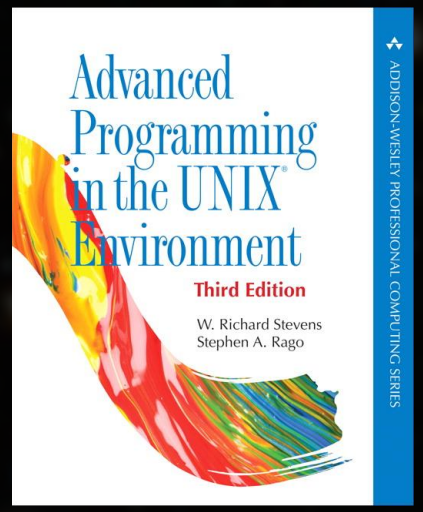
Opening Scene, Parasite (2019) - Bong Joon-ho

Lab08 and Lec08 is Extended for 1 Week!

Lab09 and Lec09 is extended for 1 week!



Chapter 08: Process Control

Chapter 10: Signal

Chapter 15: Inter-Process Communication

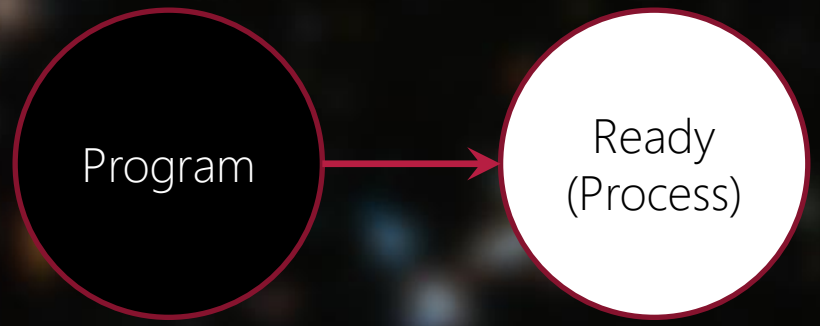
Multiprocessing

aka multiprogramming

Single Processor ~~Multiprocessor~~

Process Life Cycle

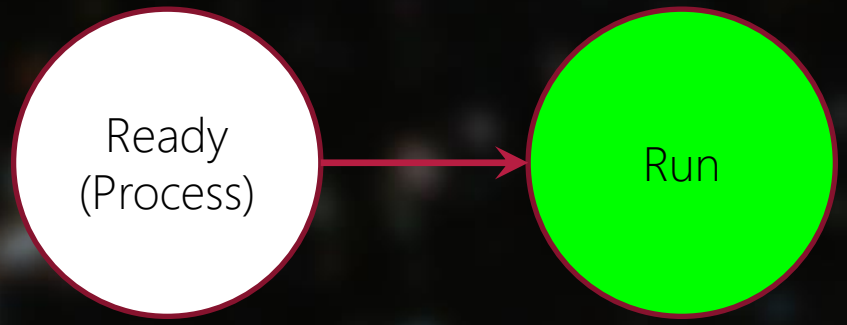
Process States



Program is bootstrapped into memory and becomes process

But still have not assigned share of processor!

Like a chess player that registered but have not been called for a game.



Process is given processor and runs.

The chess player starts the game …

Process finishes within the given time slice of processor.
The chess player checkmates in one move!

Process waits (is blocked) for different reasons:

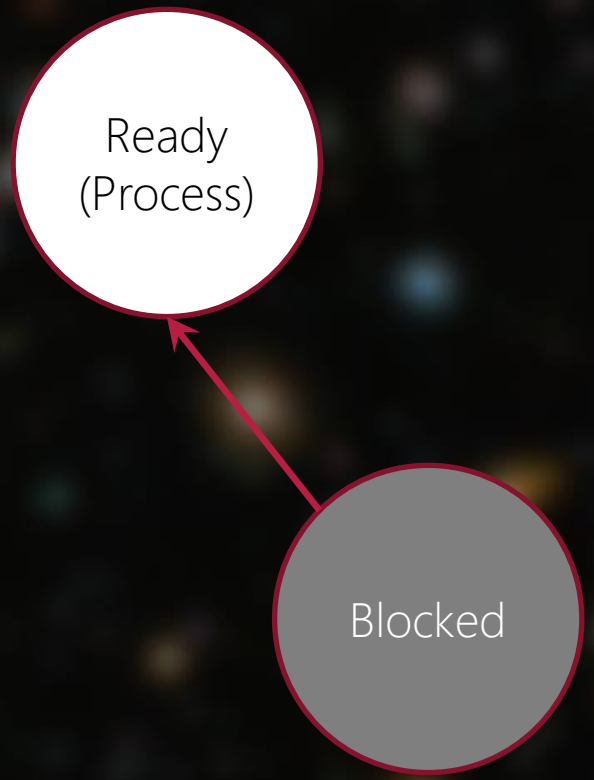
1) I/O: inputs from user, inputs from device, …
2) Child process to finish
3) ….

The chess player is waiting for her rival's next move …

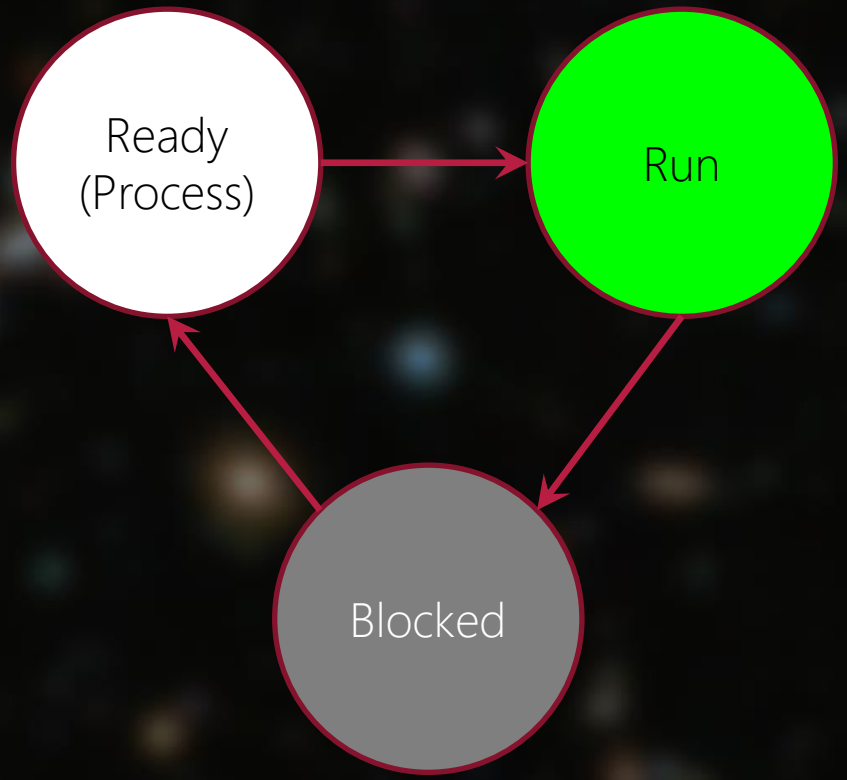
Process receives what is needed:

1) I/O: user enters inputs, device sends data, …
2) Child finishes
3) ….

The chess player's rival do his move

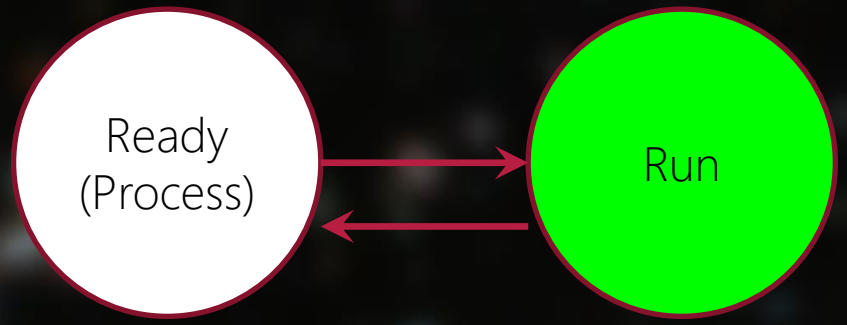


Process receives is given share of processor again:
The chess player have the chessboard again and can do her move.



