Your Tutor

Nguyen Minh Nguyen

Email: nmnguyen@nus.edu.sg

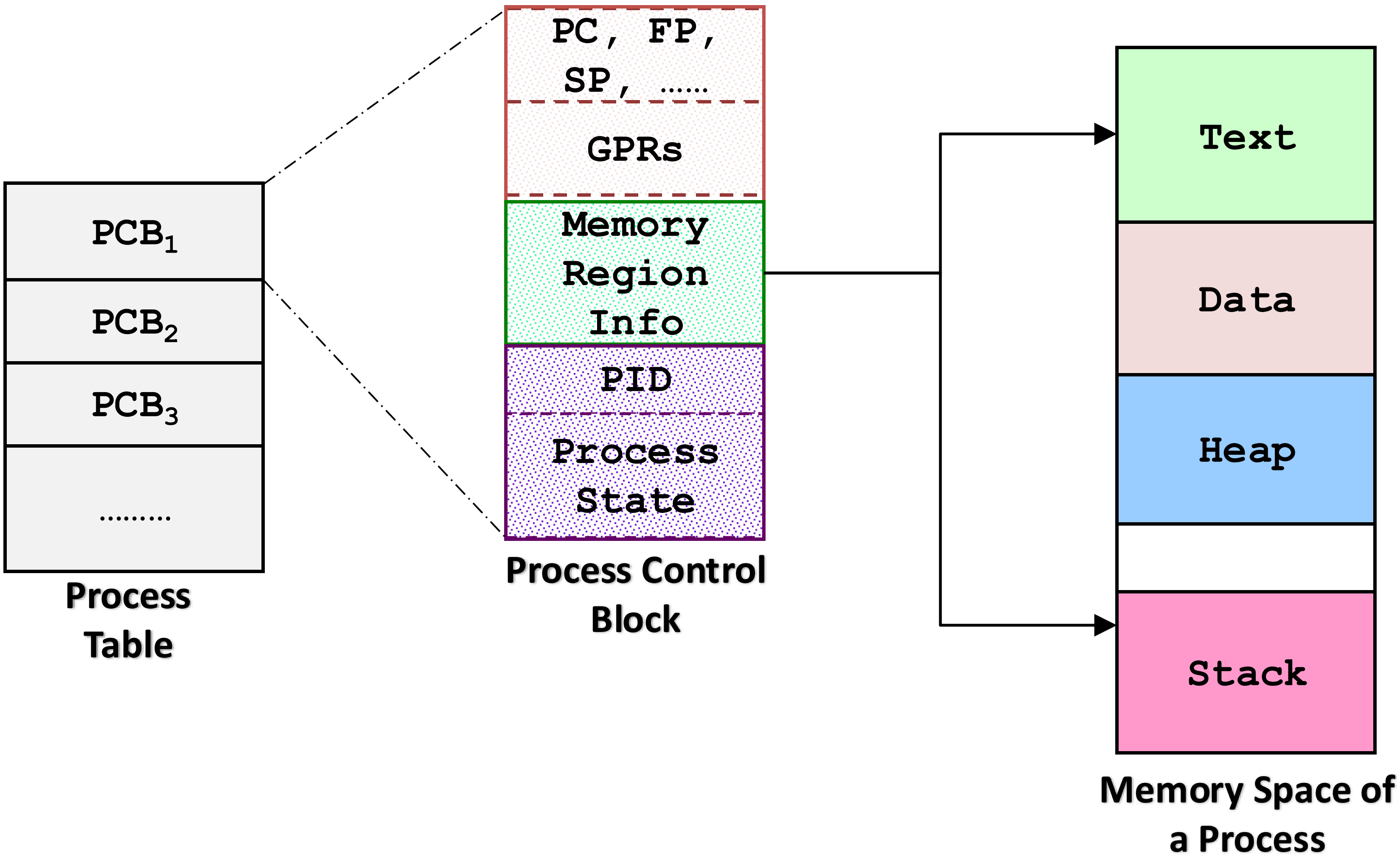
General

- I will use **PowerPoint slides** and **VSCode + Bash Terminal** for tutorials.
  - Slides will be disseminated via Canvas.

