CS2106 Tutorial 2

T01/T02

Quick Recap

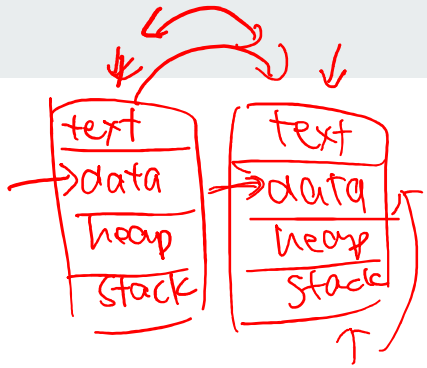
`fork()`

`fork()` creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

Return Value:

- On success, parent process gets the PID of the child process, child process gets 0.
- On failure, parent process gets -1, no child process is created.

More on `fork()`



- The entire virtual address space of the parent is replicated in the child
- The child process and the parent process run in separate memory spaces.
- Both parent and child processes continue executing after `fork()`
  - PC is also duplicated in child process
- Use the return value of `fork()` to distinguish parent and child
  - Parent process: `if (pid != 0)`
  - Child process: `if (pid == 0)`

exec()

exec() replaces current executing image with a new one.

Return only if an error has occurred.

fork() + exec()

By combining the two mechanisms, we can:

- Spawn off a child process
  - Let the child process perform a task through exec()
- Meanwhile, the parent process is still around
  - To accept another request

This combination of mechanisms is the main way in Unix:

- To get a new process for running a new program

`exit()`

exit() causes normal process termination

Status is returned to the parent process

- 0 = Normal Termination (successful execution)
- !0 = To indicate problematic execution

The function does not return

More on `exit()`

- main() implicitly calls exit()
- Most system resources used by process are released on exit()
  - E.g. File descriptors
- Some basic process resources are not releasable
  - E.g. PID & process status
  - Process control block may be still needed

wait()

- wait() suspends the calling thread until one of its children terminates.
- waitpid() suspends the calling process until a child specified by pid argument has changed state.

- wait() allows the calling process to release the resources associated with the child; if it is not performed, then the terminated child remains in a zombie state.
- On success, returns the process ID of the terminated child; on error, -1 is returned.

More on wait()

- If there are no children, wait() does not block (as stated in question 1). Process will just continue running
- Cleans up remainder of child resources

Zombie and Orphan

Zombie: Child terminates before parent but parent did not call wait()

- Child process become a zombie process
- Can exhaust process table

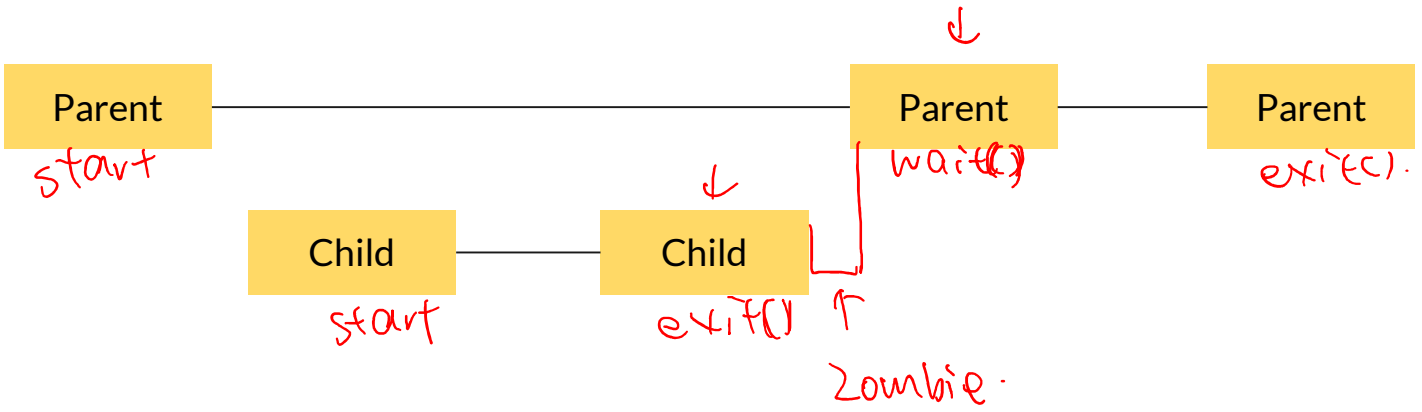
Orphan: Parent terminates before child

- init() process becomes pseudo parent of child process
- init() uses wait() to clean up child process

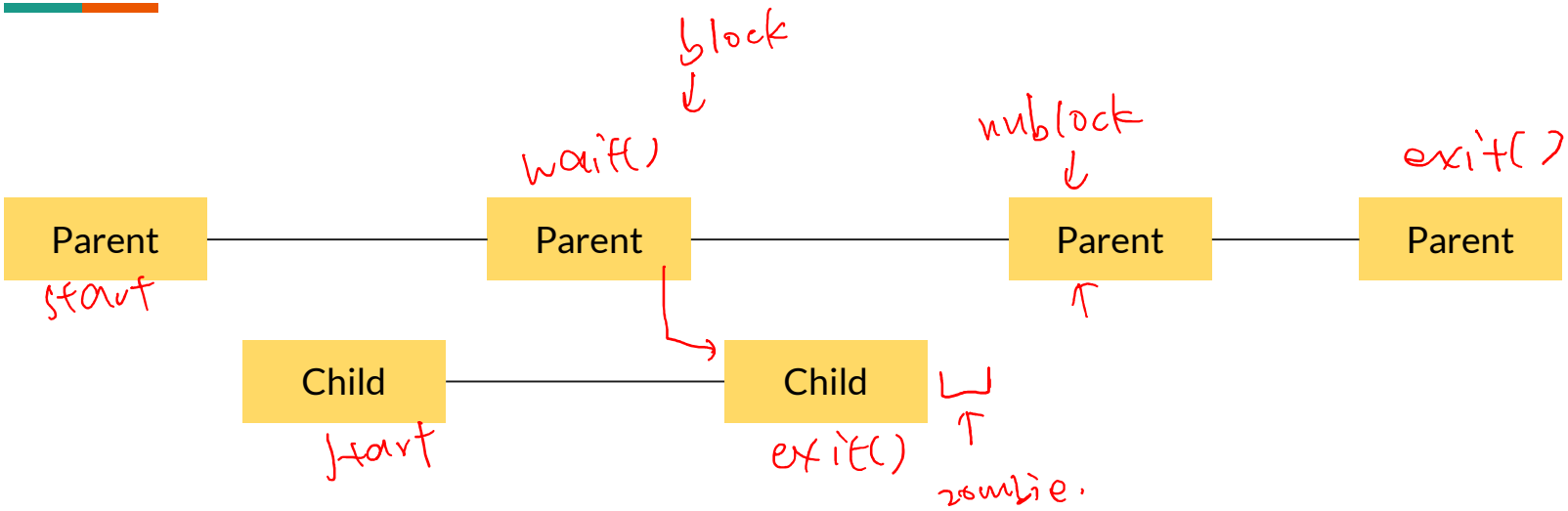
More on zombies

- Data such as exit status may be needed, so process entry will be kept in PCB
- Process always has a parent
- If the parent terminated, *init* will become the pseudo-parent and help to clean

