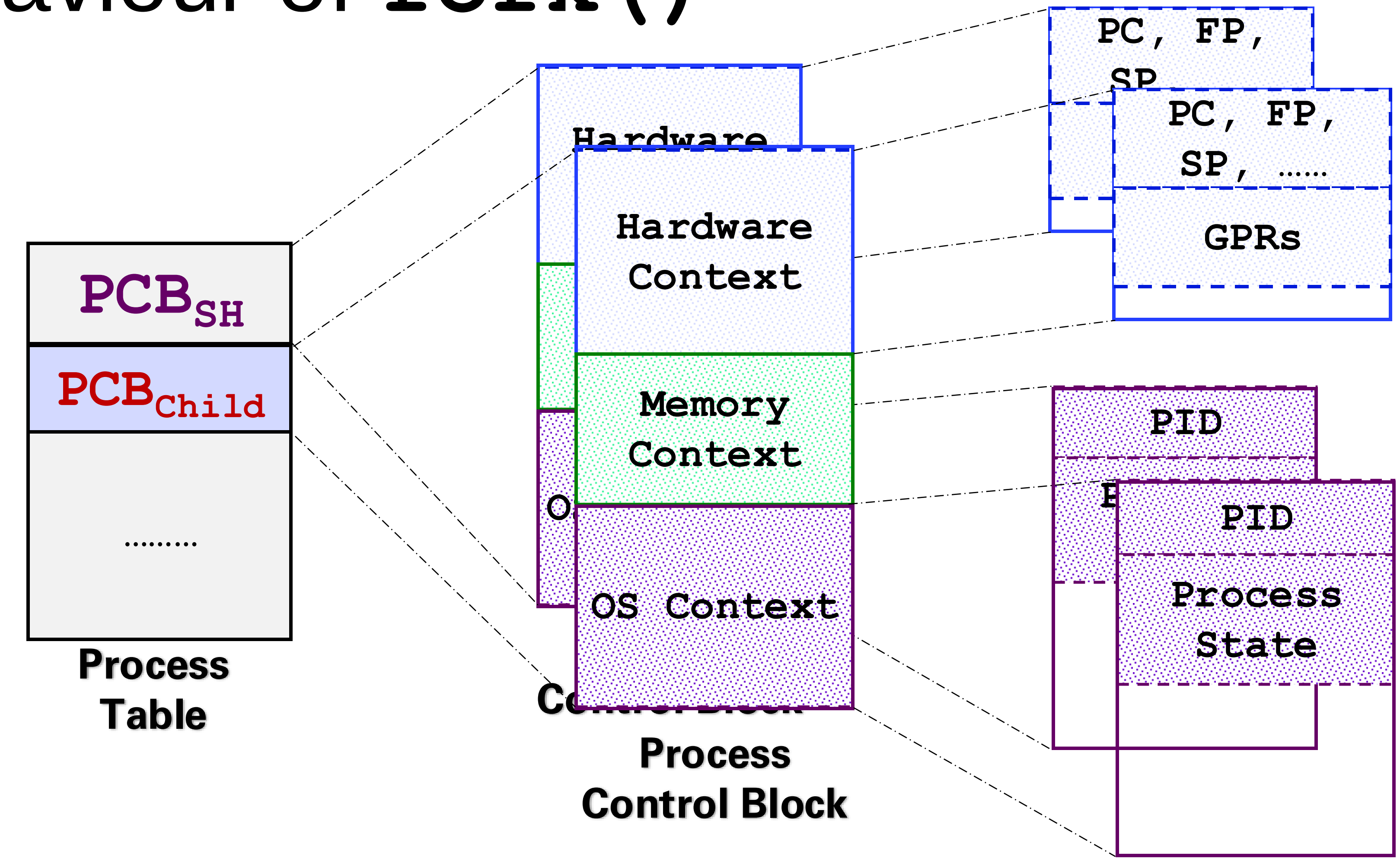
Process State Model



Behaviour of `fork()`

- Creates a new process (child process) that is almost an exact copy of the parent process.
  - Process Table Entry is created for the child process

Behaviour of `fork()`

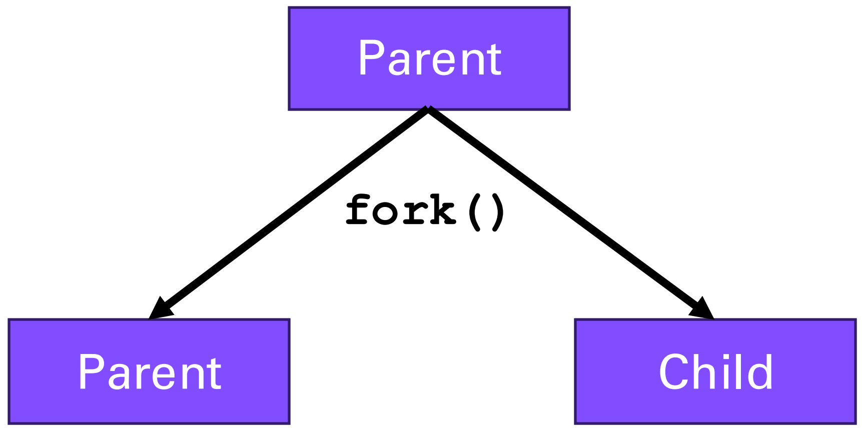


Behaviour of `fork()`

- Hardware context from child process is copied over from the parent.
  - Registers are copied over.
- Parent and child process have identical but **<u>independent memory spaces</u>**
  - Text (Code), Data, Stack and Heap region from the parent's process memory space is copied over to child process.
  - Copy on Write Optimization: Only create a copy when a write is performed on a location.
- Child process acquires shared resources from parent's OS context
  - Open Files, Current Working directories, etc…