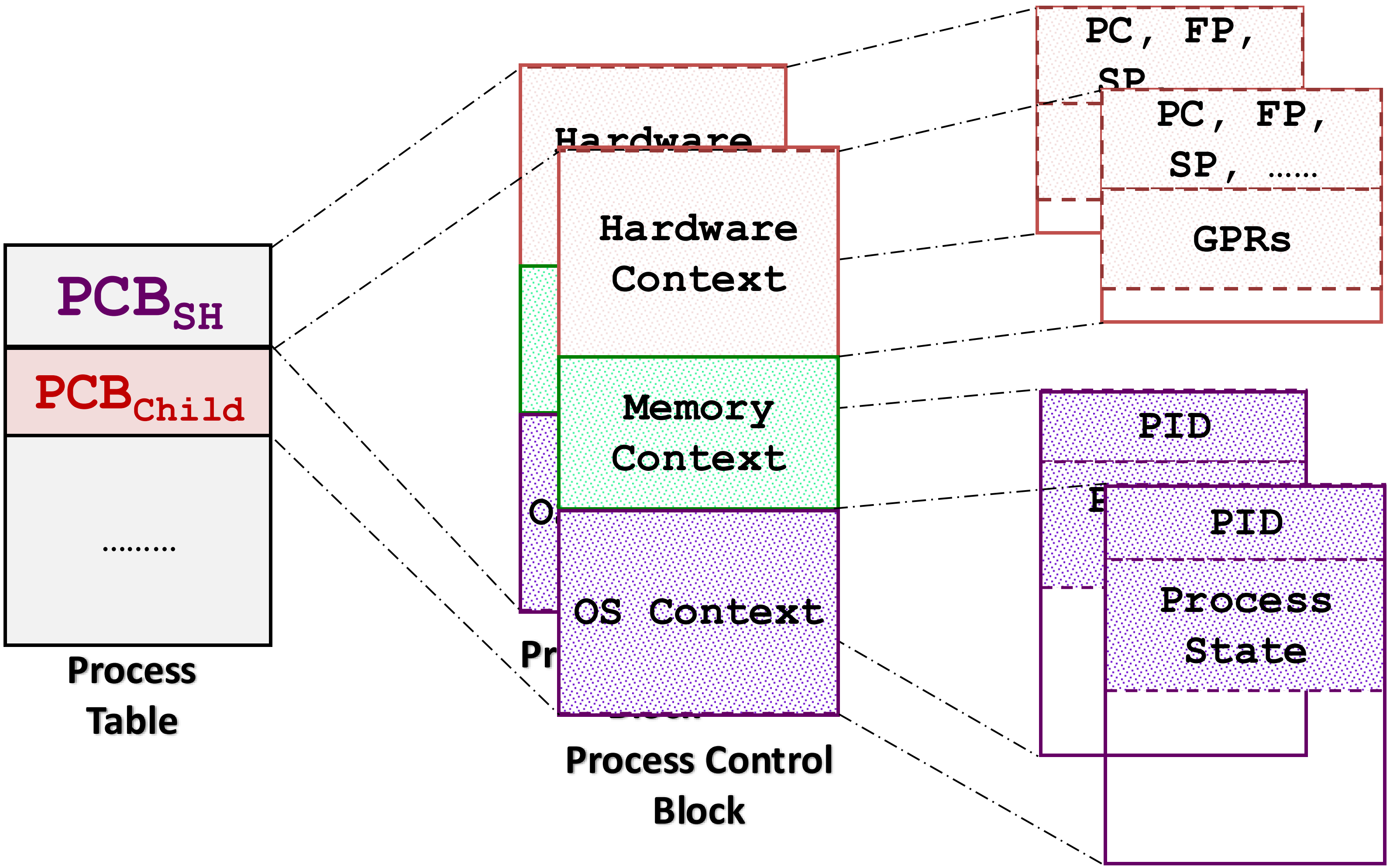
Tutorial Attendance

- Makes up **2% of overall grade**.
  - **Absent with valid reason:** Submit MC / reason to me via email.
  - **Absent without valid reason:** Also let me know via email.

