OS: Lecture 5

Review

Process-Related System Calls

System call	Purpose	Return value on success	Return value on failure
`getpid()`	Get process identification.	PID of the calling process.	It never fails.
`fork()`	Create a child process.	In the parent: PID of the child. In the child: 0.	In the parent: **-1** (check `errno`). No child process is created.
`exec*()`	Execute a program.	It never returns.	**-1** (check `errno`).
`wait()`	Wait for a child to terminate.	PID of the terminated child.(status is stored in the argument.)	**-1** (check `errno`).

Exit codes

```
[yt2475@linserv1 proc]$ cat exitcode.c
int main() {
}
[yt2475@linserv1 proc]$ gcc -o exitcode exitcode.c
[yt2475@linserv1 proc]$ ./exitcode
[yt2475@linserv1 proc]$ echo $?
0
```

- `echo $?` returns the exit code of the previous program.
- For the program above, it returns 0 if succeed
- But you can also return any byte to show the states in the program

```
1 int main() {
2   return 42;
3 }
```

```
gcc -o exitcode exitcode.c
[yt2475@linserv1 proc]$ ./exitcode
[yt2475@linserv1 proc]$ echo $?
42
```

Processes in the kernel

- Kernel-Space Memory vs User-Space Memory

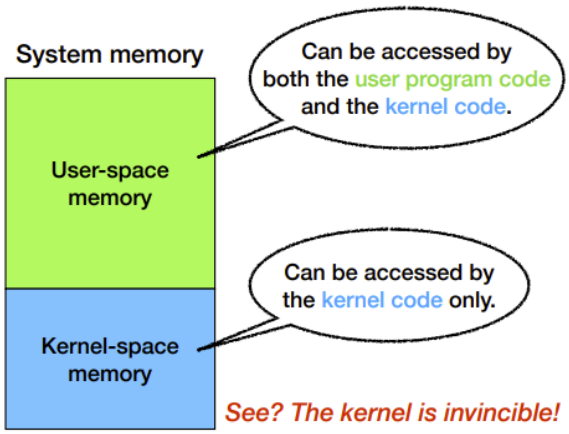
The memory is divided into two parts.

User-space memory stores…

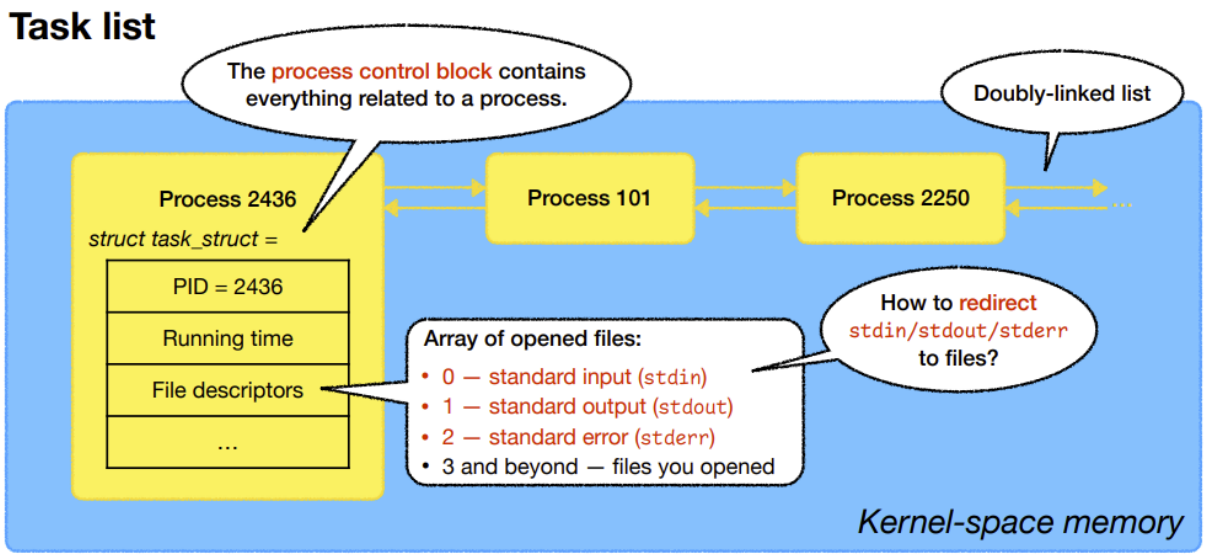
- Program code
- Process's memory
- …

Kernel-space memory stores…

- Kernel code
- Kernel data structures
- Loaded device drivers
- …



- Task Struct



- Redirect using file descriptors

How to redirect stdout to a file?

```c
int main() {
  printf("Hello CS202\n");

  int fd = open("output.txt",
                O_CREAT|O_WRONLY|O_TRUNC,
                S_IRUSR|S_IWUSR);
  dup2(fd, 1);  // duplicate the file descriptor
  close(fd);    // close the unused file descriptor

  printf("Hello CS202 again\n");
}
```

redirect.c

```
$ ./redirect
Hello CS202
$ cat output.txt
Hello CS202 again
$ _
```

- Shell commands
  - Bash in a shell : `<` , `>`

```
[yt2475@linserv1 nyuc]$ cat nyuc.c
#include <ctype.h>
#include <stdio.h>

#include "argmanip.h"

int main(int argc, const char *const *argv) {
  char **upper_args = manipulate_args(argc, argv, toupper);
  char **lower_args = manipulate_args(argc, argv, tolower);

  for (char *const *p = upper_args, *const *q = lower_args; *p && *q; ++argv, ++p, ++q)
    printf("[%s] -> [%s] [%s]\n", *argv, *p, *q);
  }

  free_copied_args(upper_args, lower_args, NULL);
}
[yt2475@linserv1 nyuc]$
```

- Output to file

```
[yt2475@linserv1 nyuc]$ cat nyuc.c > output.txt
[yt2475@linserv1 nyuc]$ cat output.txt
#include <ctype.h>
#include <stdio.h>

#include "argmanip.h"

int main(int argc, const char *const *argv) {
  char **upper_args = manipulate_args(argc, argv, toupper);
  char **lower_args = manipulate_args(argc, argv, tolower);

  for (char *const *p = upper_args, *const *q = lower_args; *p && *q; ++argv, ++p, ++q)
    printf("[%s] -> [%s] [%s]\n", *argv, *p, *q);
  }

  free_copied_args(upper_args, lower_args, NULL);
}
```

- input from file

```
[yt2475@linserv1 nyuc]$ cat < output.txt
#include <ctype.h>
#include <stdio.h>

#include "argmanip.h"

int main(int argc, const char *const *argv) {
  char **upper_args = manipulate_args(argc, argv, toupper);
  char **lower_args = manipulate_args(argc, argv, tolower);

  for (char *const *p = upper_args, *const *q = lower_args; *p && *q; ++argv, ++p, ++q)
    printf("[%s] -> [%s] [%s]\n", *argv, *p, *q);
  }

  free_copied_args(upper_args, lower_args, NULL);
}
```

- **Pipe in a shell:** `|`
  - Use on two or more program, connecting their input/output together

```
[yt2475@linserv1 nyuc]$ cat nyuc.c | wc -l
15
[yt2475@linserv1 nyuc]$ wc -l < nyuc.c
15
```

    - `wc -l` counts the number of lines of the input

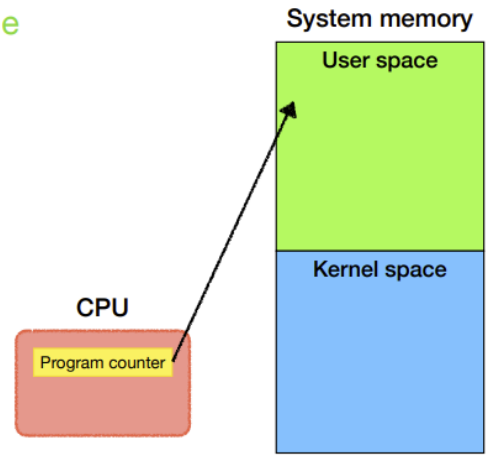
```
[yt2475@linserv1 nyuc]$ cat nyuc.c | grep args
  char **upper_args = manipulate_args(argc, argv, toupper);
  char **lower_args = manipulate_args(argc, argv, tolower);
  for (char *const *p = upper_args, *const *q = lower_args; *p && *q; ++argv, ++p, ++q)
{
  free_copied_args(upper_args, lower_args, NULL);
[yt2475@linserv1 nyuc]$ cat nyuc.c | grep args | wc -l
4
```

    - `grep` searches for a pattern

- **Process execution**

A process switches its execution from user mode to kernel mode by invoking system calls.