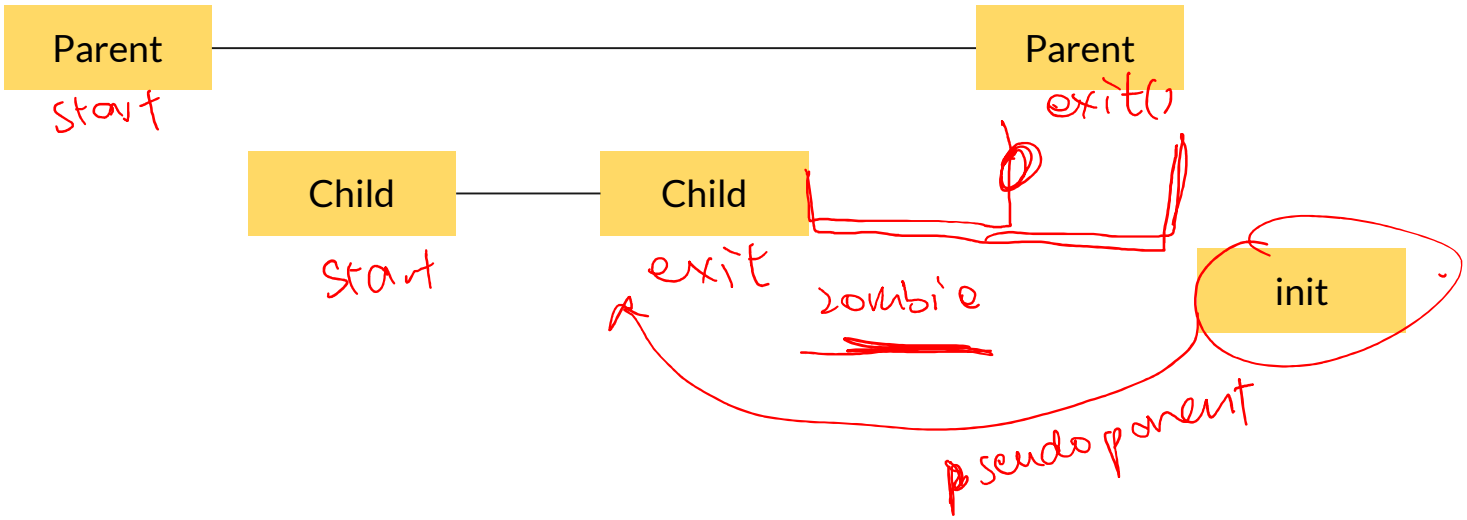
Some cases for wait() and zombies



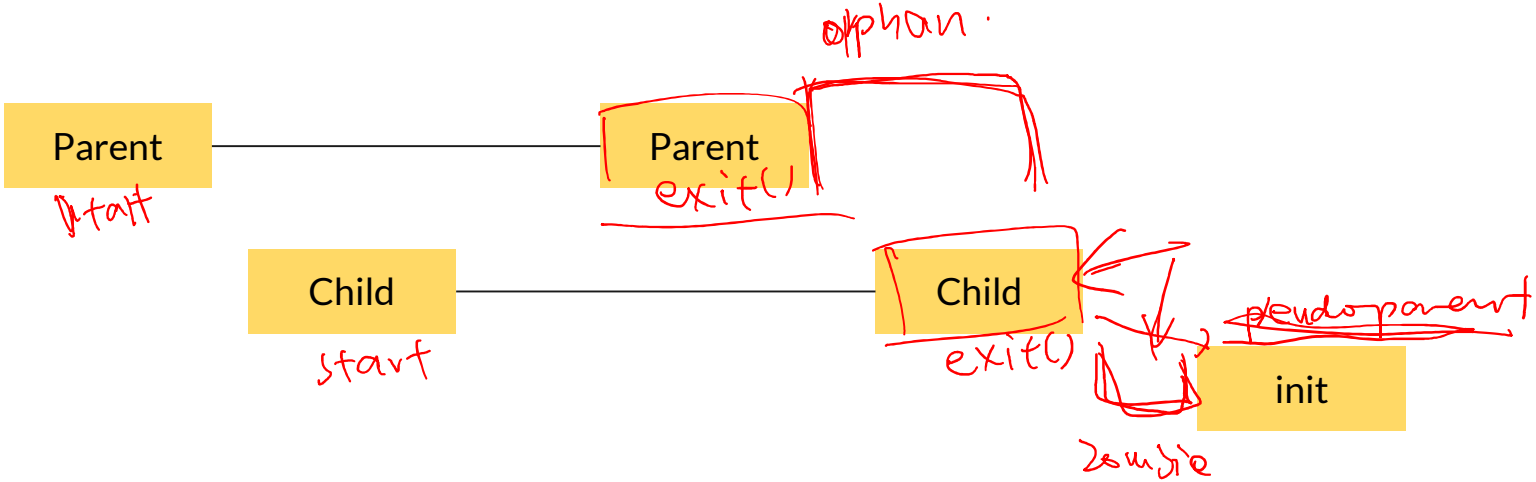
Some cases for wait() and zombies



Some cases for wait() and zombies



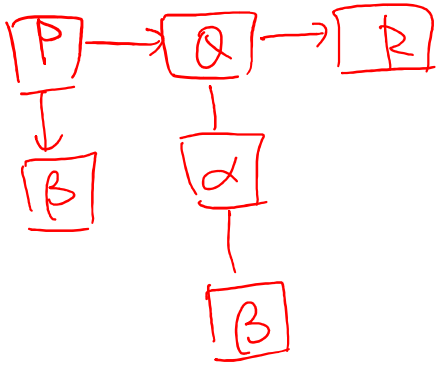
Some cases for wait() and zombies



Tutorial Questions

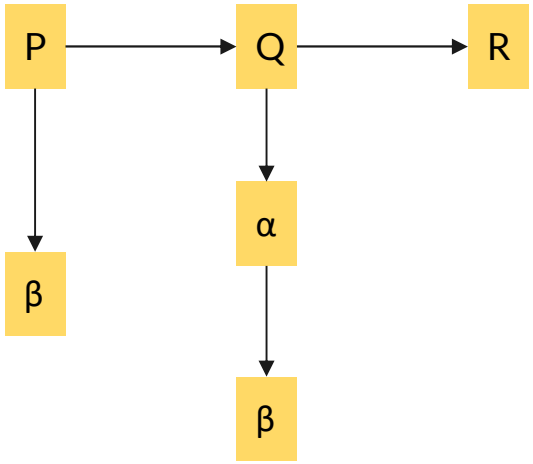
Question 1

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

    return 0;
}
```



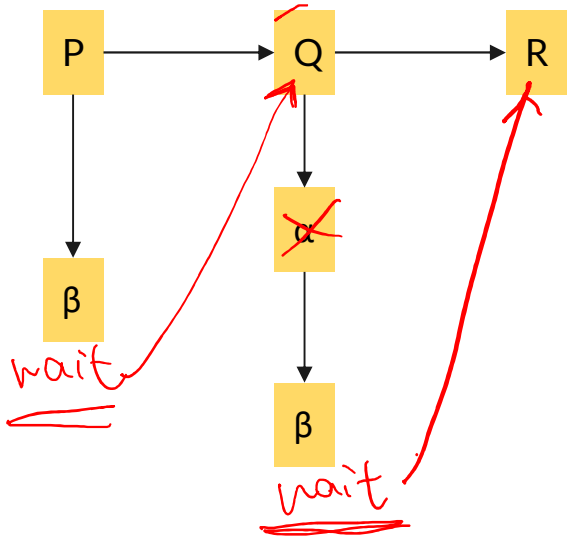
Question 1

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

    return 0;
}
```



Question 1a

```
int main( ) {
    // This is process P
    if ( fork() == 0 ){
        // This is process Q
        if ( fork() == 0 ) {
            // This is process R
            ......
            return 0;
        }
        <nothing>
    }
    wait(NULL);

    return 0;
}
```

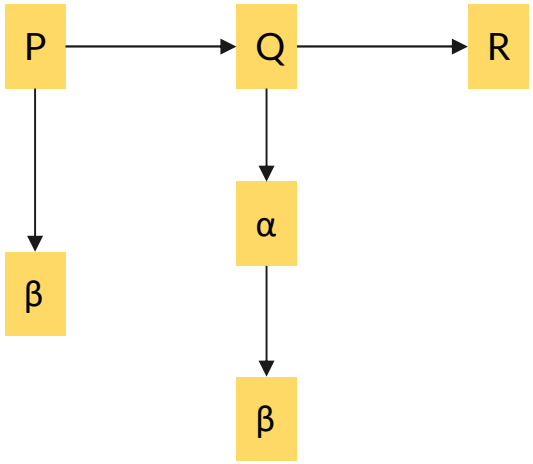


Process Q *always* terminate before P.