Topic Summary



fork( )?


PCB$_{SH}$
PCB$_{Child}$
....
Process Table
Hardware Context
Memory Context
OS Context
Process Control Block
PC, FP, SP, ...
GPRs
PID
Process State

Process Creation in Unix: `fork()` (cont)

- Behavior:
  - Creates a new process (known as ***child process***)
  - Child process is a **duplicate** of the current executable image
    - i.e. same code, same address space etc
    - Data in child is a **COPY** of the parent (ie.not shared)
  - Child **differs only in:**
    - Process id (PID)
    - Parent (PPID)
      - Parent = The process which executed the fork()
    - **`fork()`** return value

Tutorial 2

Question 1

1. fork --- wait

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

1. fork --- wait

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>
    }
     <Point β>

    return 0;
}
```

Behaviour

Process Q ***always*** terminate before P.

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

1. fork --- wait

C code:

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

    return 0;
}
```

Behaviour

Process Q ***always*** terminate before P.

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

1. fork --- wait

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.**

Tutorial 2

Question 2

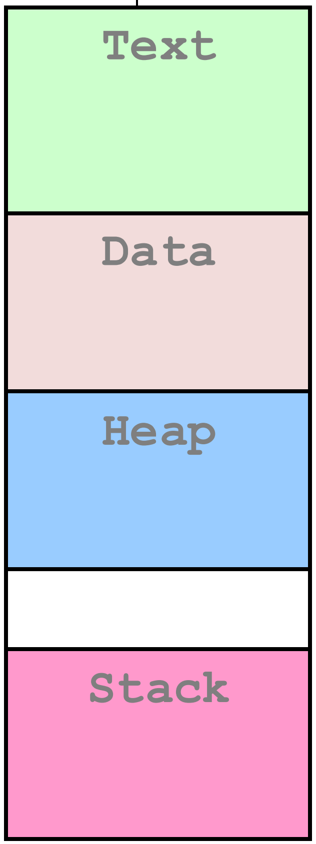
a. dataX, dataY, region pointed by dataZptr

C code:

```
int dataX = 100;

int main( )
{
    pid_t childPID;

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



Memory Space of a Process

b. Memory space after fork()?

```
PID[550761] | X = 100 | Y = 200 | Z = 300 |
*PID[550761] | X = 100 | Y = 200 | Z = 300 |
#PID[550761] | X = 101 | Y = 202 | Z = 303 |
**PID[550761] | X = 101 | Y = 202 | Z = 303 |
##PID[550761] | X = 102 | Y = 204 | Z = 306 |
*PID[550762] | X = 100 | Y = 200 | Z = 300 |
#PID[550762] | X = 101 | Y = 202 | Z = 303 |
**PID[550763] | X = 101 | Y = 202 | Z = 303 |
##PID[550763] | X = 102 | Y = 204 | Z = 306 |
**PID[550762] | X = 101 | Y = 202 | Z = 303 |
##PID[550762] | X = 102 | Y = 204 | Z = 306 |
**PID[550764] | X = 101 | Y = 202 | Z = 303 |
##PID[550764] | X = 102 | Y = 204 | Z = 306 |
```

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);

    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);
```

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);
```

```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);
```

Code Insertion Point

```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;
}
```

First

Second

Third

c. Process Tree

```
PID[550761] | X = 100 | Y = 200 | Z = 300 |
*PID[550761] | X = 100 | Y = 200 | Z = 300 |
#PID[550761] | X = 101 | Y = 202 | Z = 303 |
**PID[550761] | X = 101 | Y = 202 | Z = 303 |
##PID[550761] | X = 102 | Y = 204 | Z = 306 |
*PID[550762] | X = 100 | Y = 200 | Z = 300 |
#PID[550762] | X = 101 | Y = 202 | Z = 303 |
**PID[550763] | X = 101 | Y = 202 | Z = 303 |
##PID[550763] | X = 102 | Y = 204 | Z = 306 |
**PID[550762] | X = 101 | Y = 202 | Z = 303 |
##PID[550762] | X = 102 | Y = 204 | Z = 306 |
**PID[550764] | X = 101 | Y = 202 | Z = 303 |
##PID[550764] | X = 102 | Y = 204 | Z = 306 |
```

761

762

763

764

d, e: Message Ordering

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
    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