Creating a Child Process: **`fork()`**

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

int main() {
    int var = 1234;
    int result;

    result = fork();
    if (result != 0){
        printf("Parent: Var is %i\n", var);
        var++;
        printf("Parent: Var is %i\n", var);
    } else {
        printf("Child: Var is %i\n", var);
        var--;
        printf("Child: Var is %i\n", var);
    }
}
```



result = PID of Child

result = 0

Parent: Var is 1234
Parent: Var is 1235

Child: Var is 1234
Child: Var is 1233

Creating a Child Process: **`fork()`**

- Child process begins execution at the same point where **`fork()`** is called in the parent process.
- It shares the same executable code initially.
  - Child's executable code changes if you replace its executable image with **`execl()`**

`execl()` System Call
- Replace current executing process image with a new one
- Only code is replaced, PID and other information still intact

Question 1

Question 1(a)

Note that **`wait()`** does not block when a process has no children.

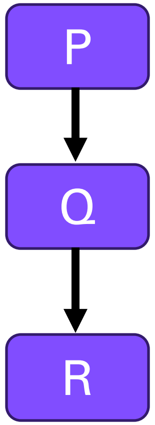
Evaluate whether the described behaviour is correct or incorrect.

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Behaviour
(i) Process Q ***always*** terminate before P.
(ii) Process R can terminate at any time w.r.t. P and Q.

Process Tree



Question 1(a)

Note that `wait()` does not block when a process has no children.

Evaluate whether the described behaviour is correct or incorrect.

C code:

```
00 int main( ) {
01     //This is process P
02     if ( fork() == 0 ){
03         //This is process Q
04         if ( fork() == 0 ) {
05             //This is process R
06             ......
07             return 0;
08         }
09         <Point α>
10     }
11     wait(NULL); <Point β>
12
13     return 0;
14 }
```

Behaviour
(i) Process Q ***always*** terminate before P.
(ii) Process R can terminate at any time w.r.t. P and Q.

Statement (i) is True
But statement (ii) is False

Thought Process:
Which processes can execute `wait(NULL)` at line 11?

Question 1(a)

Note that **`wait()`** does not block when a process has no children.

Evaluate whether the described behaviour is correct or incorrect.

C code:

```
00 int main( ) {
01     //This is process P
02     if ( fork() == 0 ){
03         //This is process Q
04         if ( fork() == 0 ) {
05             //This is process R
06             ......
07             return 0;
08         }
09         <Point α>
10     }
11     wait(NULL); <Point β>
12 
13     return 0;
14 }
```

Behaviour
(i) Process Q ***always*** terminate before P.
(ii) Process R can terminate at any time w.r.t. P and Q.

Statement (i) is True
But statement (ii) is False

Thought Process:
Which processes can execute **`wait(NULL)`** at line 11?
Answer: Processes P and Q

- At line 7, process R returns 0 and terminates.
- At line 11, process Q will wait for process R to terminate.
- At line 11, process P will wait for process Q to terminate.
- **Therefore statement (ii) is False**

Question 1(b)

Note that `wait()` does not block when a process has no children.

Evaluate whether the described behaviour is correct or incorrect.

C code:

```
00 int main( ) {
01     //This is process P
02     if ( fork() == 0 ){
03         //This is process Q
04         if ( fork() == 0 ) {
05             //This is process R
06             ......
07             return 0;
08         }
09         wait(NULL); <Point α>
10     }
11      <Point β>
12
13     return 0;
14 }
```

Behaviour

(i) Process Q ***always*** terminate before P.

(ii) Process R can terminate at any time w.r.t. P and Q.

Thought Process:

Which processes can execute `wait(NULL)` at line 9?

Question 1(b)

Note that `wait()` does not block when a process has no children.

Evaluate whether the described behaviour is correct or incorrect.

C code:

```
00 int main( ) {
01     //This is process P
02     if ( fork() == 0 ){
03         //This is process Q
04         if ( fork() == 0 ) {
05             //This is process R
06             ......
07             return 0;
08         }
09         wait(NULL); <Point α>
10     }
11      <Point β>
12
13     return 0;
14 }
```

Behaviour
(i) Process Q ***always*** terminate before P.
(ii) Process R can terminate at any time w.r.t. P and Q.

Thought Process:

Which processes can execute `wait(NULL)` at line 9?

Answer: Only Process Q

- At line 7, process R returns 0 and terminates.
- At line 9, process Q will wait for process R to terminate.
- Process P does not wait for process Q to terminate.
- **Therefore statement (i) and (ii) are FALSE**

Question 1(c)

Note that `wait()` does not block when a process has no children.

Evaluate whether the described behaviour is correct or incorrect.

C code:

```
00 int main( ) {
01     //This is process P
02     if ( fork() == 0 ){
03         //This is process Q
04         if ( fork() == 0 ) {
05             //This is process R
06             ......
07             return 0;
08         }
09         execl(valid executable....); <α>
10     }
11     wait(NULL); <Point β>
12 
13     return 0;
14 }
```

Behaviour
(i) Process Q ***always*** terminate before P.
(ii) Process R can terminate at any time w.r.t. P and Q.

Thought Process:

Which processes can execute `wait(NULL)` at line 11?

Question 1(c)

Note that `wait()` does not block when a process has no children.

Evaluate whether the described behaviour is correct or incorrect.

C code:

```
00 int main( ) {
01     //This is process P
02     if ( fork() == 0 ){
03         //This is process Q
04         if ( fork() == 0 ) {
05             //This is process R
06             ......
07             return 0;
08         }
09         execl(valid executable....); <α>
10     }
11      wait(NULL); <Point β>
12 
13     return 0;
14 }
```

Behaviour
(i) Process Q ***always*** terminate before P.
(ii) Process R can terminate at any time w.r.t. P and Q.

Thought Process:

Which processes can execute `wait(NULL)` at line 11?

Answer: Only Process P

- At line 7, process R returns 0 and terminates.
- At line 9, process Q's executable image is replaced with `execl()`. If the new executable does not have a `wait(NULL)`, then Q does not wait for R.
- At line 11, Process P waits for process Q to terminate.
- **Therefore statement (i) is TRUE but (ii) depends on if the new executable code contains a `wait(NULL)`**

Question 1(d)

Note that **`wait()`** does not block when a process has no children.

Evaluate whether the described behaviour is correct or incorrect.

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Behaviour
Process P **never terminates**.

Question 1(d)

Note that **`wait()`** does not block when a process has no children.

Evaluate whether the described behaviour is correct or incorrect.

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Behaviour
Process P **never terminates**.

False. Process P will terminate.

Although Process Q has an additional **`wait(NULL)`** after waiting for process R to terminate, the second **`wait(NULL)`** returns immediately as at that point, process Q has no more children.

Visualization for Question 1(d)

Question 1(d)

(Note that `wait()` does not block when a process has no children.)

Process P

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Process Q

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Question 1(d)

(Note that **`wait()`** does not block when a process has no children.)

Process P

C code:

```c
int main( ) {
    //This is process P









     wait(NULL); <Point β>

    return 0;
}
```

Process Q

C code:

```c
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
     wait(NULL); <Point β>

    return 0;
}
```

Question 1(d)

(Note that `wait()` does not block when a process has no children.)

Process Q

C code:

```c
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
     wait(NULL); <Point β>

    return 0;
}
```

Process R

C code:

```c
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
     wait(NULL); <Point β>

    return 0;
}
```

Question 1(d)

(Note that **`wait()`** does not block when a process has no children.)

Process Q

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>
    return 0;
}
```

Process R

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>
    return 0;
}
```

Question 1(d)

Process Q waits for Process R to terminate

(Note that `wait()` does not block when a process has no children.)

Process Q

C code:

```c
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q





        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Process R

C code:

```c
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            ......
            return 0;
        }
        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Question 1(d)

Process Q waits for Process R to terminate

(Note that **`wait()`** does not block when a process has no children.)

Process Q

```
C code:
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q





        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Process R

Terminated: Exit and return 0

PCB entry of R is removed

Question 1(d)

(Note that `wait()` does not block when a process has no children.)

Process Q

```
C code:
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q




        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Process R

Terminated: Exit and return 0

PCB entry of R is removed

Process Q has no more children, wait(NULL) does not block and returns immediately

Question 1(d)

(Note that **`wait()`** does not block when a process has no children.)

Process Q

C code:

```
int main( ) {
    //This is process P
    if ( fork() == 0 ){
        //This is process Q





        wait(NULL); <Point α>
    }
    wait(NULL); <Point β>

    return 0;
}
```

Process R

Terminated: Exit and return 0

PCB entry of R is removed

Process Q has no more children, wait(NULL) does not block and returns immediately

Question 1(d)

Process P waits for Process Q to terminate

(Note that `wait()` does not block when a process has no children.)

Process Q

Terminated: Exit and return 0

Process R

Terminated: Exit and return 0

Process P

C code:

```
int main( ) {
    //This is process P
…
     wait(NULL); <Point β>

    return 0;
}
```

PCB entry of Q is removed

Question 1(d)

Process P waits for Process Q to terminate

(Note that **`wait()`** does not block when a process has no children.)

Process Q

Terminated: Exit and return 0

Process R

Terminated: Exit and return 0

Process P

C code:

```
int main( ) {
    //This is process P
…
     wait(NULL); <Point β>

    return 0;
}
```

Question 1(d)

Process P waits for Process Q to terminate

(Note that **`wait()`** does not block when a process has no children.)

Process Q

Terminated: Exit and return 0

Process R

Terminated: Exit and return 0

Process P

Terminated: Exit and return 0

Process P will eventually terminate

Question 2

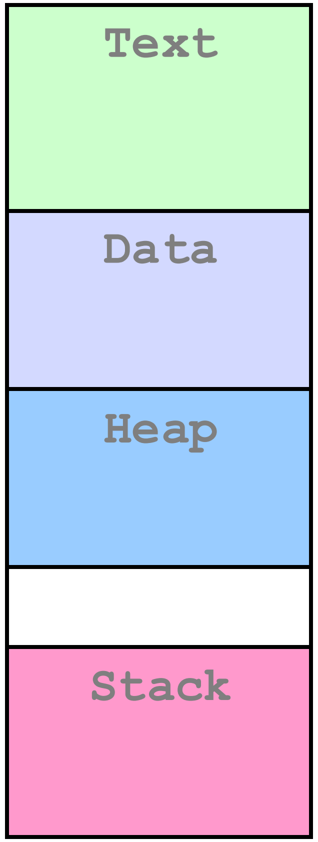
Question 2(a)

What is the difference between the 3 variables: dataX, dataY and the memory location pointed by dataZptr?

C code:

```
int dataX = 100;

int main( )
{
    pid_t childPID;

    int dataY = 200;
    int* dataZptr = (int*) malloc(sizeof(int));
```

Variable	Memory Region
`dataX`	
`dataY`	
Memory location pointed by `dataZptr`	



Memory Space of a Process

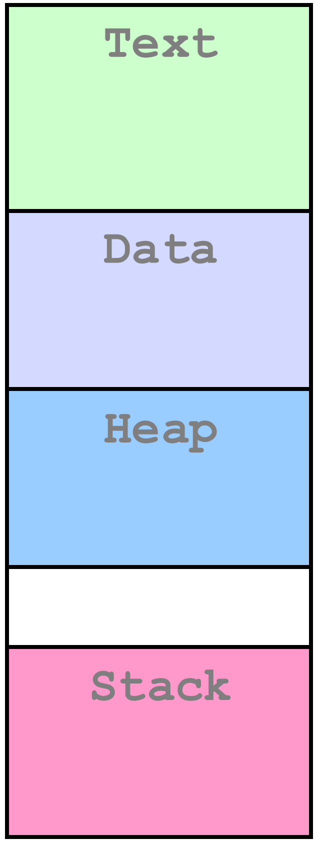
Question 2(a)

What is the difference between the 3 variables: dataX, dataY and the memory location pointed by dataZptr?

C code:

```
int dataX = 100;

int main( )
{
    pid_t childPID;

    int dataY = 200;
    int* dataZptr = (int*) malloc(sizeof(int));
```

Variable	Memory Region
`dataX`	Data
`dataY`	Stack
Memory location pointed by **`dataZptr`**	Heap



Memory Space of a Process

Question 2(b)

```
int dataX = 100;

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

    int dataY = 200;
    int* dataZptr = (int*) malloc(sizeof(int));

    *dataZptr = 300;

    //First Phase
    printf("PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    //Second Phase
    childPID = fork();
    printf("*PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    dataX += 1;
    dataY += 2;

    if(childPID == 0)
        (*dataZptr) += 3;
    else
        (*dataZptr) += 5;
    printf("#PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);
```

Focusing on the messages generated by second phase (they are prefixed with either "*" and "#"), what can you say about the behavior of the **`fork()`** system call?

```
[(base) kevin@Kevins-MacBook-Pro-5 Code % gcc ForkTest.c -o "ForkTest"
[(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
PID[4335] | X = 100 | Y = 200 | Z = 300 |
*PID[4335] | X = 100 | Y = 200 | Z = 300 |
#PID[4335] | X = 101 | Y = 202 | Z = 305 |
*PID[4336] | X = 100 | Y = 200 | Z = 300 |
#PID[4336] | X = 101 | Y = 202 | Z = 303 |
```

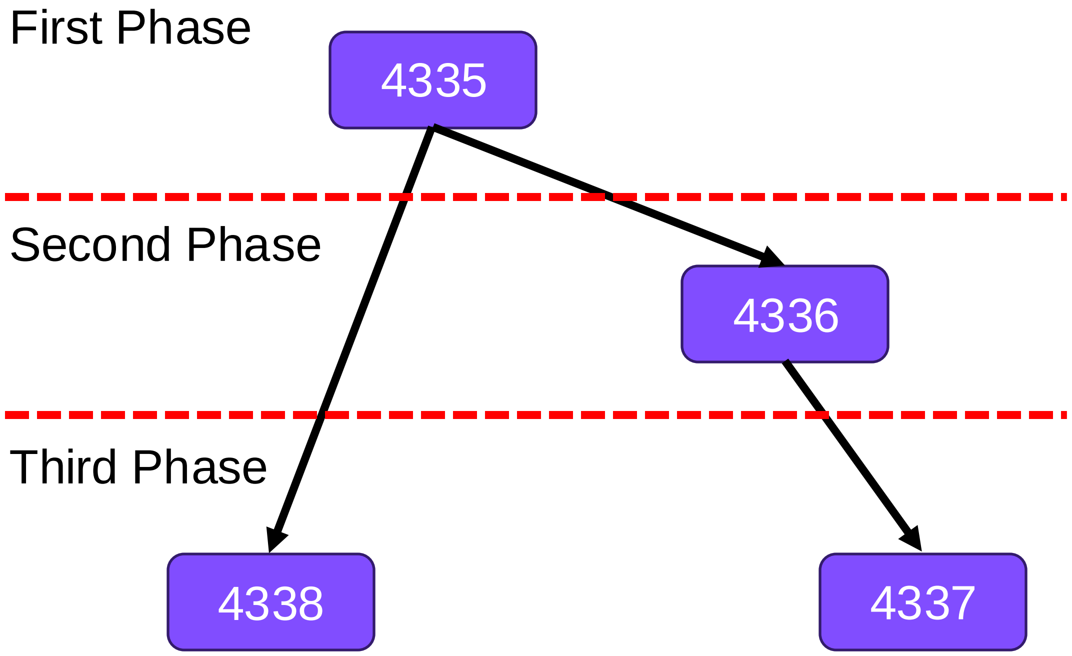
- At the “*” messages, both the parent and child initially have the same value after **`fork()`**.
- The “#” messages show the data items after change.
- Both dataX and dataY have the same value in each of their processes after the change.
- This shows that the processes have independent memory space, because the updates do not affect each other’s memory space.

Question 2(c)

Using the messages seen on your system, draw a process tree to represent the processes generated. Use the process tree to explain the values printed by the child processes.

```
[(base) kevin@Kevins-MacBook-Pro-5 Code % gcc ForkTest.c -o "ForkTest"
[(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
PID[4335] | X = 100 | Y = 200 | Z = 300 |
*PID[4335] | X = 100 | Y = 200 | Z = 300 |
#PID[4335] | X = 101 | Y = 202 | Z = 305 |
*PID[4336] | X = 100 | Y = 200 | Z = 300 |
#PID[4336] | X = 101 | Y = 202 | Z = 303 |
**PID[4336] | X = 101 | Y = 202 | Z = 303 |
##PID[4336] | X = 102 | Y = 204 | Z = 306 |
**PID[4337] | X = 101 | Y = 202 | Z = 303 |
##PID[4337] | X = 102 | Y = 204 | Z = 306 |
**PID[4335] | X = 101 | Y = 202 | Z = 305 |
##PID[4335] | X = 102 | Y = 204 | Z = 308 |
**PID[4338] | X = 101 | Y = 202 | Z = 305 |
##PID[4338] | X = 102 | Y = 204 | Z = 308 |
```

Note: The process IDs you get might be different



Question 2(d)

Do you think it is possible to get different ordering between the output messages, why?