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

Question 1a

```
int main( ) {
    // This is process P
    if ( fork() == 0 ){
        // This is process Q
        if ( fork() == 0 ) {
            // This is process R
            ......
            return 0;
        }
        <nothing>
    }
    wait(NULL);

    return 0;
}
```

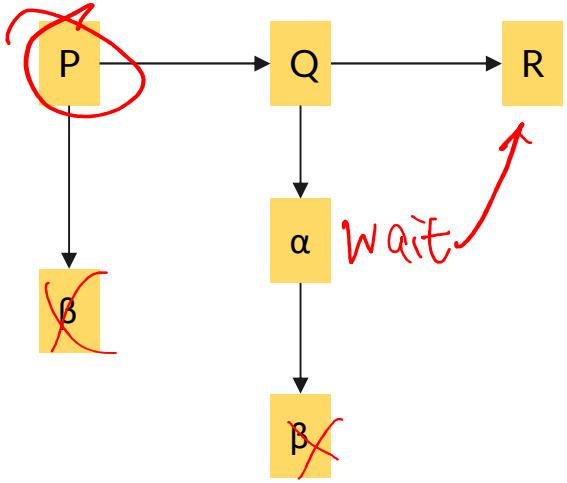


Process Q *always* terminate before P.
Process R can terminate at any time w.r.t. P and Q.

False: Q waits for R

Question 1b

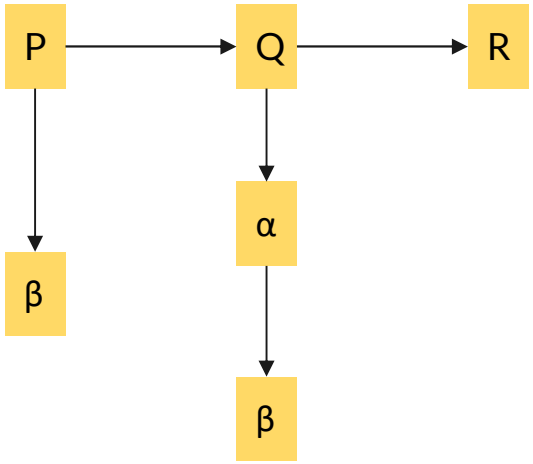
```
int main( ) {
    // This is process P
    if ( fork() == 0 ){
        // This is process Q
        if ( fork() == 0 ) {
            // This is process R
            ......
            return 0;
        }
        wait(NULL);
    }
    <nothing>

    return 0;
}
```



Process Q *always* terminate before P.
Process R can terminate at any time w.r.t. P and Q.

Question 1b

```
int main( ) {
    // This is process P
    if ( fork() == 0 ){
        // This is process Q
        if ( fork() == 0 ) {
            // This is process R
            ......
            return 0;
        }
        wait(NULL);
    }
    <nothing>

    return 0;
}
```

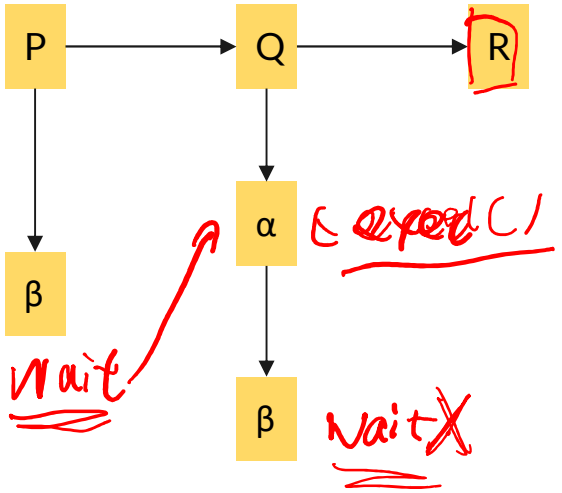


Process Q *always* terminate before P.
Process R can terminate at any time w.r.t. P and Q.

False: Q waits for R and P don't wait

Question 1c

```
int main( ) {
    // This is process P
    if ( fork() == 0 ){
        // This is process Q
        if ( fork() == 0 ) {
            // This is process R
            ......
            return 0;
        }
        execl(..);
    }
    wait(NULL);

    return 0;
}
```

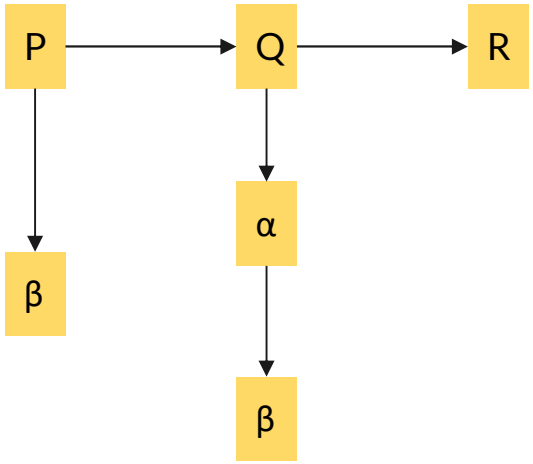


Process Q *always* terminate before P.

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

Question 1c

```
int main( ) {
    // This is process P
    if ( fork() == 0 ){
        // This is process Q
        if ( fork() == 0 ) {
            // This is process R
            ......
            return 0;
        }
        execl(..);
    }
    wait(NULL);

    return 0;
}
```

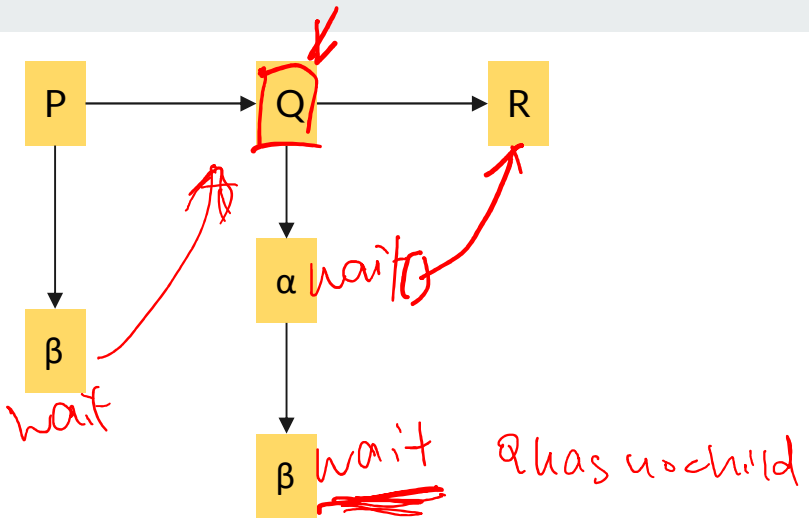


Process Q *always* terminate before P.
Process R can terminate at any time w.r.t. P and Q.

True: P wait for Q even though Q is now a "new" executable.
Q doesn't wait because execl() replace the entire context (text, data, heap, stack)

Question 1d

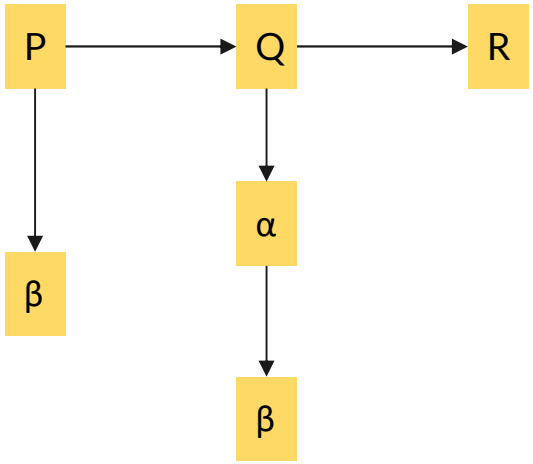
```
int main( ) {
    // This is process P
    if ( fork() == 0 ){
        // This is process Q
        if ( fork() == 0 ) {
            // This is process R
            ......
            return 0;
        }
        wait(NULL);
    }
    wait(NULL);

    return 0;
}
```



Process P never terminates.