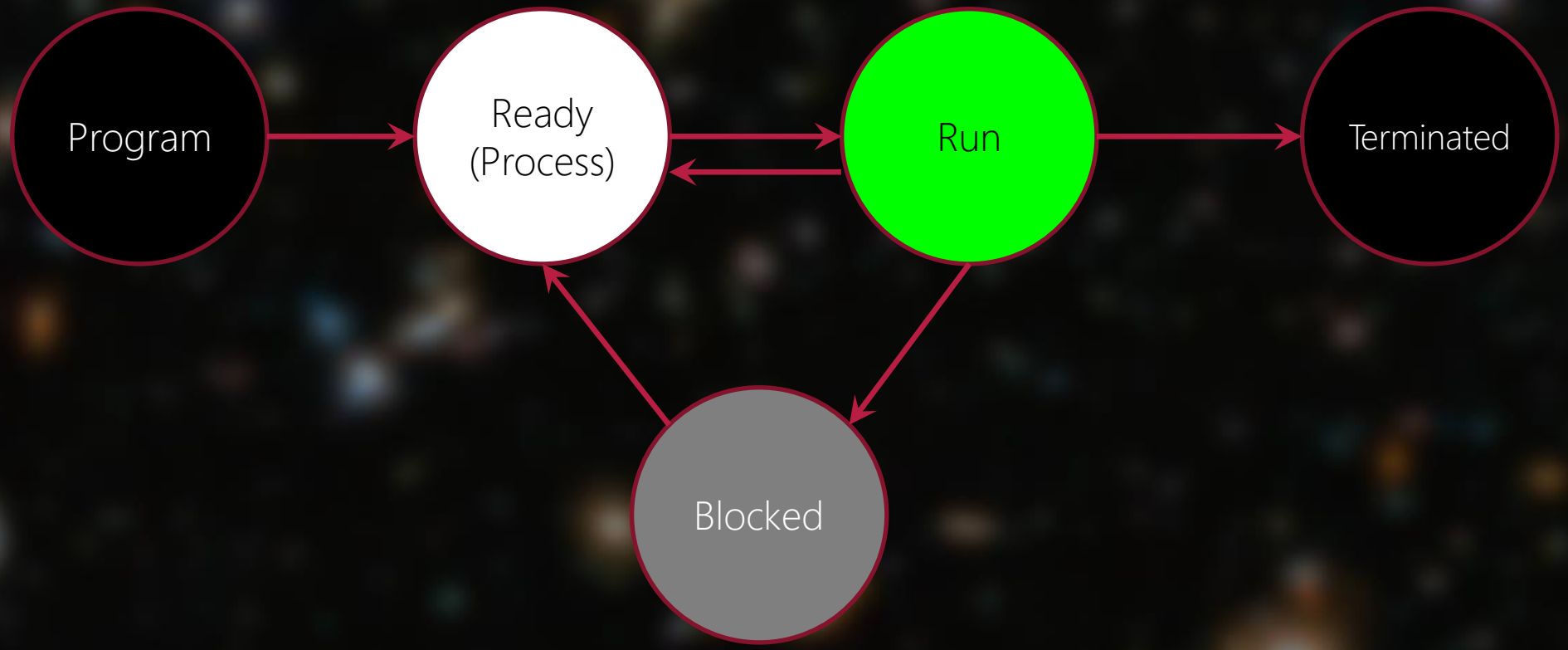
If there is no I/O or child, but the time slices are passed:

The chess player have done 2 moves, now it's others turn to do their move



Process finishes after many loops of blocked, ready, run

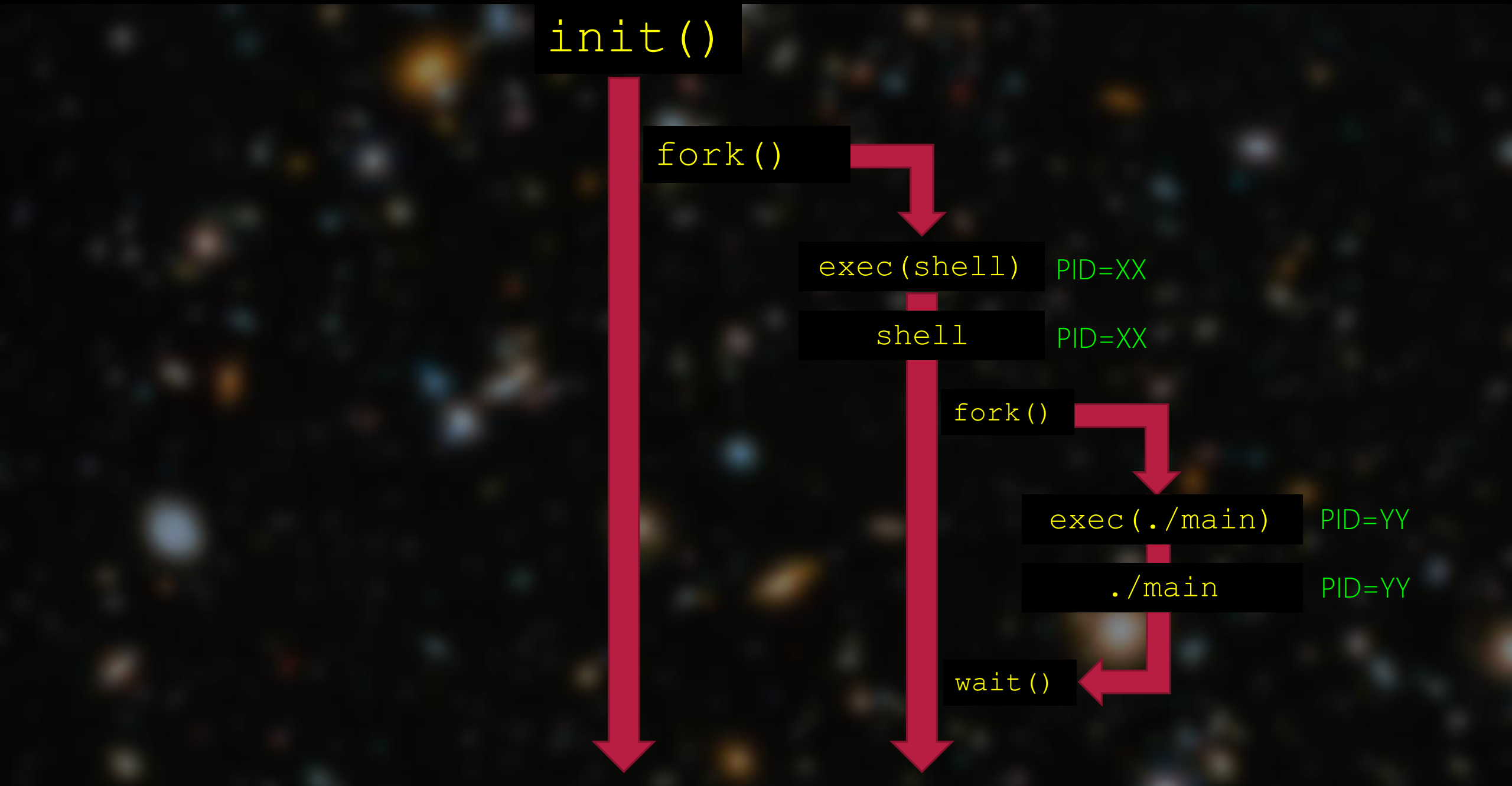
The chess player checkmates after many waiting of moves.