```
[(base) kevin@Kevins-MacBook-Pro-5 Code % gcc ForkTest.c -o "ForkTest"
[(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
PID[18860] | X = 100 | Y = 200 | Z = 300 |
*PID[18860] | X = 100 | Y = 200 | Z = 300 |
#PID[18860] | X = 101 | Y = 202 | Z = 305 |
*PID[18861] | X = 100 | Y = 200 | Z = 300 |
#PID[18861] | X = 101 | Y = 202 | Z = 303 |
**PID[18860] | X = 101 | Y = 202 | Z = 305 |
##PID[18860] | X = 102 | Y = 204 | Z = 308 |
**PID[18861] | X = 101 | Y = 202 | Z = 303 |
##PID[18861] | X = 102 | Y = 204 | Z = 306 |
**PID[18862] | X = 101 | Y = 202 | Z = 305 |
##PID[18862] | X = 102 | Y = 204 | Z = 308 |
**PID[18863] | X = 101 | Y = 202 | Z = 303 |
##PID[18863] | X = 102 | Y = 204 | Z = 306 |
[(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
PID[18867] | X = 100 | Y = 200 | Z = 300 |
*PID[18867] | X = 100 | Y = 200 | Z = 300 |
#PID[18867] | X = 101 | Y = 202 | Z = 305 |
**PID[18867] | X = 101 | Y = 202 | Z = 305 |
##PID[18867] | X = 102 | Y = 204 | Z = 308 |
*PID[18868] | X = 100 | Y = 200 | Z = 300 |
#PID[18868] | X = 101 | Y = 202 | Z = 303 |
**PID[18869] | X = 101 | Y = 202 | Z = 305 |
**PID[18868] | X = 101 | Y = 202 | Z = 303 |
##PID[18869] | X = 102 | Y = 204 | Z = 308 |
##PID[18868] | X = 102 | Y = 204 | Z = 306 |
**PID[18870] | X = 101 | Y = 202 | Z = 303 |
##PID[18870] | X = 102 | Y = 204 | Z = 306 |
(base) kevin@Kevins-MacBook-Pro-5 Code %
```

Question 2(d)

Do you think it is possible to get different ordering between the output messages, why?

- Yes.
- Once the processes are created, they can be independently chosen by the OS to run.
- Depending on the existence of other processes at that time, it is possible that OS chooses differently between runs of the program

```
[(base) kevin@Kevins-MacBook-Pro-5 Code % gcc ForkTest.c -o "ForkTest"
[(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
 PID[18860] | X = 100 | Y = 200 | Z = 300 |
 *PID[18860] | X = 100 | Y = 200 | Z = 300 |
 #PID[18860] | X = 101 | Y = 202 | Z = 305 |
 *PID[18861] | X = 100 | Y = 200 | Z = 300 |
 #PID[18861] | X = 101 | Y = 202 | Z = 303 |
 **PID[18860] | X = 101 | Y = 202 | Z = 305 |
 ##PID[18860] | X = 102 | Y = 204 | Z = 308 |
 **PID[18861] | X = 101 | Y = 202 | Z = 303 |
 ##PID[18861] | X = 102 | Y = 204 | Z = 306 |
 **PID[18862] | X = 101 | Y = 202 | Z = 305 |
 ##PID[18862] | X = 102 | Y = 204 | Z = 308 |
 **PID[18863] | X = 101 | Y = 202 | Z = 303 |
 ##PID[18863] | X = 102 | Y = 204 | Z = 306 |
[(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
 PID[18867] | X = 100 | Y = 200 | Z = 300 |
 *PID[18867] | X = 100 | Y = 200 | Z = 300 |
 #PID[18867] | X = 101 | Y = 202 | Z = 305 |
 **PID[18867] | X = 101 | Y = 202 | Z = 305 |
 ##PID[18867] | X = 102 | Y = 204 | Z = 308 |
 *PID[18868] | X = 100 | Y = 200 | Z = 300 |
 #PID[18868] | X = 101 | Y = 202 | Z = 303 |
 **PID[18869] | X = 101 | Y = 202 | Z = 305 |
 **PID[18868] | X = 101 | Y = 202 | Z = 303 |
 ##PID[18869] | X = 102 | Y = 204 | Z = 308 |
 ##PID[18868] | X = 102 | Y = 204 | Z = 306 |
 **PID[18870] | X = 101 | Y = 202 | Z = 303 |
 ##PID[18870] | X = 102 | Y = 204 | Z = 306 |
 (base) kevin@Kevins-MacBook-Pro-5 Code %
```

Question 2(e)

Can you point how which pair(s) of messages can never swap places? i.e. their relative order is always the same?

```
[(base) kevin@Kevins-MacBook-Pro-5 Code % gcc ForkTest.c -o "ForkTest"
[(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
PID[18860] | X = 100 | Y = 200 | Z = 300 |
*PID[18860] | X = 100 | Y = 200 | Z = 300 |
#PID[18860] | X = 101 | Y = 202 | Z = 305 |
*PID[18861] | X = 100 | Y = 200 | Z = 300 |
#PID[18861] | X = 101 | Y = 202 | Z = 303 |
**PID[18860] | X = 101 | Y = 202 | Z = 305 |
##PID[18860] | X = 102 | Y = 204 | Z = 308 |
**PID[18861] | X = 101 | Y = 202 | Z = 303 |
##PID[18861] | X = 102 | Y = 204 | Z = 306 |
**PID[18862] | X = 101 | Y = 202 | Z = 305 |
##PID[18862] | X = 102 | Y = 204 | Z = 308 |
**PID[18863] | X = 101 | Y = 202 | Z = 303 |
##PID[18863] | X = 102 | Y = 204 | Z = 306 |
[(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
PID[18867] | X = 100 | Y = 200 | Z = 300 |
*PID[18867] | X = 100 | Y = 200 | Z = 300 |
#PID[18867] | X = 101 | Y = 202 | Z = 305 |
**PID[18867] | X = 101 | Y = 202 | Z = 305 |
##PID[18867] | X = 102 | Y = 204 | Z = 308 |
*PID[18868] | X = 100 | Y = 200 | Z = 300 |
#PID[18868] | X = 101 | Y = 202 | Z = 303 |
**PID[18869] | X = 101 | Y = 202 | Z = 305 |
**PID[18868] | X = 101 | Y = 202 | Z = 303 |
##PID[18869] | X = 102 | Y = 204 | Z = 308 |
##PID[18868] | X = 102 | Y = 204 | Z = 306 |
**PID[18870] | X = 101 | Y = 202 | Z = 303 |
##PID[18870] | X = 102 | Y = 204 | Z = 306 |
(base) kevin@Kevins-MacBook-Pro-5 Code %
```

Question 2(e)

Can you point how which pair(s) of messages can never swap places? i.e. their relative order is always the same?

- "*" and "#" messages from the same process can never change place as sequential ordering is still preserved in the same process.
- Likewise, messages from the same process will always follow the phases, i.e. "*", "#" before "**" and "##".
- Message from the first phase (only one) must precede all other messages. This is obviously correct as there is only one process executing at that time.