When the system call finishes, the execution switches from kernel mode back to user mode.



- Handling system calls - Example: `getpid()`

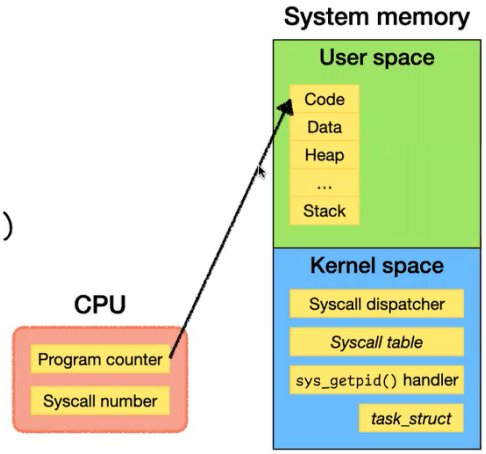
The CPU is running a process in user mode.

It wants to invoke the `getpid()` system call.

Each system call has a unique syscall number.

The process puts the syscall number of `getpid()` in a specific CPU register (*e.g.*, `%rax`).

Then it executes a `TRAP` instruction to switch from user mode to kernel mode.

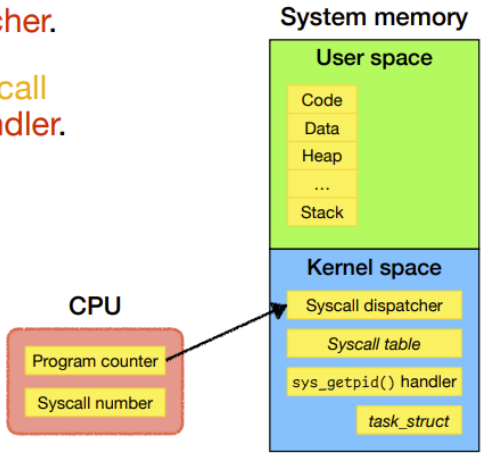


The kernel starts execution at the syscall dispatcher.

It examines the syscall number, looks up the syscall table, and invokes the corresponding syscall handler.

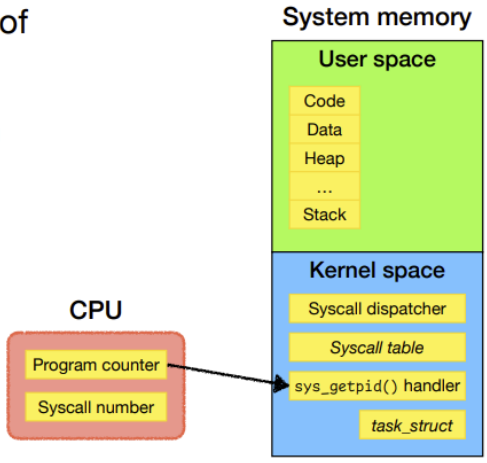
syscall_table[] =

Syscall #	Syscall handler
0	sys_read()
1	sys_write()
...	...
39	sys_getpid()
...	...



The `sys_getpid()` handler reads the Process ID of the calling process from `task_struct`.

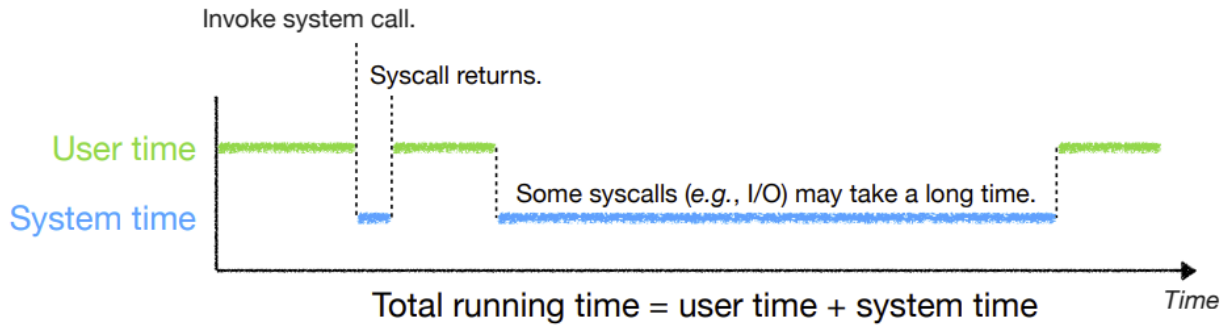
Then it executes a `RETURN-FROM-TRAP` instruction to switch from kernel mode to user mode.



- Process Time

User time vs. system time

The execution of a process is also divided into two parts.



The `time` command

```c
const int LOOP_MAX = 1000000;

int main() {
  for (int i = 0; i < LOOP_MAX; ++i) {
    printf("\n");
  }
}
```
time1.c

```c
const int LOOP_MAX = 1000000;

int main() {
  for (int i = 0; i < LOOP_MAX / 10; ++i) {
    printf("\n\n\n\n\n\n\n\n\n\n");
  }
}
```
time2.c

System calls can play a major role in performance.

- **Blocking system calls:** some system calls even stop your process until the data is available.

```
$ time ./time1
real    0m10.045s
user    0m0.204s
sys     0m3.793s
$ time ./time2
real    0m6.808s
user    0m0.090s
sys     0m2.218s
$ _
```

- the `time` command gives the runtime of the program

```
[yt2475@linserv1 proc]$ cat time1.c
#include <stdio.h>

const int LOOP_MAX = 1000000;

int main() {
  for (int i = 0; i < LOOP_MAX; ++i) {
    printf("\n");
  }
}
[yt2475@linserv1 proc]$ gcc -o time1 time1.c
[yt2475@linserv1 proc]$ time ./time1
```

```
real    0m8.300s
user    0m0.209s
sys     0m3.345s
```

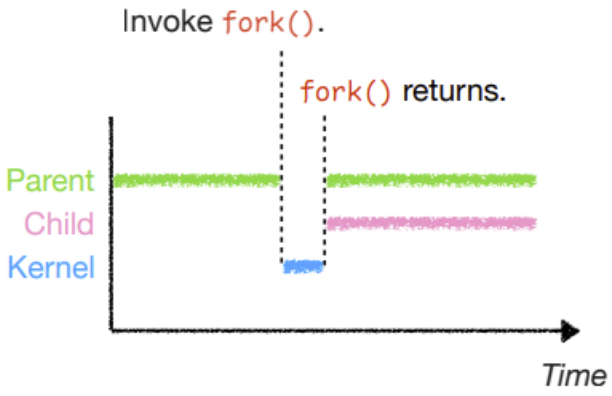
```
[yt2475@linserv1 proc]$ cat time2.c
#include <stdio.h>

const int LOOP_MAX = 1000000;

int main() {
  for (int i = 0; i < LOOP_MAX / 10; ++i) {
    printf("\n\n\n\n\n\n\n\n\n\n");
  }
}
[yt2475@linserv1 proc]$ gcc -o time2 time2.c
[yt2475@linserv1 proc]$ time ./time2
```

```
real    0m5.259s
user    0m0.096s
sys     0m1.841s
```