Question 1d

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

    return 0;
}
```



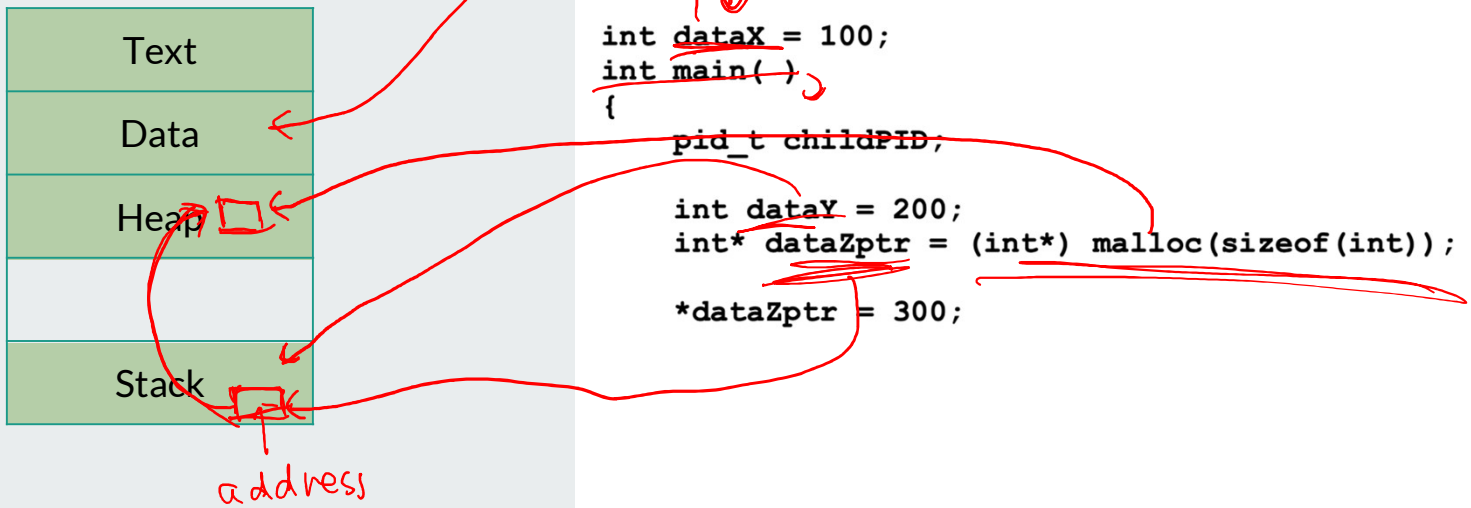
Process P never terminates.

False: Although Q has an additional wait, the wait will return immediately as there is no child.
wait() doesn't block on grand children.

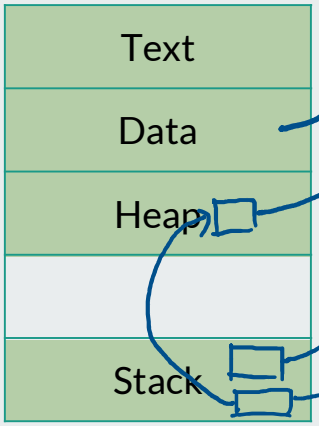
Question 2

```
int dataX = 100;
int main( )
{
    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;
    (*dataZptr) += 3;
    printf("#PID[%d] | X = %d | Y = %d | Z = %d |\n",
            getpid(),  dataX, dataY, *dataZptr);

    //Insertion Point (for parts (g), (h))

    //Third Phase
    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;
}
```

Question 2a

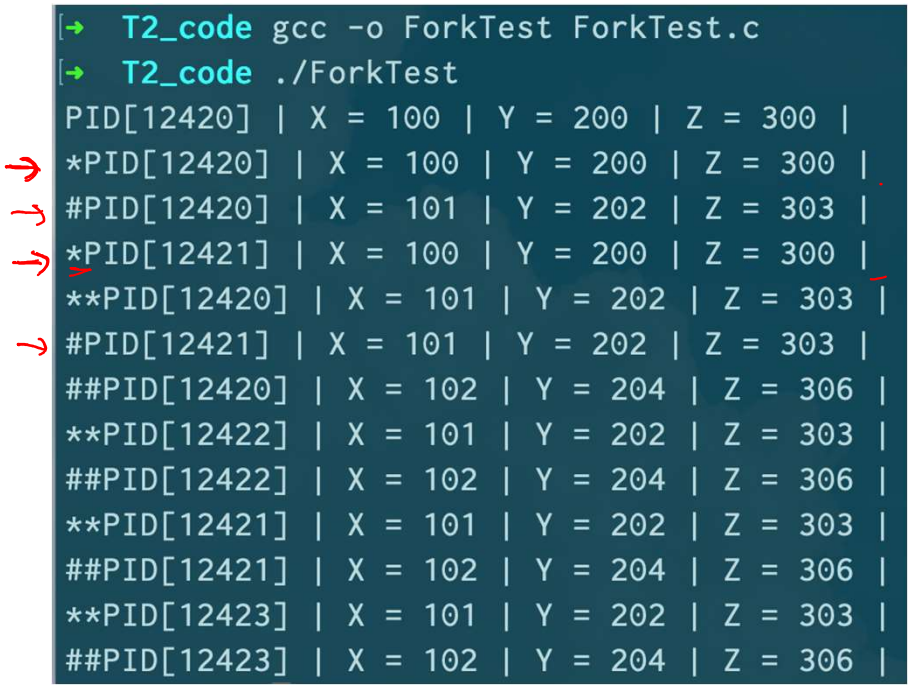

Text
Data
Heap
Stack
address
global var
int dataX = 100;
int main( )
{
pid_t childPID;
int dataY = 200;
int* dataZptr = (int*) malloc(sizeof(int));
*dataZptr = 300;

Question 2a




```
int dataX = 100;
int main( )
{
    pid_t childPID;