Program
Ready
(Process)
Run
Terminated
Blocked

UNIX Startup

`init()`

BIOS → MBR → Kernel → PID=1



Food for Thought

1) Asking for more processor sharing (changing process priority)
2) Asking for the list of Ready, Blocked, Zombies, Orphans …
3) Asking for more children
4) Asking for grandchild

Inter-Process Communication

Parent ↔ Child

Any Process ↔ Any Other Process

Single Processor ~~Multiprocessor~~

Parent → Child

Passing Tasks

Passing Information

Parent → Child

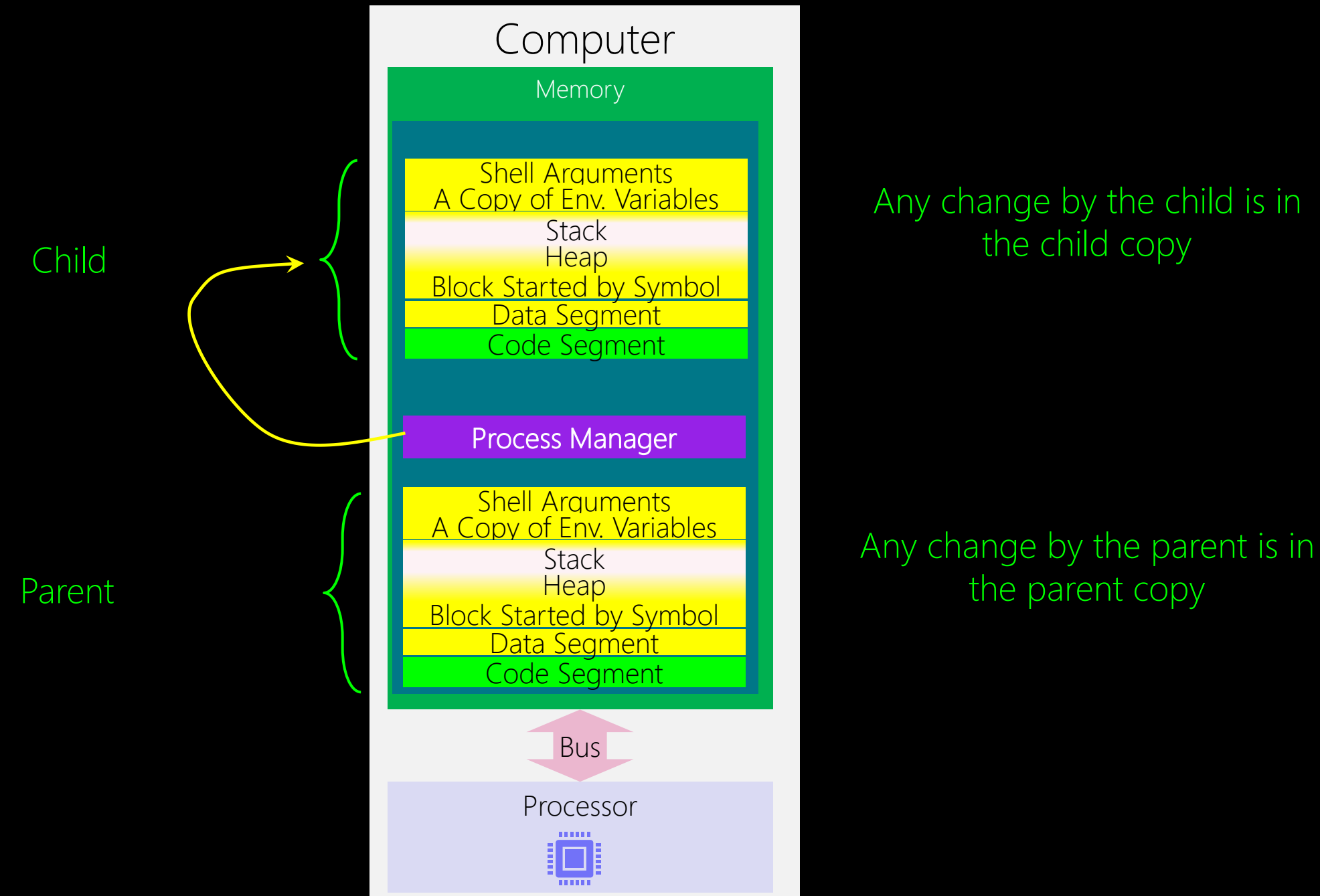
```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent pid=%d\n", getppid());
                //Assign child's tasks here
                exit(0);
        }
}
//Assign parent tasks here
```