```
(base) kevin@Kevins-MacBook-Pro-5 Code % gcc ForkTest.c -o "ForkTest"
(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
PID[18860] | X = 100 | Y = 200 | Z = 300 |
*PID[18860] | X = 100 | Y = 200 | Z = 300 |
#PID[18860] | X = 101 | Y = 202 | Z = 305 |
*PID[18861] | X = 100 | Y = 200 | Z = 300 |
#PID[18861] | X = 101 | Y = 202 | Z = 303 |
**PID[18860] | X = 101 | Y = 202 | Z = 305 |
##PID[18860] | X = 102 | Y = 204 | Z = 308 |
**PID[18861] | X = 101 | Y = 202 | Z = 303 |
##PID[18861] | X = 102 | Y = 204 | Z = 306 |
**PID[18862] | X = 101 | Y = 202 | Z = 305 |
##PID[18862] | X = 102 | Y = 204 | Z = 308 |
**PID[18863] | X = 101 | Y = 202 | Z = 303 |
##PID[18863] | X = 102 | Y = 204 | Z = 306 |
(base) kevin@Kevins-MacBook-Pro-5 Code % ./ForkTest
PID[18867] | X = 100 | Y = 200 | Z = 300 |
*PID[18867] | X = 100 | Y = 200 | Z = 300 |
#PID[18867] | X = 101 | Y = 202 | Z = 305 |
**PID[18867] | X = 101 | Y = 202 | Z = 305 |
##PID[18867] | X = 102 | Y = 204 | Z = 308 |
*PID[18868] | X = 100 | Y = 200 | Z = 300 |
#PID[18868] | X = 101 | Y = 202 | Z = 303 |
**PID[18869] | X = 101 | Y = 202 | Z = 305 |
**PID[18868] | X = 101 | Y = 202 | Z = 303 |
##PID[18869] | X = 102 | Y = 204 | Z = 308 |
##PID[18868] | X = 102 | Y = 204 | Z = 306 |
**PID[18870] | X = 101 | Y = 202 | Z = 303 |
##PID[18870] | X = 102 | Y = 204 | Z = 306 |
(base) kevin@Kevins-MacBook-Pro-5 Code %
```

Question 2(f)

- How does this change the ordering of the output messages?

C code:

```c
int dataX = 100;

int main( )
{
    pid_t childPID;

    int dataY = 200;
    int* dataZptr = (int*) malloc(sizeof(int));
    *dataZptr = 300;

    printf("PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);
```

First

```c
    childPID = fork();
    printf("*PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    dataX += 1;
    dataY += 2;
    (*dataZptr) += 3;
    printf("#PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);
```

Second

Code Insertion Point

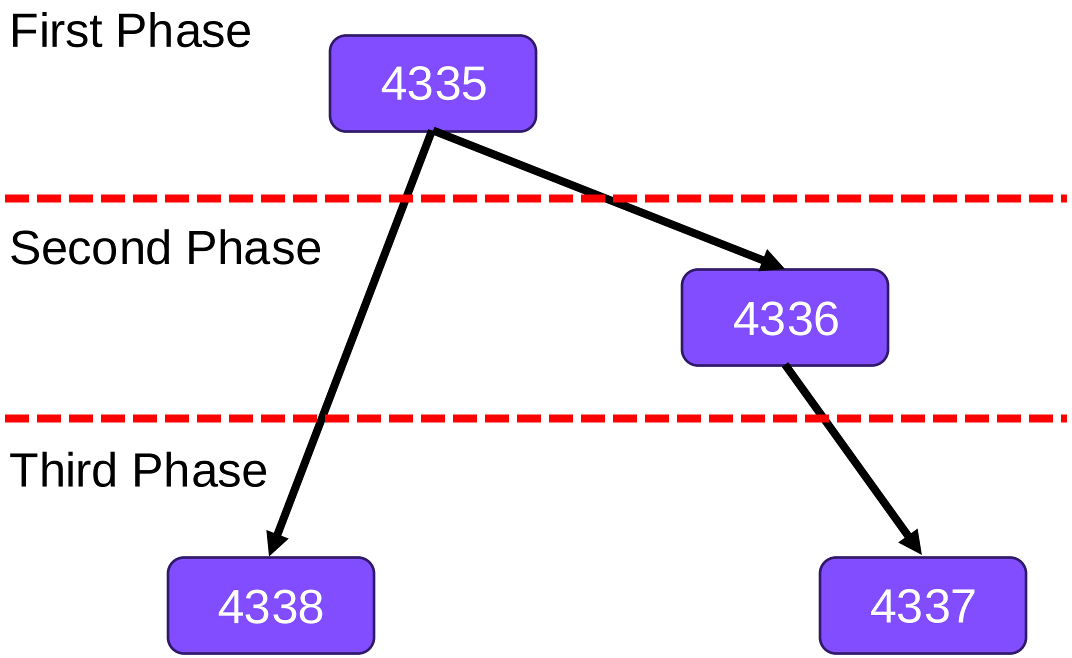
```c
if (childPID == 0){
    sleep(5);
}
```