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

    *dataZptr = 300;
```

Question 2b


T2_code gcc -o ForkTest ForkTest.c
T2_code ./ForkTest
PID[12420] | X = 100 | Y = 200 | Z = 300 |
*PID[12420] | X = 100 | Y = 200 | Z = 300 |
#PID[12420] | X = 101 | Y = 202 | Z = 303 |
*PID[12421] | X = 100 | Y = 200 | Z = 300 |
**PID[12420] | X = 101 | Y = 202 | Z = 303 |
#PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12420] | X = 102 | Y = 204 | Z = 306 |
**PID[12422] | X = 101 | Y = 202 | Z = 303 |
##PID[12422] | X = 102 | Y = 204 | Z = 306 |
**PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12421] | X = 102 | Y = 204 | Z = 306 |
**PID[12423] | X = 101 | Y = 202 | Z = 303 |
##PID[12423] | X = 102 | Y = 204 | Z = 306 |

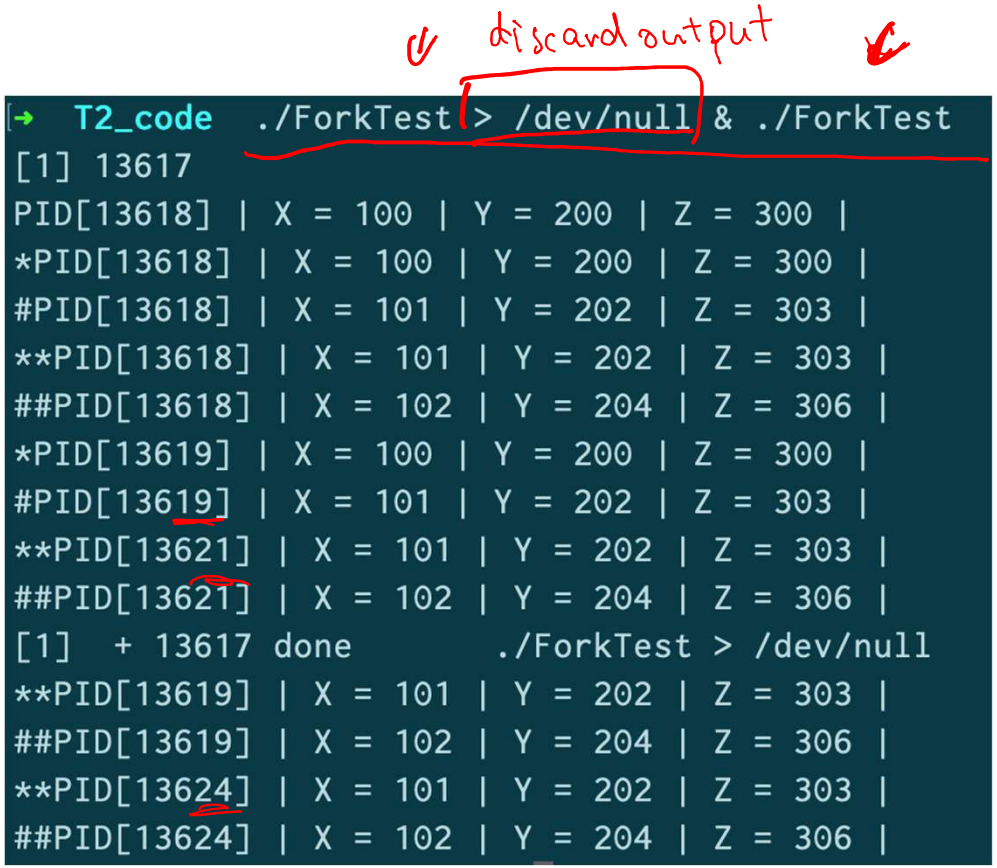
Question 2b

- Every process in the same phase prints out the same values.
- Once the process begins, each process is running completely independently of the others from when it begins to when it exits.

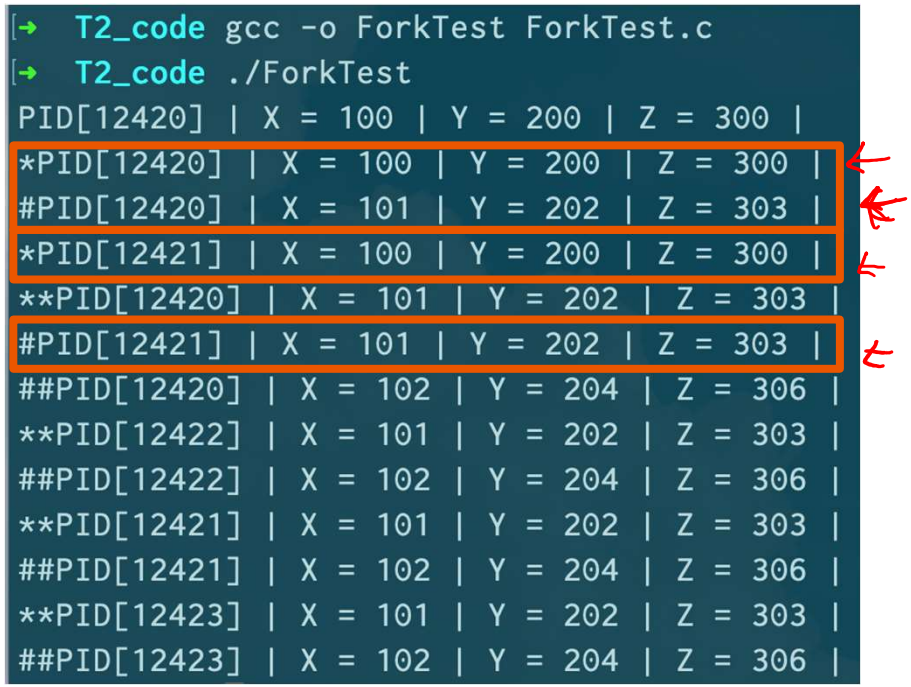
```
[➜  T2_code gcc -o ForkTest ForkTest.c
[➜  T2_code ./ForkTest
PID[12420] | X = 100 | Y = 200 | Z = 300 |
*PID[12420] | X = 100 | Y = 200 | Z = 300 |
#PID[12420] | X = 101 | Y = 202 | Z = 303 |
*PID[12421] | X = 100 | Y = 200 | Z = 300 |
**PID[12420] | X = 101 | Y = 202 | Z = 303 |
#PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12420] | X = 102 | Y = 204 | Z = 306 |
**PID[12422] | X = 101 | Y = 202 | Z = 303 |
##PID[12422] | X = 102 | Y = 204 | Z = 306 |
**PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12421] | X = 102 | Y = 204 | Z = 306 |
**PID[12423] | X = 101 | Y = 202 | Z = 303 |
##PID[12423] | X = 102 | Y = 204 | Z = 306 |
```

Question 2b

The PIDs appear to be assigned sequentially, but it depends on the number of new processes being created at the same time.



Question 2c


[➜ T2_code gcc -o ForkTest ForkTest.c
[➜ T2_code ./ForkTest
PID[12420] | X = 100 | Y = 200 | Z = 300 |
*PID[12420] | X = 100 | Y = 200 | Z = 300 |
#PID[12420] | X = 101 | Y = 202 | Z = 303 |
*PID[12421] | X = 100 | Y = 200 | Z = 300 |
**PID[12420] | X = 101 | Y = 202 | Z = 303 |
#PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12420] | X = 102 | Y = 204 | Z = 306 |
**PID[12422] | X = 101 | Y = 202 | Z = 303 |
##PID[12422] | X = 102 | Y = 204 | Z = 306 |
**PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12421] | X = 102 | Y = 204 | Z = 306 |
**PID[12423] | X = 101 | Y = 202 | Z = 303 |
##PID[12423] | X = 102 | Y = 204 | Z = 306 |

Question 2c

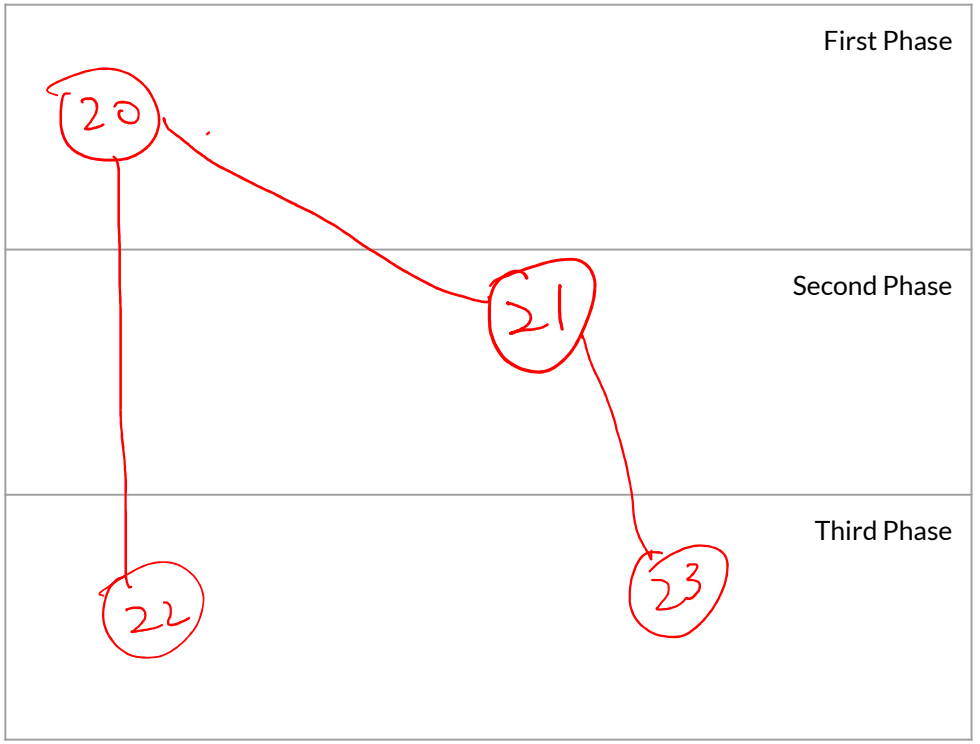
- All 3 data items are duplicated.
- Both the parent and child process has the same values after `fork()`.

- Parent & child have independent memory space.
- Updates do not impact each other's memory space.