Child's Tasks

Parent's Tasks

Wait for the child

```
exit(0);
```


Computer
Memory
Shell Arguments
A Copy of Env. Variables
Stack
Heap
Block Started by Symbol
Data Segment
Code Segment
Child
Any change by the child is in the child copy
Process Manager
Shell Arguments
A Copy of Env. Variables
Stack
Heap
Block Started by Symbol
Data Segment
Code Segment
Parent
Any change by the parent is in the parent copy
Bus
Processor

Parent → Child

Passing Tasks

Passing Information

After `fork()`, any change to the variables are local to the parent and child processes.

After `fork()`, there is no conversation/communication until …

Parent

Parent → Child

13

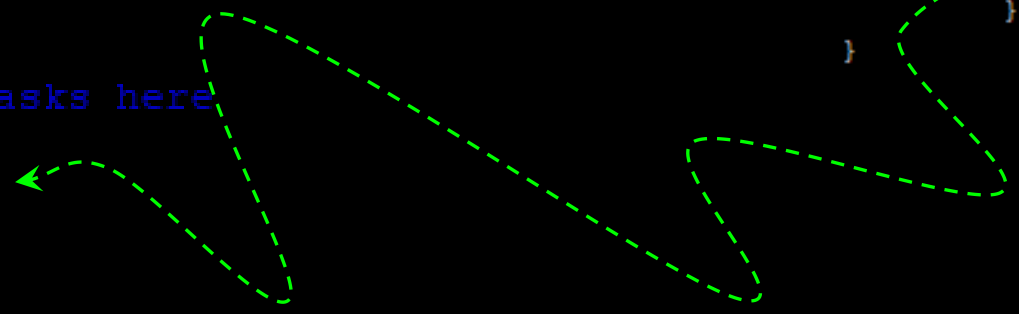
```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
}

//Assign parent tasks here
int *child_exit;
wait(child_exit);
```

Child

Parent → Child

0

```
int child_pid = fork();
if(child_pid == -1){
        perror("impossible to have a child!");
        exit(1);
}
if(child_pid >= 0){//(child_pid != -1)
        if(child_pid > 0)
                printf("I am the parent, pid=%d\n", getpid());
        else{//(child_pid == 0)
                printf("I am the child, pid=%d\n", getpid());
                printf("My parent is pid=%d\n", getppid());
        }
        exit(0);
}
```



Wait for Child Process be over

System Calls: `wait()` in `sys/wait.h`

Like HLT (HALT) to processor, kernel can also halt a process:

- Not give any processor time/slices
- It is called blocking for processes instead of halting.

Wait for Child Process be over

System Calls: `wait()` in `sys/wait.h`

```
#include <sys/wait.h>
pid_t wait(int *statloc);
```

Return Child's PID if OK, or −1 on error

Wait for Child Process be over

System Calls: `wait()` in `sys/wait.h`

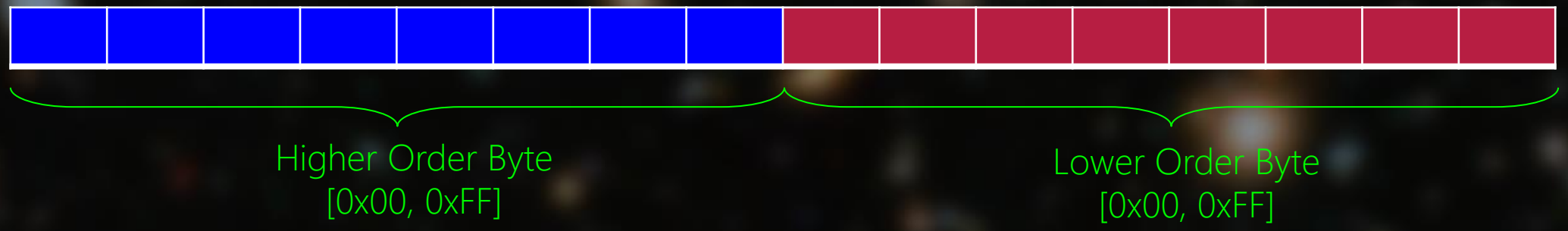
```
#include <sys/wait.h>
pid_t wait(0);
```

Parent does not care about how the child terminates!

