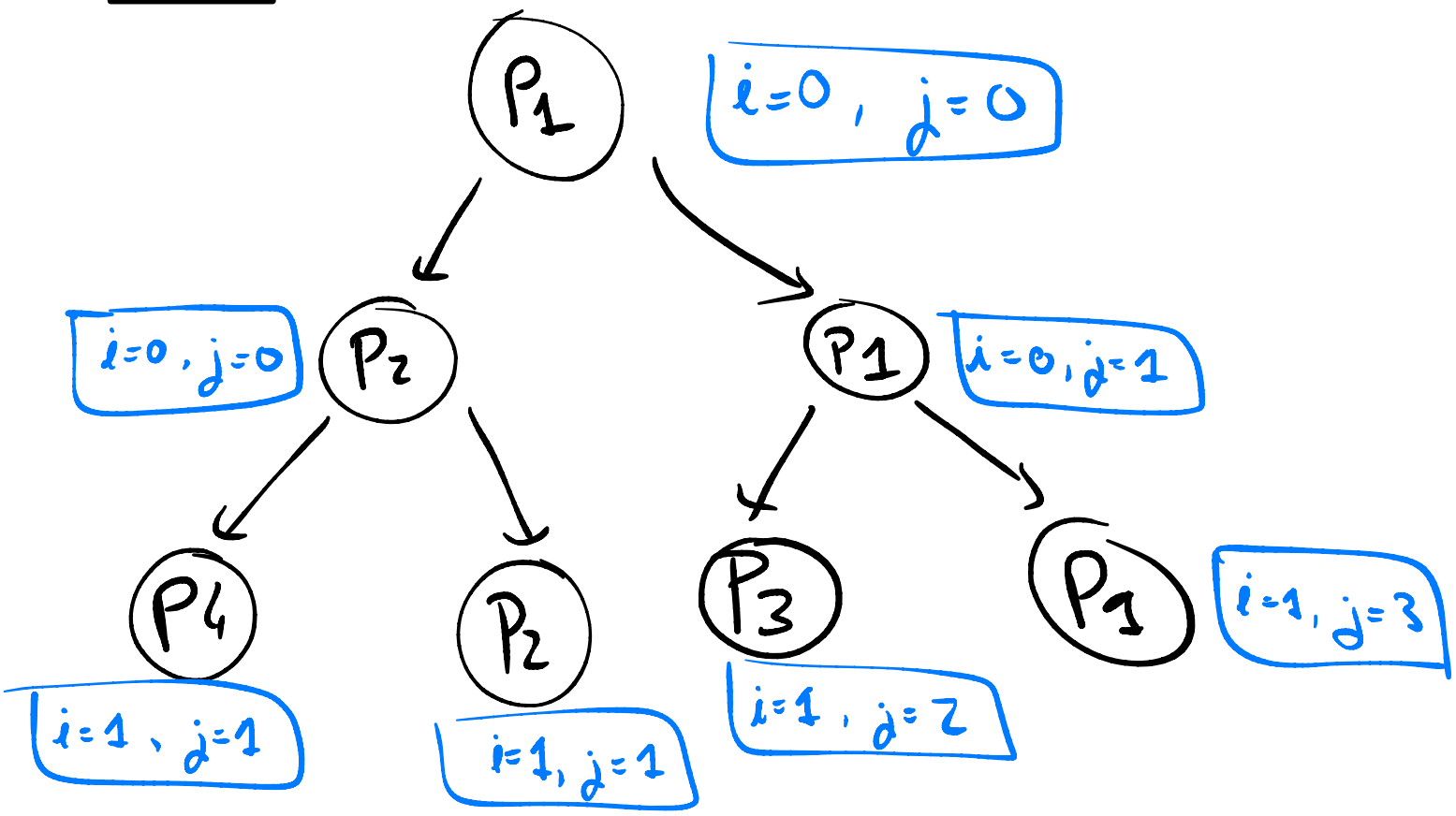
Ex 6



① P_1 starts at $\begin{cases} i=0 \\ j=0 \end{cases}$

↳ fork() is called and generates P_2.

② P_1 continues with $j=1$ and loops $i=0$.

↳ waits for P_2, for next iteration.

③ P_1 starts next iteration with $i=1, j=1$.

↳ Calls fork() → generates P_3

④ P_2 goes to next iteration, calls fork() → generates P_4

↳ P_2 continues to the next iteration $i=1, j=1$.

Ex 7

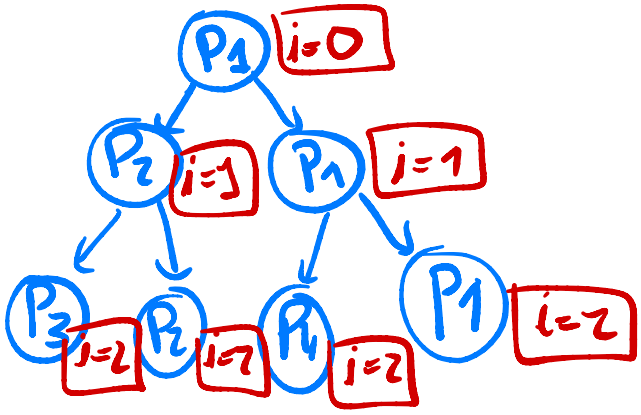
```
#include <unistd.h>

int main() {
    int pid;
    write(1, "I am the parent\n", 16)
    for (int i=0; i<3; i++) {
        pid = fork();
        if (pid == 0){
            write(1, "I am a child\n", 13);
            _exit(0);
        }
    }
    while (waitpid(-1, NULL, 0)>0);
    return 0;
}
```

Ex 8

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

int main(){

    int N = 3;
    int pid;

    for (int i=0; i<N; i++){

        pid = fork();

        if (pid == 0){
            // Do nothing  → This is the block for the child
        } else {
            wait(NULL); // Waiting for the child to complete.
            printf("I am the parent %d,
                my child is %d\n",
                getpid(), pid());
            return 0;
        }
    }

    return 0;
}
```

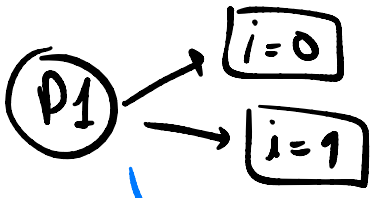
Ex 9

a) Each iteration with fork() produces a new process.



b) Messages printed:

Assuming we start at 100,



i=2, won't be printed as (101%2==1)

P2 → i=1

↳ i=2 won't be printed again

P3 → i=2

P4 → Won't print because (103%2==1)

c) (P1) prints for → [i=2] (101%2 == 1)

(P2) prints for → [i=2] (101%2 == 1)

P_3 and P_4 won't print.

d) We should add:

```
if(pid != 0) return;
```

↳ if we add this after the fork(), we will exit the parent process from the loop.