```
[→  T2_code gcc -o ForkTest ForkTest.c
[→  T2_code ./ForkTest
PID[12420] | X = 100 | Y = 200 | Z = 300 |
*PID[12420] | X = 100 | Y = 200 | Z = 300 |
#PID[12420] | X = 101 | Y = 202 | Z = 303 |
*PID[12421] | X = 100 | Y = 200 | Z = 300 |
**PID[12420] | X = 101 | Y = 202 | Z = 303 |
#PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12420] | X = 102 | Y = 204 | Z = 306 |
**PID[12422] | X = 101 | Y = 202 | Z = 303 |
##PID[12422] | X = 102 | Y = 204 | Z = 306 |
**PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12421] | X = 102 | Y = 204 | Z = 306 |
**PID[12423] | X = 101 | Y = 202 | Z = 303 |
##PID[12423] | X = 102 | Y = 204 | Z = 306 |
```

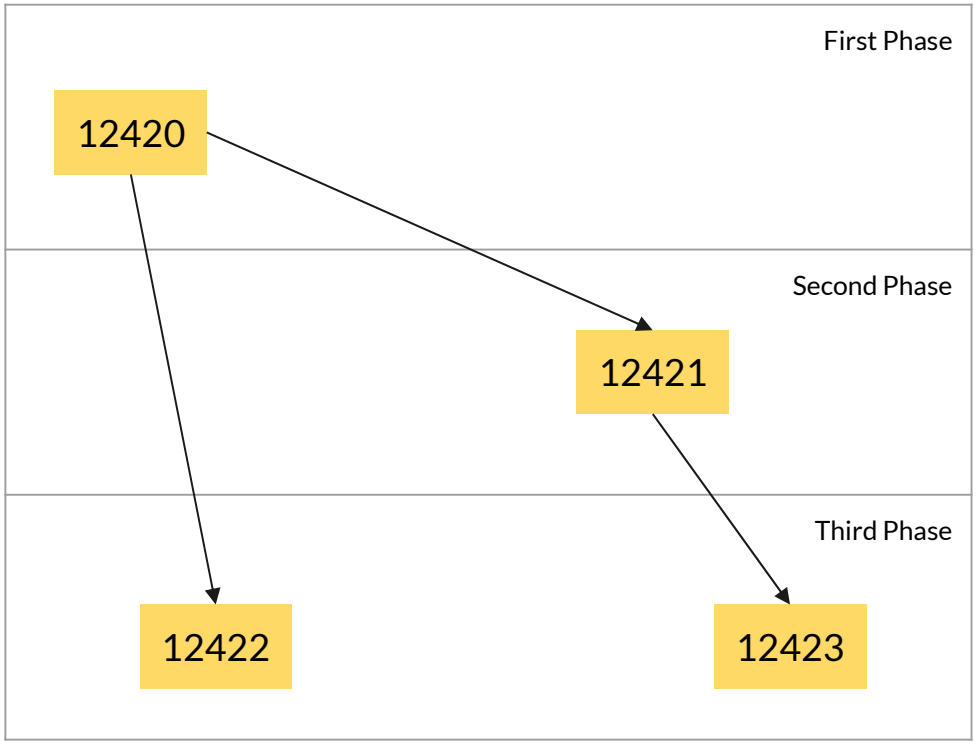
Question 2d

```
[➜  T2_code gcc -o ForkTest ForkTest.c
[➜  T2_code ./ForkTest
PID[12420] | X = 100 | Y = 200 | Z = 300 |
*PID[12420] | X = 100 | Y = 200 | Z = 300 |
#PID[12420] | X = 101 | Y = 202 | Z = 303 |
*PID[12421] | X = 100 | Y = 200 | Z = 300 |
**PID[12420] | X = 101 | Y = 202 | Z = 303 |
#PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12420] | X = 102 | Y = 204 | Z = 306 |
**PID[12422] | X = 101 | Y = 202 | Z = 303 |
##PID[12422] | X = 102 | Y = 204 | Z = 306 |
**PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12421] | X = 102 | Y = 204 | Z = 306 |
**PID[12423] | X = 101 | Y = 202 | Z = 303 |
##PID[12423] | X = 102 | Y = 204 | Z = 306 |
```



Question 2d

```
[➜  T2_code gcc -o ForkTest ForkTest.c
[➜  T2_code ./ForkTest
PID[12420] | X = 100 | Y = 200 | Z = 300 |
*PID[12420] | X = 100 | Y = 200 | Z = 300 |
#PID[12420] | X = 101 | Y = 202 | Z = 303 |
*PID[12421] | X = 100 | Y = 200 | Z = 300 |
**PID[12420] | X = 101 | Y = 202 | Z = 303 |
#PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12420] | X = 102 | Y = 204 | Z = 306 |
**PID[12422] | X = 101 | Y = 202 | Z = 303 |
##PID[12422] | X = 102 | Y = 204 | Z = 306 |
**PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12421] | X = 102 | Y = 204 | Z = 306 |
**PID[12423] | X = 101 | Y = 202 | Z = 303 |
##PID[12423] | X = 102 | Y = 204 | Z = 306 |
```



Note that all 4 processes are alive at the last stage.

Question 2e

Is it possible to get different ordering between the output messages?

Question 2e

Is it possible to get different ordering between the output messages?

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 choose differently between runs of the program.