Return Child's PID if OK, or –1 on error

Wait for Child Process be over

`int *statloc → status`



Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.

`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute
	`WTERMSIG(`*status*`)`
	to fetch the signal number that caused the termination.
	Additionally, some implementations (but not the Single UNIX Specification) define the macro
	`WCOREDUMP(`*status*`)`
	that returns true if a core file of the terminated process was generated.

`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`

Macro vs. Function

Reminder from C Program

```c
#include <stdio.h>
#define MAX(x,y) ((x>y)?x:y)
void main()
{
        int a, b, max;

        printf("Enter first number: \n");
        scanf("%d",&a);
        printf("Enter second number: \n");
        scanf("%d",&b);

        max = MAX(a,b);
        printf("Maximum number is: %d\n",max);
}
```

Before Compile Time

```
cc max.c -o max
```

```c
#include <stdio.h>
#define MAX(x,y) ((x>y)?x:y)
void main()
{
        int a, b, max;

        printf("Enter first number: \n");
        scanf("%d",&a);
        printf("Enter second number: \n");
        scanf("%d",&b);

        max = ((a>b)?a:b);
        printf("Maximum number is: %d\n",max);
}
```

Macros for Child Exit Status

`int *statloc → status`

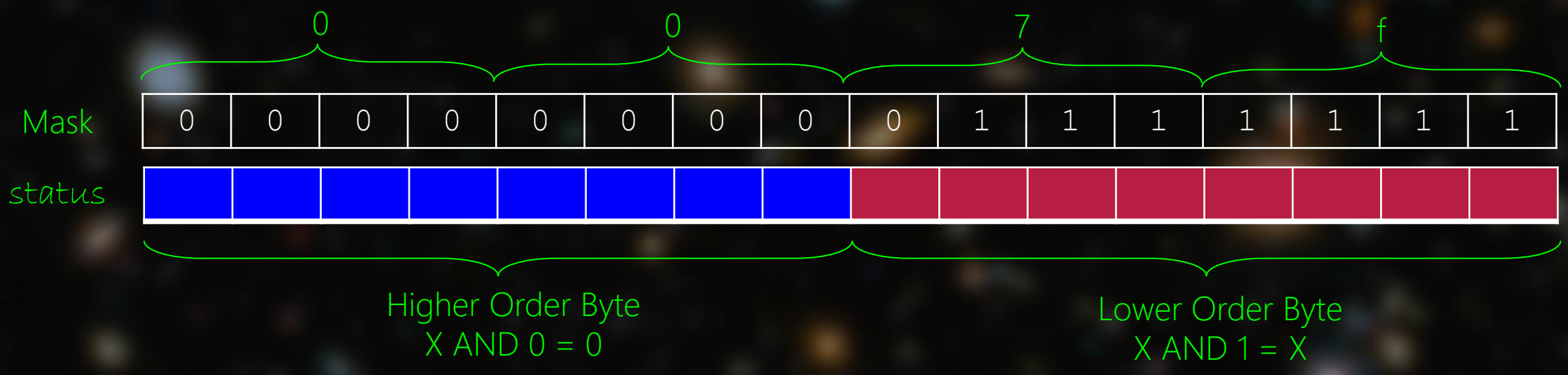
https://code.woboq.org/gcc/include/sys/wait.h.html