```c
    childPID = fork();
    printf("**PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    dataX += 1;
    dataY += 2;
    (*dataZptr) += 3;
    printf("##PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    return 0;
}
```

Third

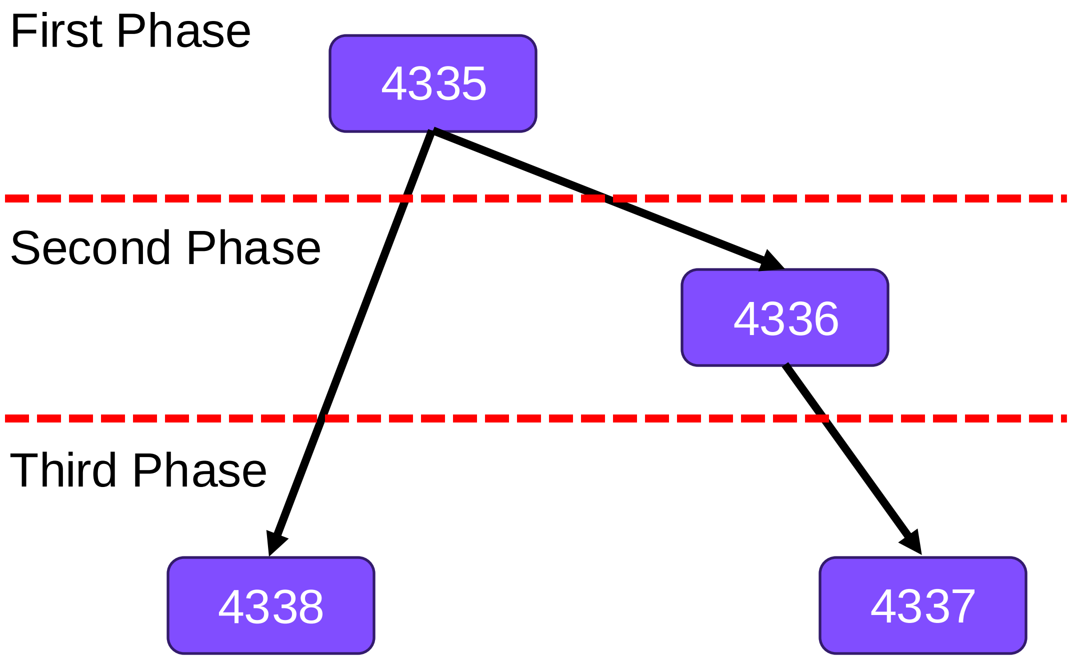
Question 2(f)

- The inserted code "pause" the first child process (i.e. 4336) for 5 seconds.
- So, if we assume process 4338 takes less than 5 seconds to create and run, then it is likely that both process 4335 and 4338 will finish execution before 4336 and 4337.
- **However, this is not deterministic as messages from both branches 4335 and 4336 can still mix**
- **Let's see an example where we add another sleep statement before we add values to dataX, dataY and dataZptr in phase 3.**



Question 2(f)

```
PID[4335] | X = 100 | Y = 200 | Z = 300 |
*PID[4335] | X = 100 | Y = 200 | Z = 300 |
#PID[4335] | X = 101 | Y = 202 | Z = 305 |
**PID[4335] | X = 101 | Y = 202 | Z = 305 |
*PID[4336] | X = 100 | Y = 200 | Z = 300 |
#PID[4336] | X = 101 | Y = 202 | Z = 303 |
**PID[4338] | X = 101 | Y = 202 | Z = 305 |
##PID[4338] | X = 102 | Y = 204 | Z = 308 |
**PID[4336] | X = 101 | Y = 202 | Z = 303 |
**PID[4337] | X = 101 | Y = 202 | Z = 303 |
##PID[4335] | X = 102 | Y = 204 | Z = 308 |
##PID[4336] | X = 102 | Y = 204 | Z = 306 |
##PID[4337] | X = 102 | Y = 204 | Z = 306 |
```



102
204
308

102
204
306

Question 2(g)

- How does this change the ordering of the output messages?

C code:

```
int dataX = 100;

int main( )
{
    pid_t childPID;

    int dataY = 200;
    int* dataZptr = (int*) malloc(sizeof(int));
    *dataZptr = 300;

    printf("PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);
```

First

```
    childPID = fork();
    printf("*PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    dataX += 1;
    dataY += 2;
    (*dataZptr) += 3;
    printf("#PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);
```

Second

Code Insertion Point