Process-management syscalls in the kernel

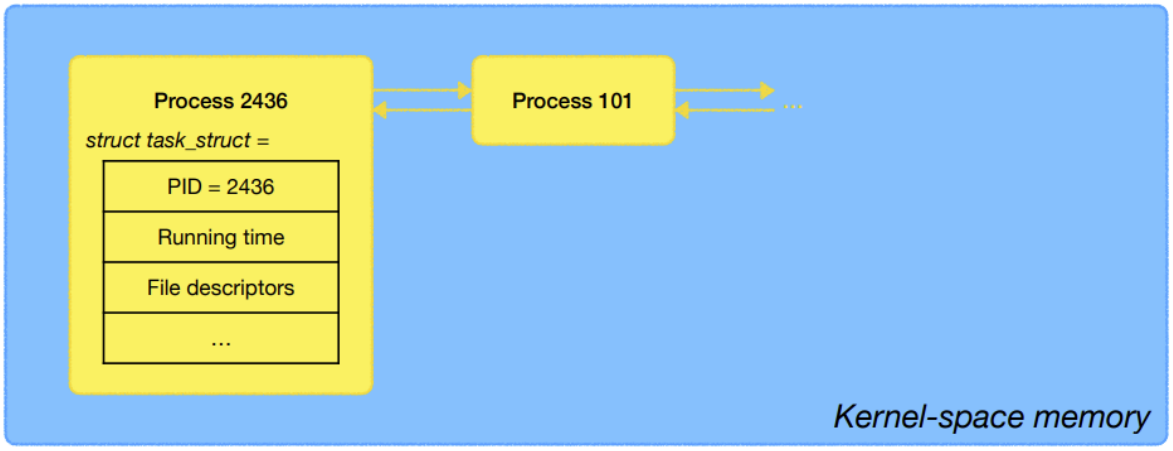
- `fork()`

From a programmer's perspective…

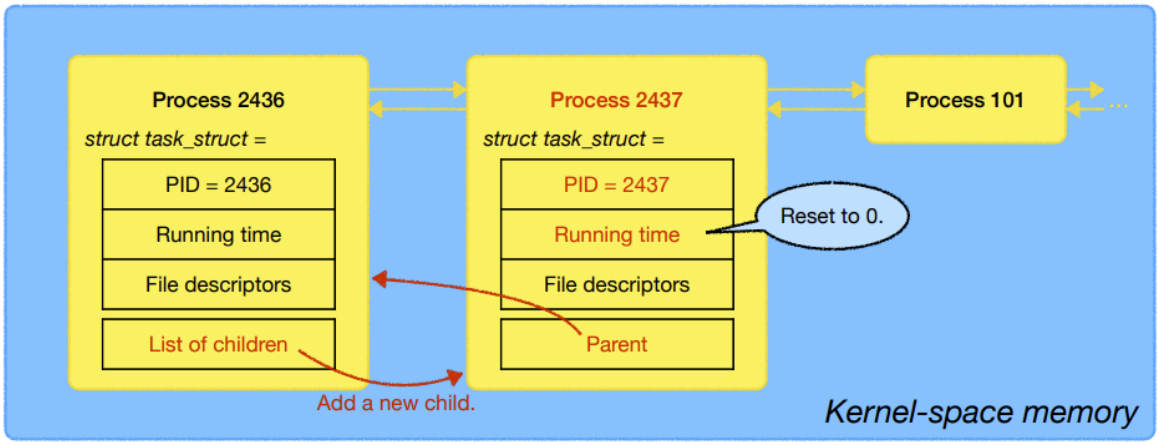


In the Kernal-space memory....

- Process control blocks before calling `fork()`
  - Process 2436

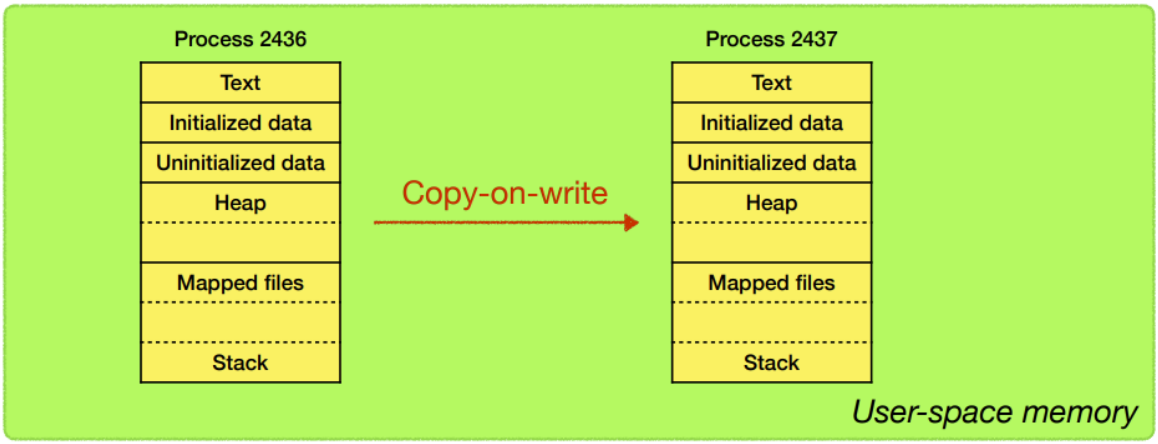


- After calling `fork()` a new process is created. Let the PID be 2437
  - A new task struct is created.
  - Inserted into the double linked list

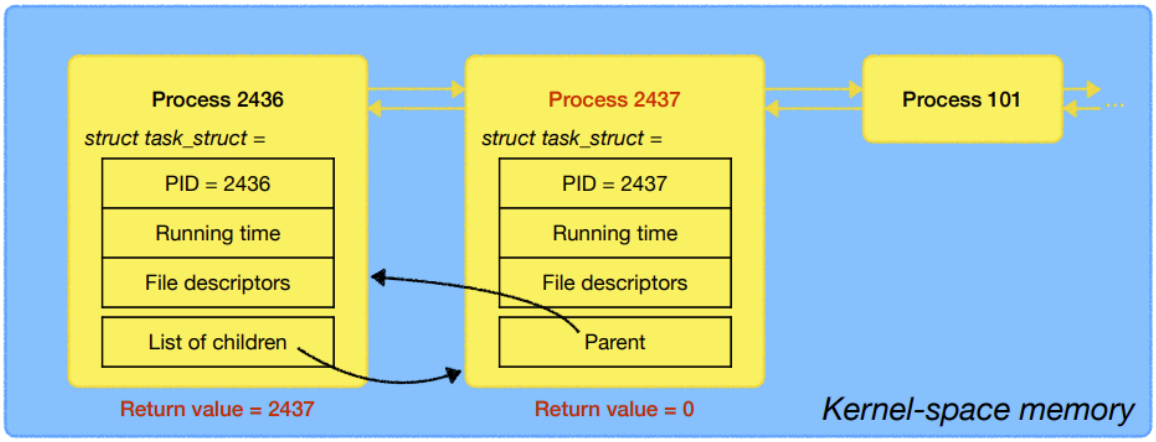


In the User-space memory....

- Copy-on-write
  - Not really copied until you actually modify something



- Another difference is the return value of `fork()`.
- In the parent process, the returned value is the PID of the child
- In the child process, the returned value is 0



What happen if two process writes to the same file (Race Condition)

(00:40:00 -)

What if two processes, sharing the same opened file, write to that file together?

```c
int main() {
  FILE *fp = fopen("file.txt", "w");

  pid_t pid = fork();
  for (int i = 0; i < 5; ++i) {
    fprintf(fp, "[%d] %d\n", getpid(), i);
    fflush(fp);
    sleep(rand() % 2);
  }
  fclose(fp);

  if (pid)
    wait(NULL);
}
```

fork_fd.c

```
$ cat file.txt
[2436] 0
[2437] 0
[2436] 1
[2436] 2
[2437] 1
[2437] 2
[2436] 3
[2437] 3
[2436] 4
[2437] 4
$ _
```

> “
>
> - Used `fflush()` because writing to a file is fully buffered
> - After printing each line, it randomly chooses to sleep for a second or not

- `if (pid)` tells if it's the parent process, because if it's the parent process, pid is not 0.
  - If i'm a parent, I wait until the children terminates. So that I can make sure both parent and child has printed 5 lines

- Run:

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

int main() {
    FILE *fp = fopen("file.txt", "w");

    pid_t pid = fork();

    int i;
    for (i = 0; i < 5; ++i) {
        fprintf(fp, "[%d] %d\n", getpid(), i);
        fflush(fp);
        sleep(rand() % 2);
    }
    fclose(fp);

    if (pid)
        wait(NULL);
}
```

- We get

```
[23033] 0
[23034] 0
[23033] 1
[23033] 2
[23034] 1
[23034] 2
[23033] 3
[23034] 3
[23033] 4
[23034] 4
```

  - Note that for a single process (like 23033), the number is printed in order

- If we remove the `sleep()` line