https://code.woboq.org/qt5/include/bits/waitstatus.h.html

Child EXIT_SUCESS

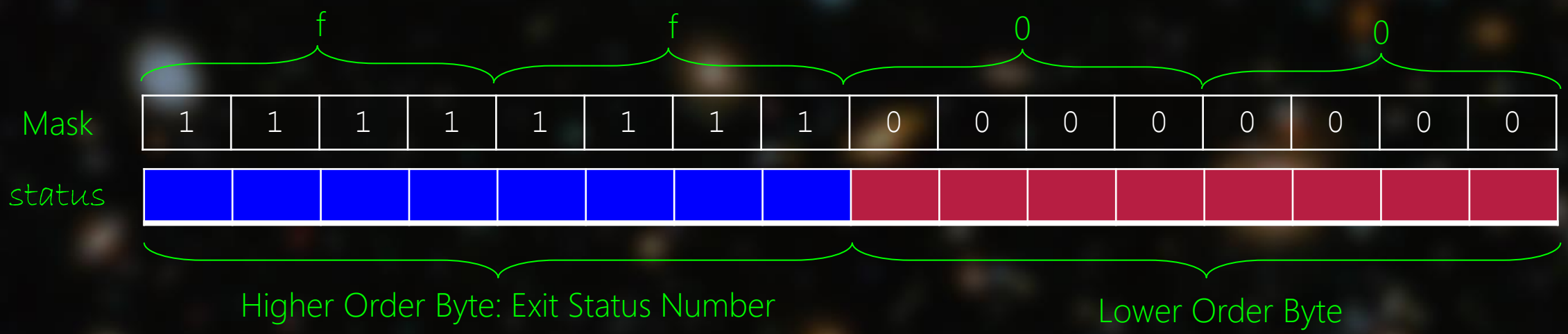
```
/* Nonzero if STATUS indicates normal termination.  */
#define __WIFEXITED(status)     (__WTERMSIG(status) == 0)
```

```
/* If WIFSIGNALED(STATUS), the terminating signal.  */
#define __WTERMSIG(status)      ((status) & 0x7f)
```



Child EXIT_SUCESS

```
/* If WIFEXITED(STATUS), the low-order 8 bits of the status.  */
#define __WEXITSTATUS(status)   (((status) & 0xff00) >> 8)
```



```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
int main(int argc, char *argv[]){
        int a = 0;
        int b = 0;
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        printf("I am a lonely process, pid=%d\n", getpid());
        int child_pid = fork();
        if(child_pid == -1){
                perror("impossible to have a child!\n");
                exit(1);
        }
        if(child_pid >= 0){//(child_pid != -1)
                if(child_pid > 0)
                        printf("I am the parent, pid=%d\n", getpid());
                else{//(child_pid == 0)
                        printf("I am the child, pid=%d\n", getpid());
                        //Assign child's tasks here
                        printf("child: %d + %d = %d\n", a, b, a - b);
                        exit(0);
                }
        }
        //Assign parent tasks here
        printf("parent: %d + %d = %d\n", a, b, a + b);
    int child_exit;
    wait(&child_exit);

    if (WIFEXITED(child_exit))
            printf("normal termination, exit status = %d\n", WEXITSTATUS(child_exit));
    else if (WIFSIGNALED(child_exit))
            printf("abnormal termination, signal number = %d\n", WTERMSIG(child_exit));
```

```
hfani@alpha:~$ ./child_exit_status 3 5
I am a lonely process, pid=1911307
I am the parent, pid=1911307
parent: 3 + 5 = 8
I am the child, pid=1911308
child: 3 + 5 = -2
normal termination, exit status = 0
```

Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.

`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute
	`WTERMSIG(`*status*`)`
	to fetch the signal number that caused the termination.
	Additionally, some implementations (but not the Single UNIX Specification) define the macro
	`WCOREDUMP(`*status*`)`
	that returns true if a core file of the terminated process was generated.

`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`

Opening Scene, Parasite (2019) - Bong Joon-ho
https://www.youtube.com/watch?v=TpqGsRX0bhE

Signaling

Like Electric Shock (IRQ) from Devices to Processor (hardware), Signals are Process Shock to Another Process (software)

Software Interrupts

Kernel Process to Other Processes
Parent to Child
Ancestor Process to Grandchildren