```
→  T2_code ./ForkTest
PID[12144] | X = 100 | Y = 200 | Z = 300 |
*PID[12144] | X = 100 | Y = 200 | Z = 300 |
#PID[12144] | X = 101 | Y = 202 | Z = 303 |
*PID[12145] | X = 100 | Y = 200 | Z = 300 |
#PID[12145] | X = 101 | Y = 202 | Z = 303 |
**PID[12144] | X = 101 | Y = 202 | Z = 303 |
##PID[12144] | X = 102 | Y = 204 | Z = 306 |
**PID[12145] | X = 101 | Y = 202 | Z = 303 |
**PID[12146] | X = 101 | Y = 202 | Z = 303 |
##PID[12146] | X = 102 | Y = 204 | Z = 306 |
##PID[12145] | X = 102 | Y = 204 | Z = 306 |
**PID[12147] | X = 101 | Y = 202 | Z = 303 |
##PID[12147] | X = 102 | Y = 204 | Z = 306 |
```

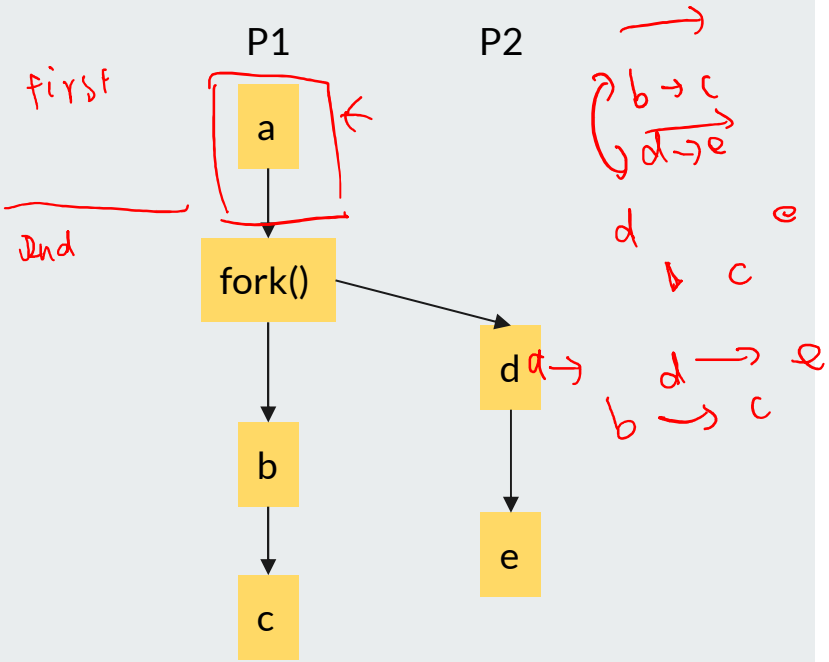
```
→  T2_code ./ForkTest
PID[12420] | X = 100 | Y = 200 | Z = 300 |
*PID[12420] | X = 100 | Y = 200 | Z = 300 |
#PID[12420] | X = 101 | Y = 202 | Z = 303 |
*PID[12421] | X = 100 | Y = 200 | Z = 300 |
**PID[12420] | X = 101 | Y = 202 | Z = 303 |
#PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12420] | X = 102 | Y = 204 | Z = 306 |
**PID[12422] | X = 101 | Y = 202 | Z = 303 |
##PID[12422] | X = 102 | Y = 204 | Z = 306 |
**PID[12421] | X = 101 | Y = 202 | Z = 303 |
##PID[12421] | X = 102 | Y = 204 | Z = 306 |
**PID[12423] | X = 101 | Y = 202 | Z = 303 |
##PID[12423] | X = 102 | Y = 204 | Z = 306 |
```

Question 2f

Which messages can never swap places?

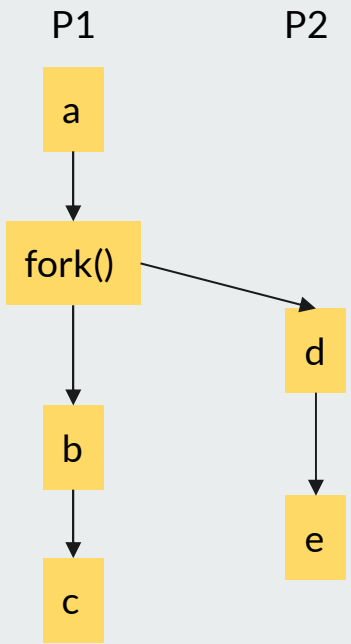
Question 2f

Which messages can never swap places?



Question 2f

Which messages can never swap places?



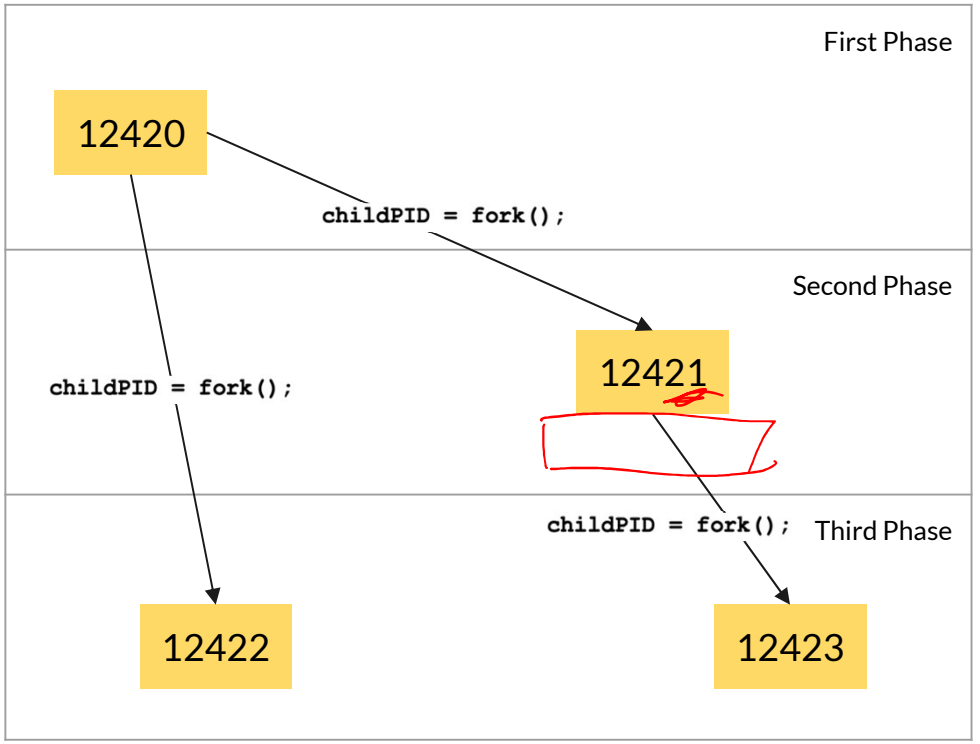
- "*" and "#" messages from the same process can never change place as sequential ordering is still preserved in the same process.
- Likewise, messages from the same process will always follow the phases, i.e. "*", "#" before "**" and "##".
- Message from the first phase (only one) must precede all other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

Question 2g

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

//Insertion Point (for parts (g), (h))

if (childPID == 0){
    sleep(5);  //sleep for 5 seconds
}

//Third Phase
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;
```

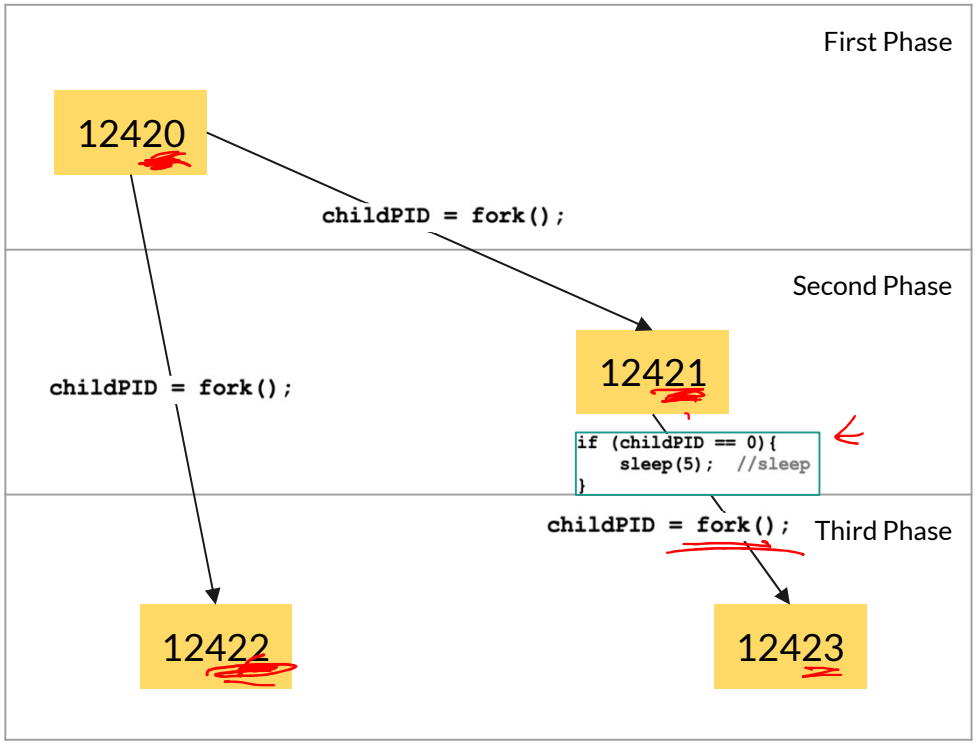
Question 2g

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

//Insertion Point (for parts (g), (h))
if (childPID == 0){
    sleep(5); //sleep for 5 seconds
}
//Third Phase
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;
```



Question 2g

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

//Insertion Point (for parts (g), (h))
if (childPID == 0){
    sleep(5);  //sleep for 5 seconds
}
//Third Phase
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;
```