```
[23178] 0
[23178] 1
[23178] 2
[23178] 3
[23178] 4
[23179] 0
[23179] 1
[23179] 2
[23179] 3
[23179] 4
```

    - This is because the program is too short
        - The scheduler has not get the chance to interrupt the running
        - The parent process runs until it terminates, and the scheduler kicks in and start another process
        - If the program is long enough, the scheduler will come in and interupt your execution and start another process
    - The order of process is not deterministic
        - The scheduler could choose to run the child process immediately after the `fork()`
- If we remove the `wait()`

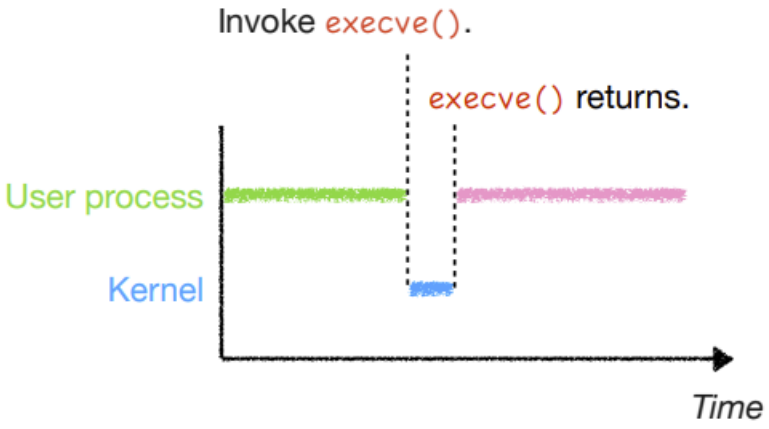
```
[24222] 0
[24223] 0
[24222] 1
[24222] 2
[24223] 1
[24223] 2
[24222] 3
[24223] 3
[24222] 4
[24223] 4
```

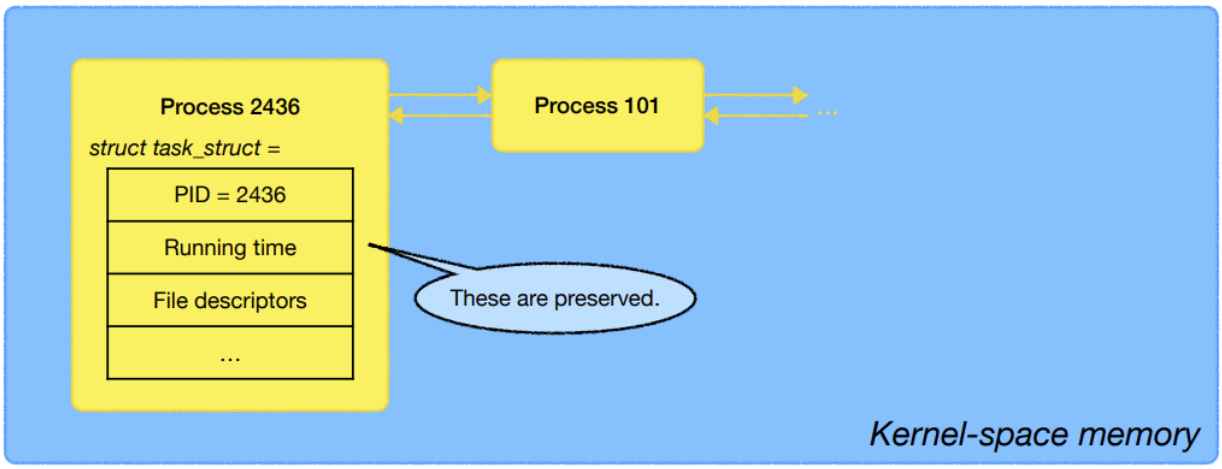
    - the program should terminate when the parent process reaches the end of the program

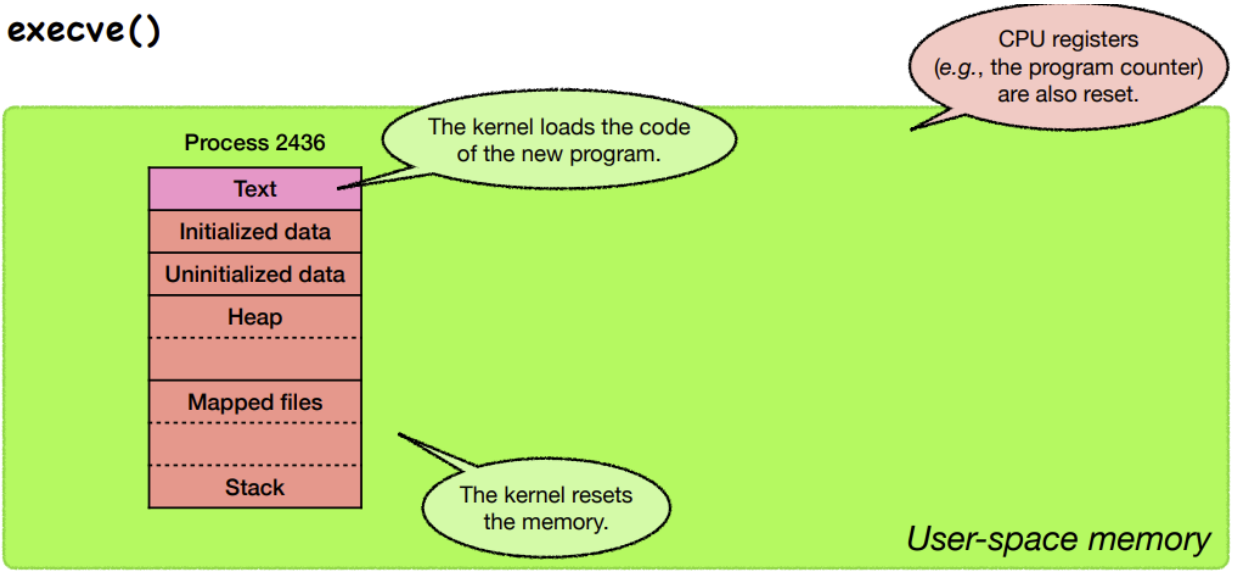
> “
>
> This is what we called a **race condition**

- **`execve()`**

From a programmer's perspective…





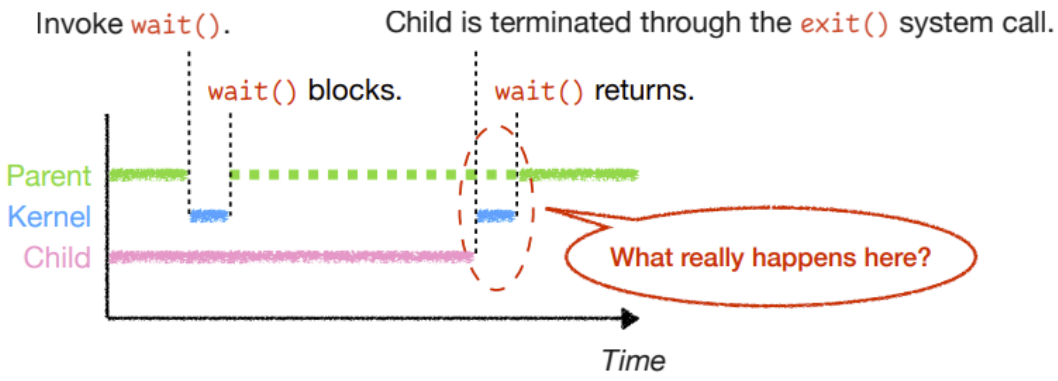


> "
>