```
PID[550761] | X = 100 | Y = 200 | Z = 300 |
*PID[550761] | X = 100 | Y = 200 | Z = 300 |
#PID[550761] | X = 101 | Y = 202 | Z = 303 |
**PID[550761] | X = 101 | Y = 202 | Z = 303 |
##PID[550761] | X = 102 | Y = 204 | Z = 306 |
*PID[550762] | X = 100 | Y = 200 | Z = 300 |
#PID[550762] | X = 101 | Y = 202 | Z = 303 |
**PID[550763] | X = 101 | Y = 202 | Z = 303 |
##PID[550763] | X = 102 | Y = 204 | Z = 306 |
**PID[550762] | X = 101 | Y = 202 | Z = 303 |
##PID[550762] | X = 102 | Y = 204 | Z = 306 |
**PID[550764] | X = 101 | Y = 202 | Z = 303 |
##PID[550764] | X = 102 | Y = 204 | Z = 306 |
```

f: Sleepy Child?

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

```
PID[550761] | X = 100 | Y = 200 | Z = 300 |
*PID[550761] | X = 100 | Y = 200 | Z = 300 |
#PID[550761] | X = 101 | Y = 202 | Z = 303 |
**PID[550761] | X = 101 | Y = 202 | Z = 303 |
##PID[550761] | X = 102 | Y = 204 | Z = 306 |
*PID[550762] | X = 100 | Y = 200 | Z = 300 |
#PID[550762] | X = 101 | Y = 202 | Z = 303 |
**PID[550763] | X = 101 | Y = 202 | Z = 303 |
##PID[550763] | X = 102 | Y = 204 | Z = 306 |
**PID[550762] | X = 101 | Y = 202 | Z = 303 |
##PID[550762] | X = 102 | Y = 204 | Z = 306 |
**PID[550764] | X = 101 | Y = 202 | Z = 303 |
##PID[550764] | X = 102 | Y = 204 | Z = 306 |
```

g: No Child left behind?

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
    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

```
PID[550761] | X = 100 | Y = 200 | Z = 300 |
*PID[550761] | X = 100 | Y = 200 | Z = 300 |
#PID[550761] | X = 101 | Y = 202 | Z = 303 |
**PID[550761] | X = 101 | Y = 202 | Z = 303 |
##PID[550761] | X = 102 | Y = 204 | Z = 306 |
*PID[550762] | X = 100 | Y = 200 | Z = 300 |
#PID[550762] | X = 101 | Y = 202 | Z = 303 |
**PID[550763] | X = 101 | Y = 202 | Z = 303 |
##PID[550763] | X = 102 | Y = 204 | Z = 306 |
**PID[550762] | X = 101 | Y = 202 | Z = 303 |
##PID[550762] | X = 102 | Y = 204 | Z = 306 |
**PID[550764] | X = 101 | Y = 202 | Z = 303 |
##PID[550764] | X = 102 | Y = 204 | Z = 306 |
```

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

Tutorial 2

Question 3

```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);
    }

}
```

```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;
}
```

```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));
    }
}
```

WEXITSTATUS

WEXITSTATUS is a **macro** used in C programming (specifically in POSIX-compliant systems, like Linux and Unix) to extract the **exit status** of a terminated child process.

The reason we don’t directly use status instead of WEXITSTATUS(status) is that status contains **more than just the exit code**. It is an encoded integer with multiple bits that store different types of information about how the child process terminated.

When a process terminates, the wait() or waitpid() function stores information in an int status variable. This integer is **not just the exit code**; it also includes:
- **Exit Status (Bit 8-15)**: The actual exit code (exit(n) or return n).
- **Signal Information**: If the process was killed by a signal (e.g., SIGKILL), the signal number is stored.
- **Core Dump Info**: Indicates if the process dumped core before termination.

Because of this encoding, you cannot directly use status to get the exit code.

Alternatively, you can use **status >> 8** to get the **exit code**.