```
if (childPID != 0){
      wait(NULL);
}
```

```
    childPID = fork();
    printf("**PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    dataX += 1;
    dataY += 2;
    (*dataZptr) += 3;
    printf("##PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    return 0;
}
```

Third

Question 2(g)

Minimally, all second and third messages for process 4336 must be printed for process 4335 to resume from the third phase to spawn process 4337

(1) The inserted code will pause process 4335 after printing the “*” and “#” messages.

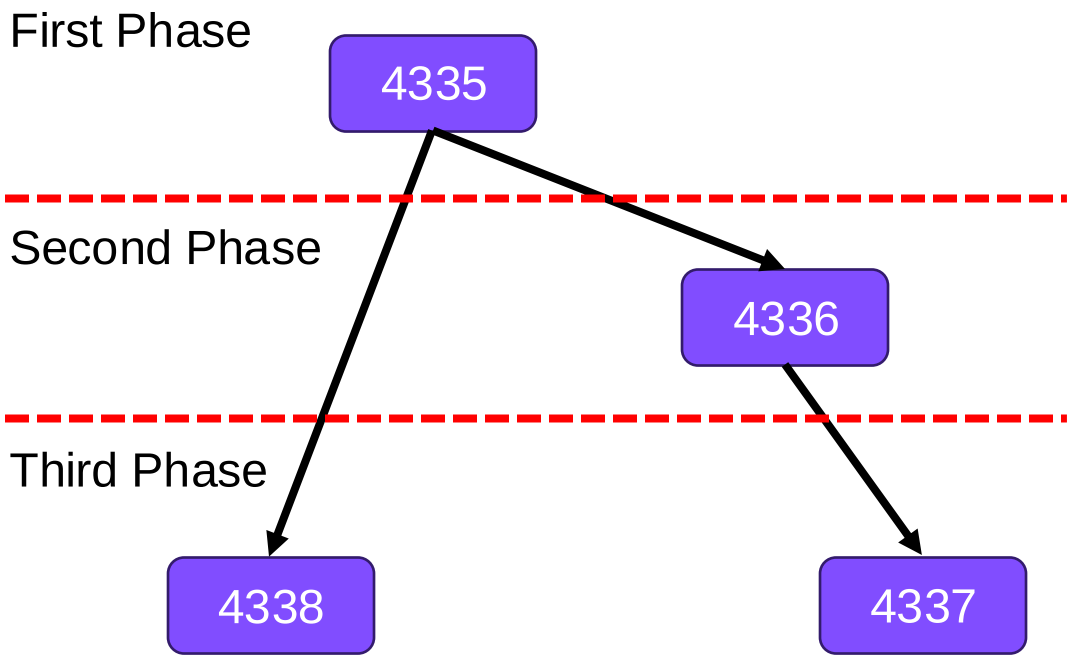
(2) Process 4336 will carry on to spawn its child (process 4337).

(3) 4336 will continue to print its ** and ## messages and exit.

(4) Once 4336 exits, the wait in 4335 will also exit, and 4335 can continue to spawn 4338

(5) 4335 and 4338 will print their ** and ## messages independently.

(6) Note also that 4337 will continue printing ** and ## messages independently of what’s happening in steps (4) and (5) above.



The difference to (f) is that this is deterministic regardless of how long process 4337/4338 takes to finish its execution.

Question 3

Question 3(i)

- Modify only Parallel.c such that we can now initiate prime factorization on [1-9] user inputs simultaneously.
- More importantly, we want to report result as soon as they are ready **regardless of the user input order**.

PrimeFactor.c

```
gcc PrimeFactor.c -o "PF"
```

```c
int main( int argc, char* argv[])
{
    int nFactor = 0, userInput, factor;

    //Convert string to number
    userInput = atoi( argv[1] );

    nFactor = 0;
    factor = 2;

    //quick hack to get the number of prime factors
    // only for positive integer
    while (userInput > 1){
        if (userInput % factor == 0){
            userInput /= factor;
            nFactor++;
        } else {
            factor++;
        }
    }

    return nFactor;
}
```

Parallel.c

gcc Parallel.c

```
int main()
{
    int userInput, childPid, childResult;
    //Since largest number is 10 digits, a 12 characters string is more
    //than enough
    char cStringExample[12];

    scanf("%d", &userInput);

    childPid = fork();

    if (childPid != 0 ){
        wait( &childResult);
        printf("%d has %d prime factors\n", userInput,
                WEXITSTATUS(childResult));

    } else {
        //Easy way to convert a number into a string
        sprintf(cStringExample, "%d", userInput);

        execl("./PF", "PF", cStringExample, NULL);
    }

}
```

Solved Parallel.c

```c
int main()
{
    int i, j, userInput[9], nInput, childPid[9], childResult, pid;
    char cStringExample[12];
    scanf("%d", &nInput);

    for (i = 0; i < nInput; i++){
        scanf("%d", &userInput[i]);

        childPid[i] = fork();
        if (childPid[i] == 0){
            sprintf(cStringExample, "%d", userInput[i]);
            execl("./PF", "PF", cStringExample, NULL);
            return 0; //Redundant. Everything from here downwards
                           // is replaced by PF in the child.
        }
    }
    for (i = 0; i < nInput; i++){
        pid = wait( &childResult );

        //match pid with child pid
        for (j = 0; j < nInput; j++){
            if (pid == childPid[j])
                break;
        }
         //Special note: Original solution used childresult >> 8. Here
         // we use the official WEXITSTATUS macro to ensure portability.
        printf("%d has %d prime factors\n", userInput[j],
                    WEXITSTATUS(childresult));
    }
}
```

Question 3(ii)

After you have solved the problem, find a way to change your **`wait()`** to **`waitpid()`**, what do you think is the effect of this change?

```
int main()
{
    int i, userInput[9], nInput, childPid[9], childResult;
    char cStringExample[12];
    scanf("%d", &nInput);

    for (i = 0; i < nInput; i++){
        scanf("%d", &userInput[i]);

        childPid[i] = fork();

        if (childPid[i] == 0){
            sprintf(cStringExample, "%d", userInput[i]);
            execl("./PF", "PF", cStringExample, NULL);
            return 0;
        }
    }
    for (i = 0; i < nInput; i++){
        waitpid(childPid[i], &childResult, 0);
        printf("%d has %d prime factors\n", userInput[i], WEXITSTATUS(childresult));
    }
}
```

Question 3(ii)

After you have solved the problem, find a way to change your **`wait()`** to **`waitpid()`**, what do you think is the effect of this change?

The change of **`wait()`** to **`waitpid()`** forces the main process to wait for the child process in the creation order of the child processes. This means that the results are returned according to the user input order.

```
(base) kevin@Kevins-MacBook-Pro-5 Code % ./Parallel
5
1987
8
236
5
10
1987 has 1 prime factors
8 has 3 prime factors
236 has 3 prime factors
5 has 1 prime factors
10 has 2 prime factors
(base) kevin@Kevins-MacBook-Pro-5 Code %
```