> The task_struct in the kernel space memory is preserved
>
> But in the user-space memory, the entire stuff would be replaced by the new program's code
>
> - The code is changed

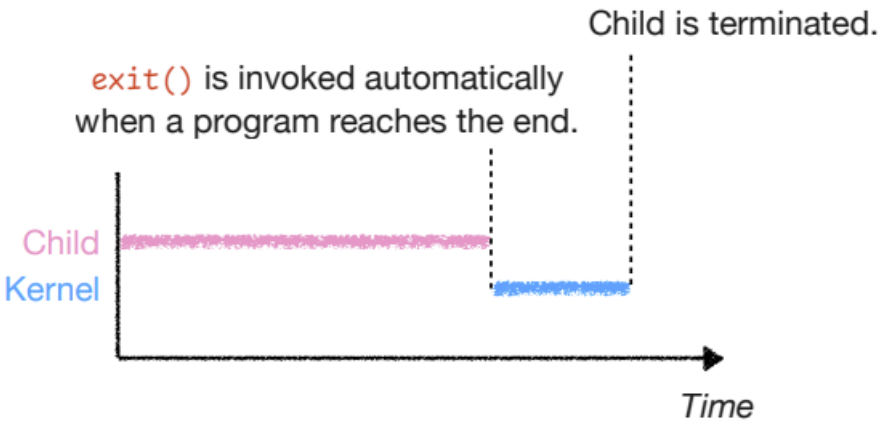
- Allocated memory changed
- Stack is reset
- Data is reset
- CPU registers (Program counter) are also reset

• `wait()` and `waitpid()`

From a programmer's perspective…



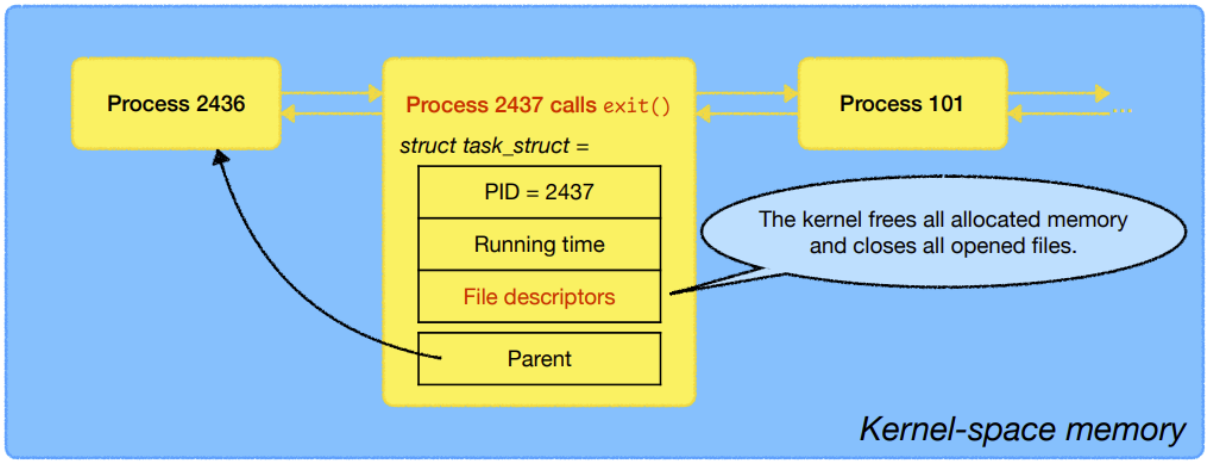
To understand `wait()`, we have to look at `exit()` first…

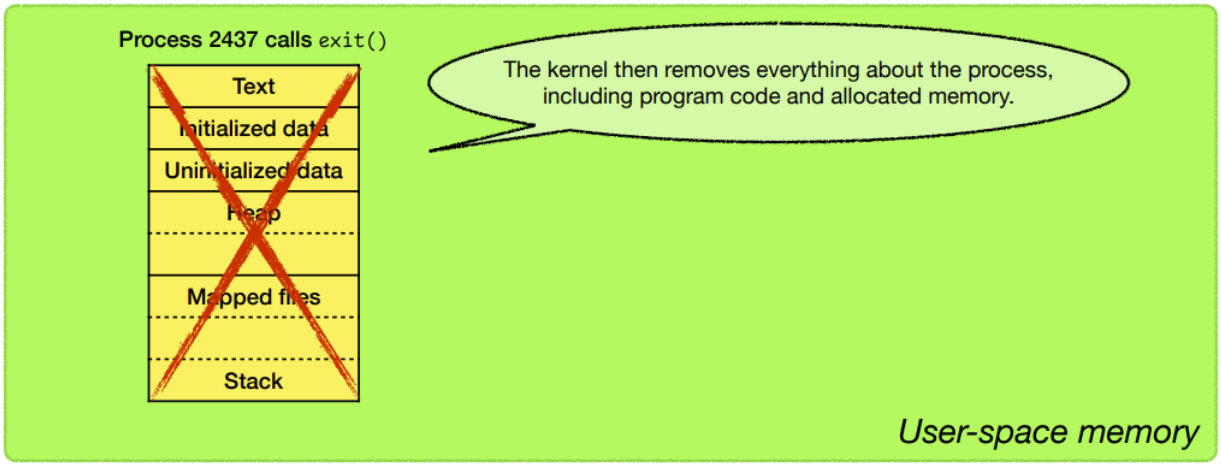


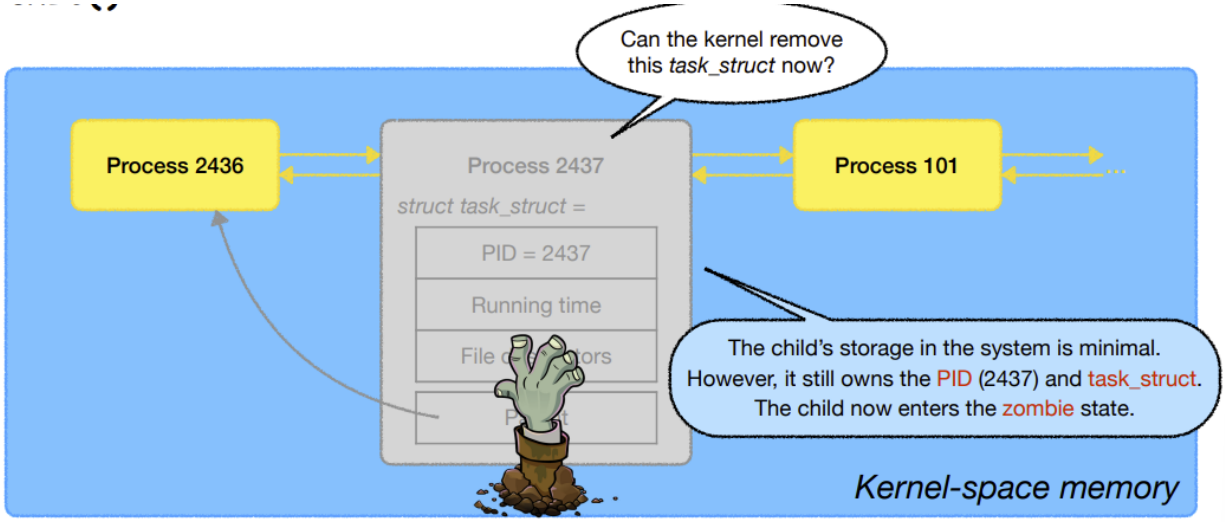
> "
>
> We need a way for **Interprocess Communication**

- `exit()`

exit()

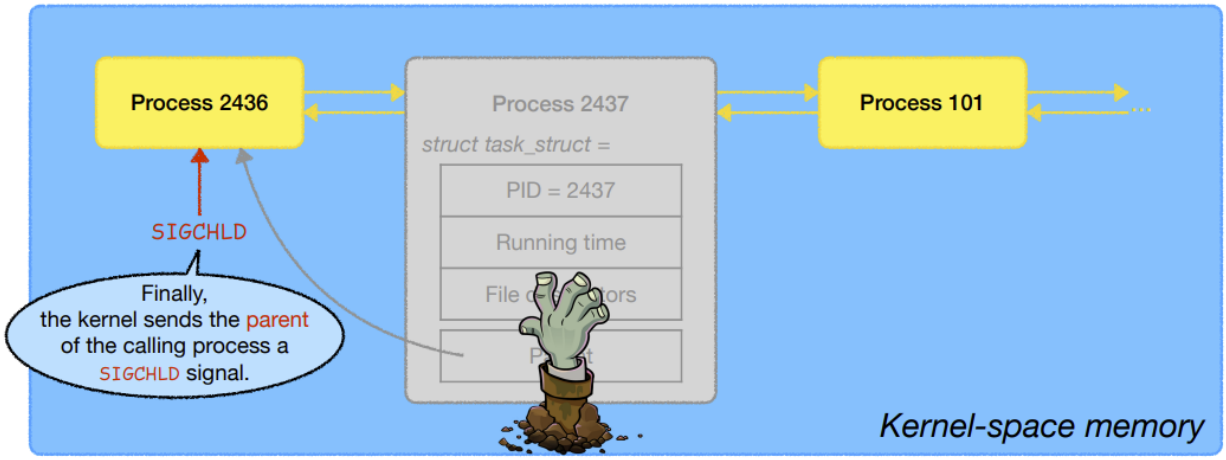




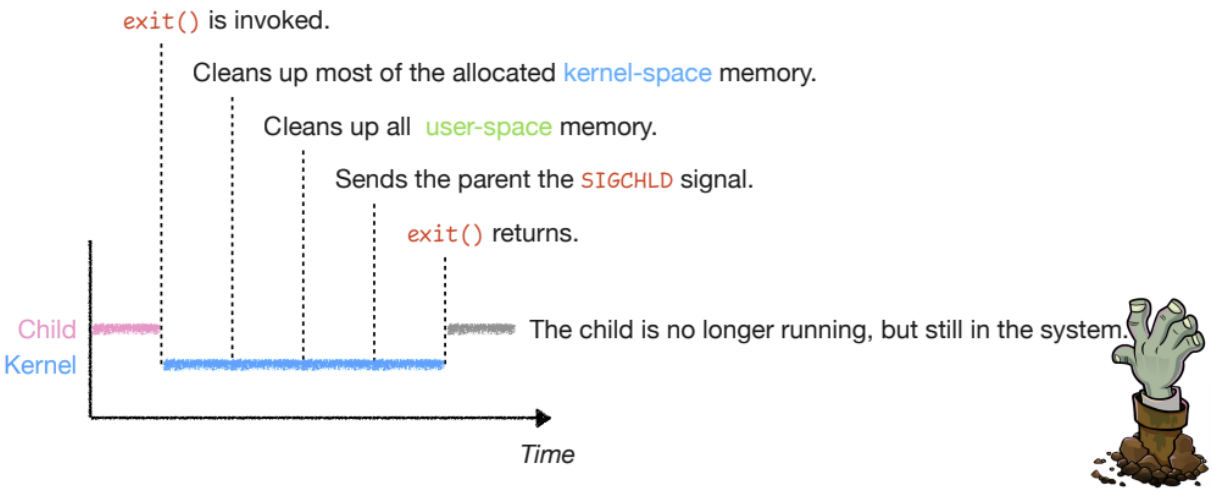


- We shouldn't remove the *task_struct* of the process
  - The parents maybe still waiting for the status, and the exit code.
  - Therefore you need a place to store the status and exit code
    - the *task_struct*
- This is what we called a "**zombie**" state
  - A process that's already terminated

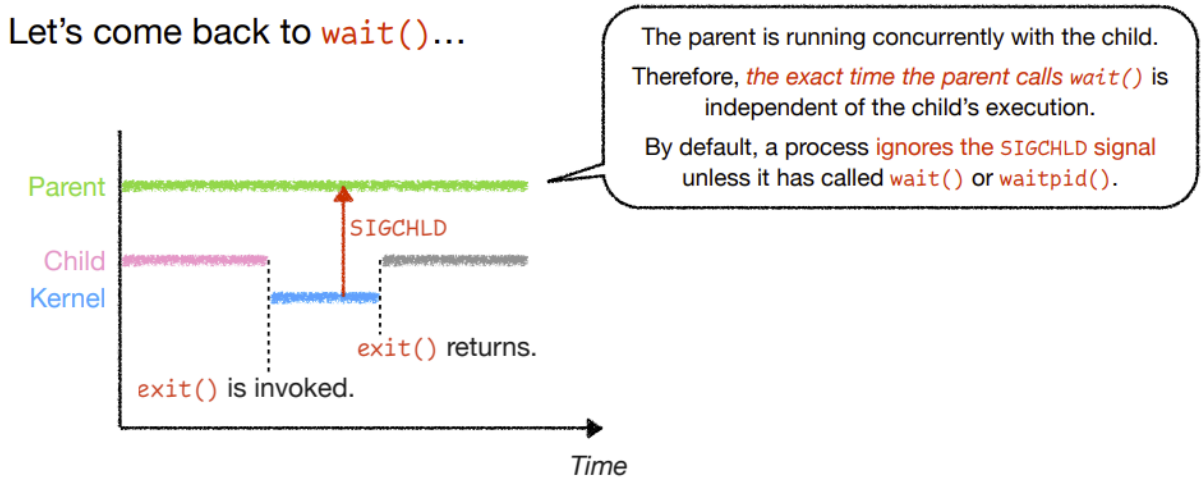
- but not yet removed from the kernel data structure