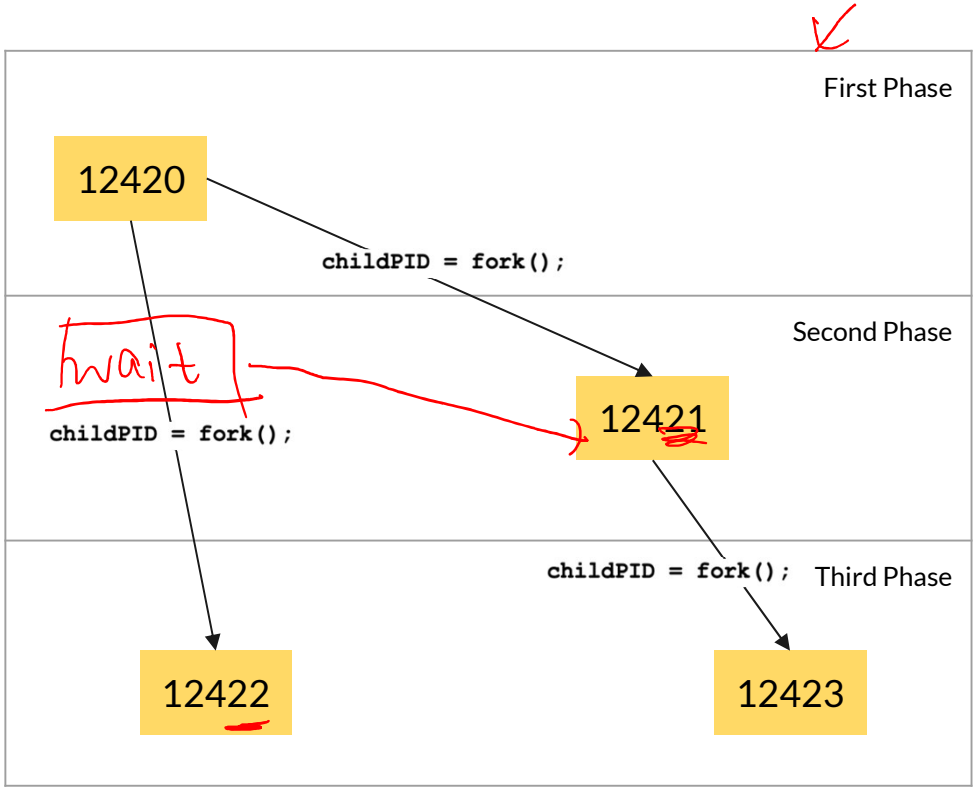
Possible scenario: both 12420 and 12422 terminates before 12421 wakes up

Question 2h

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

//Insertion Point (for parts (g), (h))
if (childPID != 0){
    wait(NULL);  //NULL means we don't care
                 //   about the return result
}
//Third Phase
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;
```

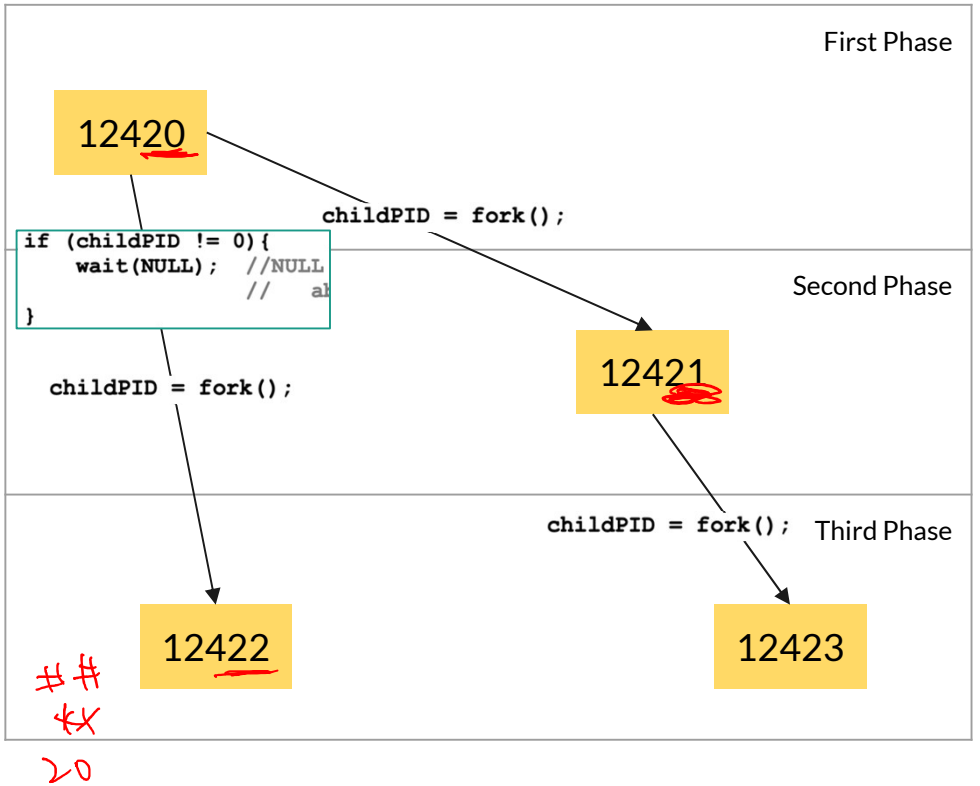
Question 2h

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

//Insertion Point (for parts (g), (h))
if (childPID != 0){
    wait(NULL);  //NULL means we don't care
                 //   about the return result
}
//Third Phase
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;
```



Question 2h

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

//Insertion Point (for parts (g), (h))
if (childPID != 0){
    wait(NULL);  //NULL means we don't care
                 //   about the return result
}
//Third Phase
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;
```



Fixed behaviour: all messages from 12421 must be printed before process 12420 and subsequently 12422 can be printed.

Question 3

PrimeFactors.c

- Prime factorisation program
- Compile to PF (executable, run by calling ./PF)

Parallel.c (on the right)

- Use fork() and execl() to run PF (executable)

```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, childResult >> 8);

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

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

}
```

Question 3

```c
int main()
{
    int i, j, userInput[9], nInput, childPid[9], childResult, pid;
    //Since largest number is 10 digits, a 12 characters string is more
    //than enough
    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);

        }
    }

    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;
        }
        printf("%d has %d prime factors\n", userInput[j], childResult >> 8);
    }
}
```

Question 3

waitpid() forces the main process to wait for the child process in certain order, e.g. the creation order of the child processes.

Possible outcome: (faster processes may get to printed later because of creation order)

```
$> a.out < test2.in
//Results
44721359 has 1 prime factors
99999989 has 1 prime factors
9 has 2 prime factors
111113111 has 1 prime factors
118689518 has 3 prime factors
```

```c
// use waitpid()
for (i = 0; i < nInput; i++){
    pid = waitpid(childPid[i], &childResult, 0);
    printf("%d has %d prime factors\n", userInput[i], childResult >> 8);
}
```

Question 3

One thing to note:

We are "paying" process spawning overhead to "earn" (real) parallel execution.

If there is only a single processor, or the overhead > earning from parallel execution, the solution will NOT show any improvement.

Thank you!