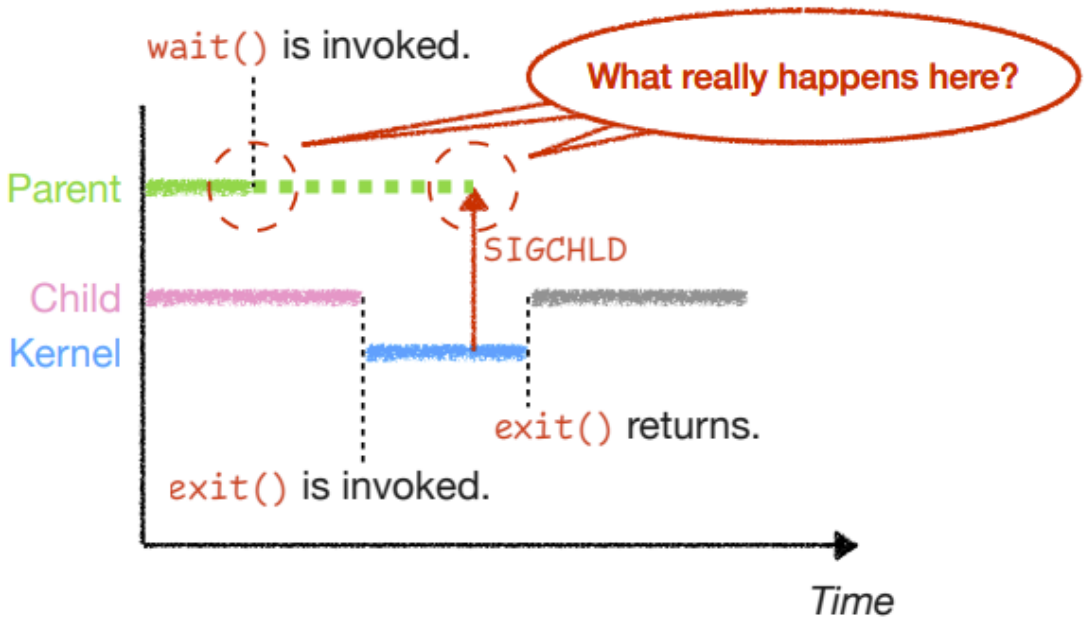
> `SIGCHLD` is a **signal** that tells the parents that the child has terminated
>
> This is what we called the **Interprocess Communication**



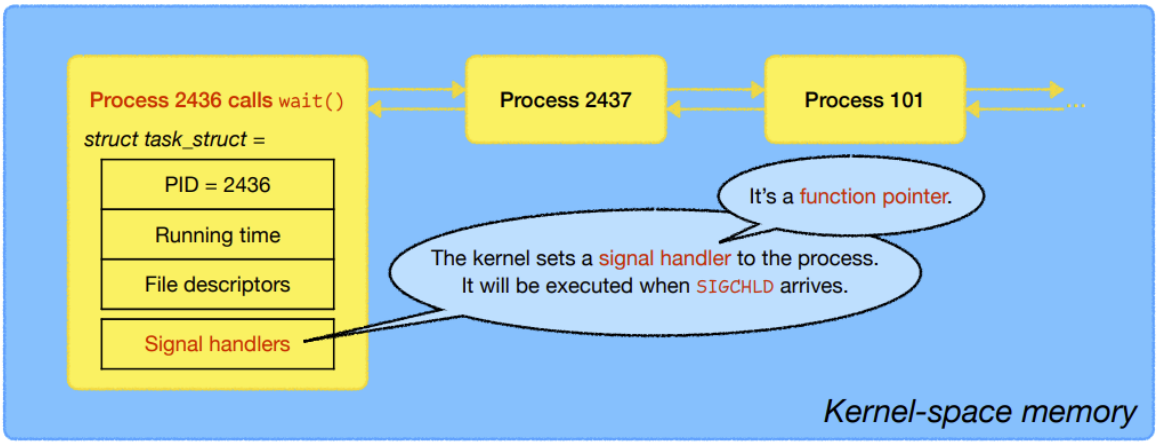
- Come back to `wait()`



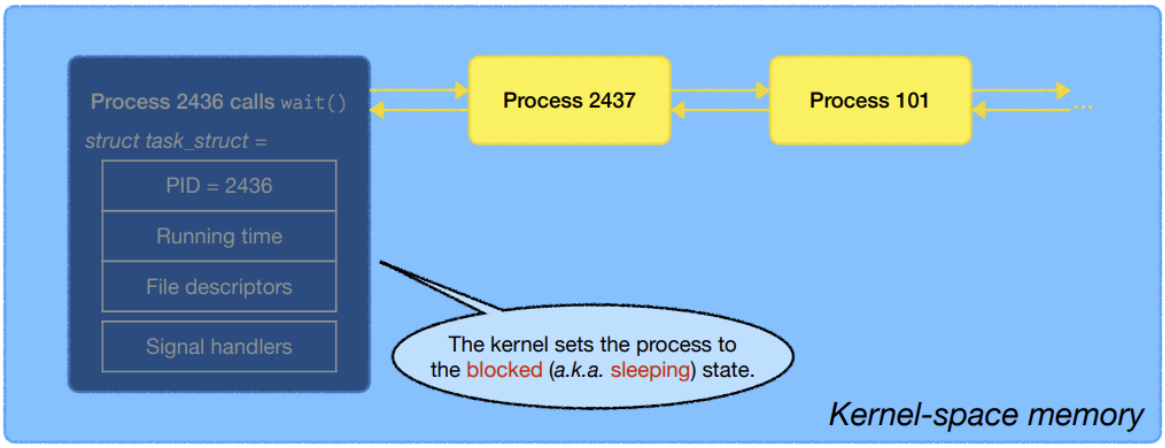
- Case 1: `wait()` before `SIGCHLD` arrives



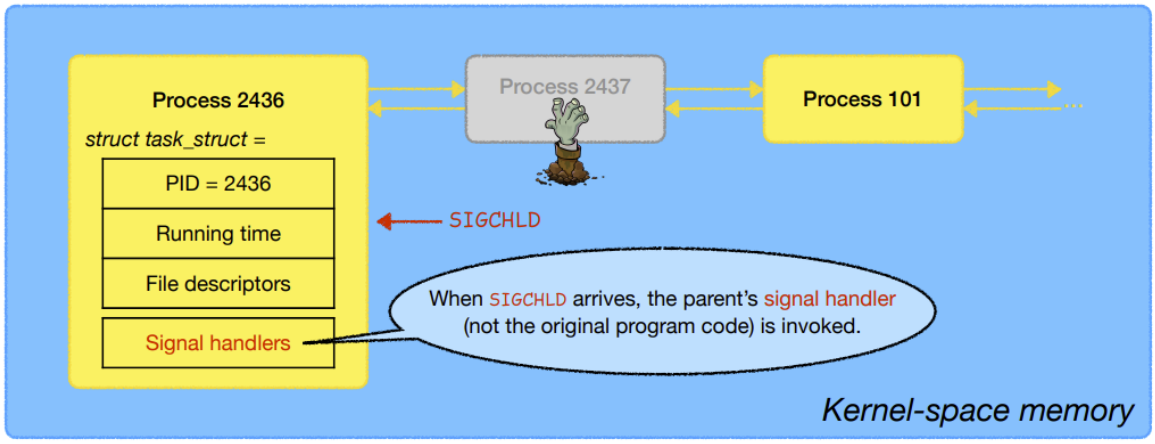
- The *task_struct* has a section called Signal handlers, which executes the signals
  - It's a function pointer



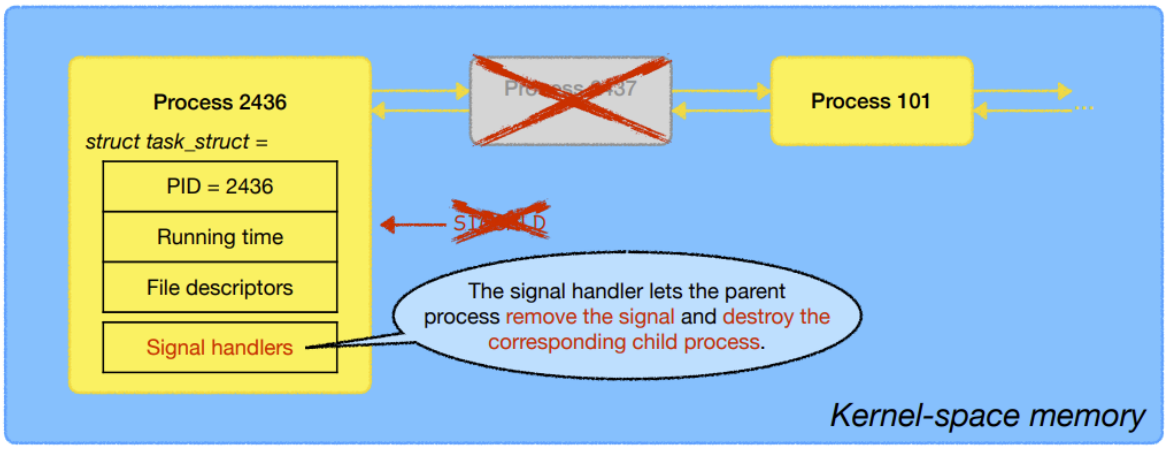
- When the process calls `wait()`, the kernel sets the process to a blocked (sleeping) state



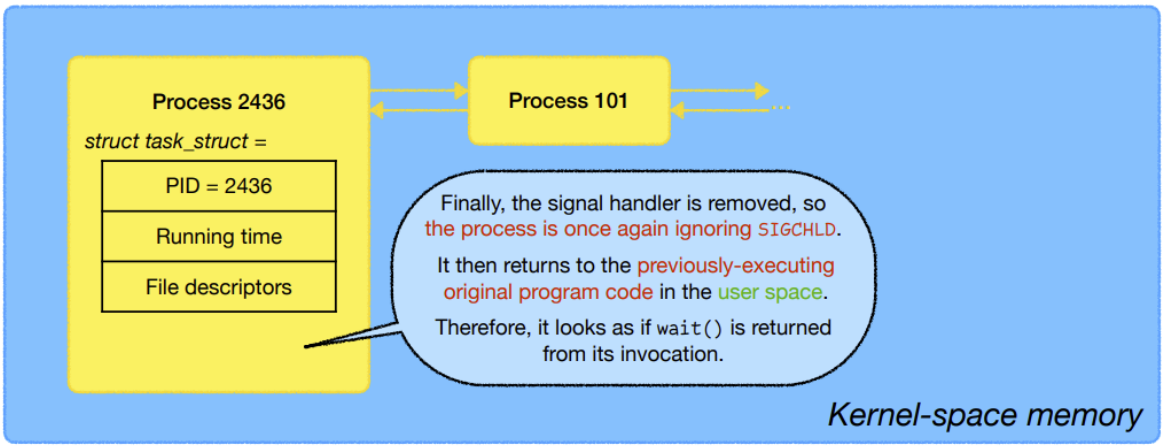
- When `SIGCHLD` arrives, the parent's signal handler(not the original program code) is invoked.
  - executes the stuff pointed by the signal handler

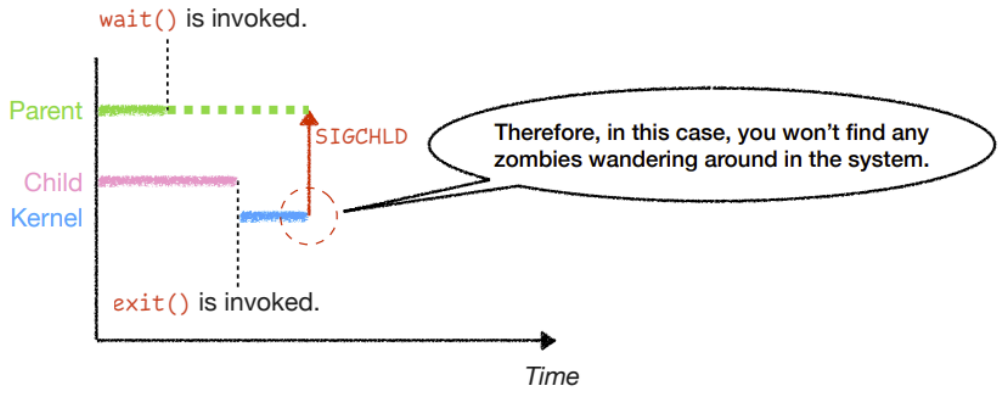


- Then the signal handler can let the parent process to remove the signal and destroy the child process
- In this way, the parent process can read the status of the program, and have the exit code of the child process



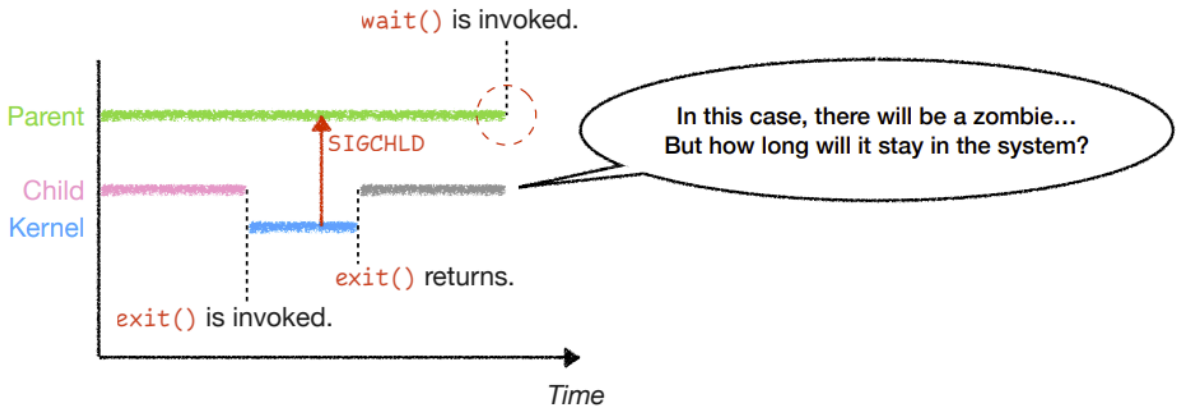
- Finally, the signal handler is removed, so the process is once again ignoring SIGCHLD.
- It then returns to the previously-executing original program code in the user space.
- Therefore, it looks as if `wait()` is returned from its invocation.
  - Program code after this `wait()` would be executed

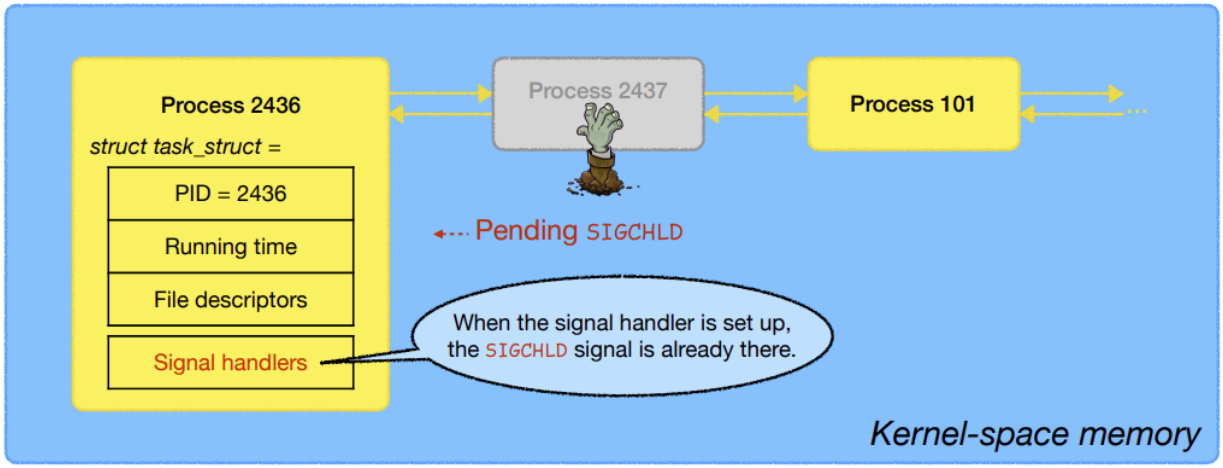


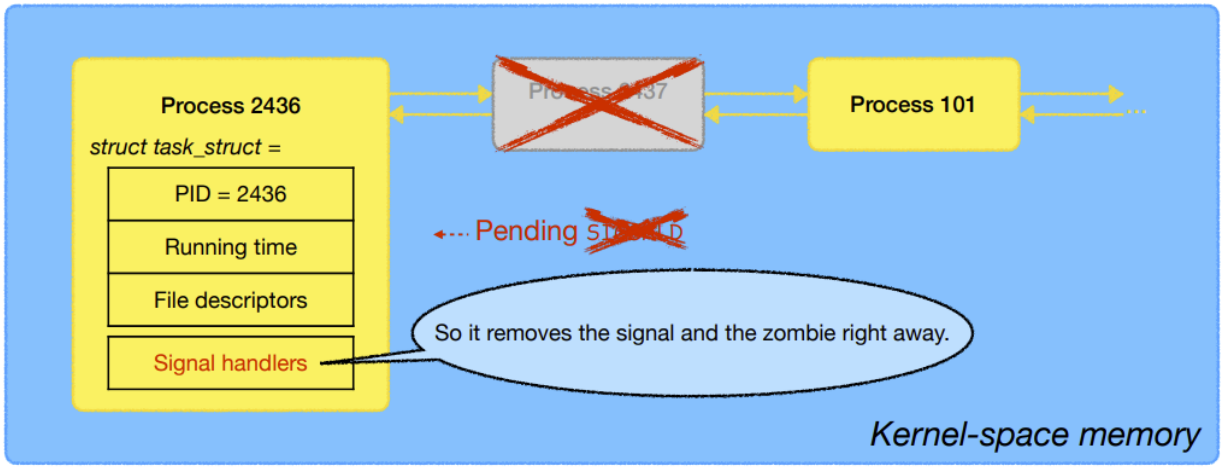


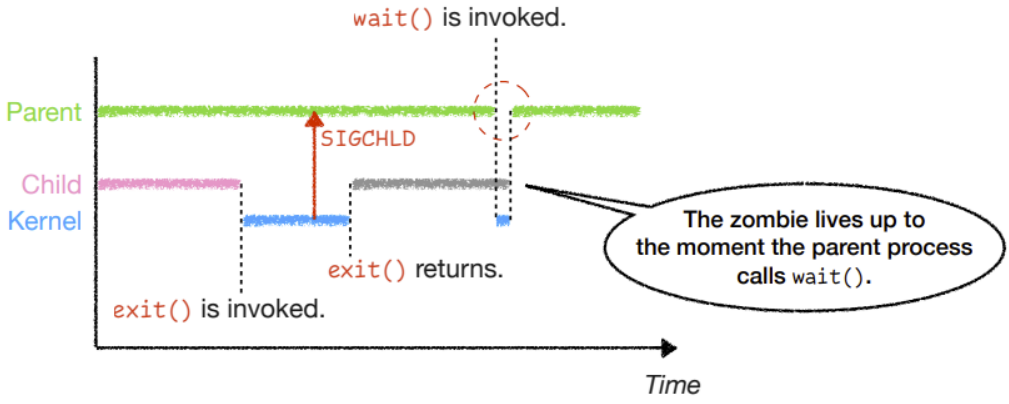
- No zombie in this case
- Because the a SIGCHLD is sent to the parent, and the zombie is taken care immediately

- Case 2: `wait()` after `SIGCHLD` arrives




Process 2436
struct task_struct =
PID = 2436
Running time
File descriptors
Signal handlers
Process 2437
Process 101
Pending SIGCHLD
When the signal handler is set up, the SIGCHLD signal is already there.
Kernel-space memory


Process 2436
struct task_struct =
PID = 2436
Running time
File descriptors
Signal handlers
Process 101
So it removes the signal and the zombie right away.
Kernel-space memory


wait() is invoked.
Parent
SIGCHLD
Child
Kernel
exit() returns.
exit() is invoked.
The zombie lives up to the moment the parent process calls wait().
Time

- Zombies

`wait()` and `waitpid()` are to reap zombie children.

You should never leave any zombies in the system. Why?

```
$ ./zombie
Look at Process 2437...
Look again!
$ _
```

Linux will label zombie processes as "`<defunct>`".

```c
int main() {
  pid_t pid;
  if ((pid = fork()) != 0) {
    printf("Look at Process %d...", pid);
    getchar();  // Press ENTER to continue
    wait(NULL);
    printf("Look again!");
    getchar();  // Press ENTER to continue
  }
}
```

zombie.c

```
$ ps x
  PID TTY      STAT   TIME COMMAND
 2436 pts/0    SN+    0:00 ./zombie
 2437 pts/0    ZN+    0:00 [zombie] <defunct>
$ ps x
  PID TTY      STAT   TIME COMMAND
 2436 pts/0    SN+    0:00 ./zombie
$ _
```

- We didn't do anything in the child process, so it will terminate immediately, before the parent e
- Before calling `wait()`, you can tell that there's a zombie
  - labeled `<defunct>` in the system
- After calling `wait()`, there's no zombie
- Note:
  - If the parent never calls `wait()`, the zombie will stay in the system until the parent process terminates