OS Lecture 4

Process

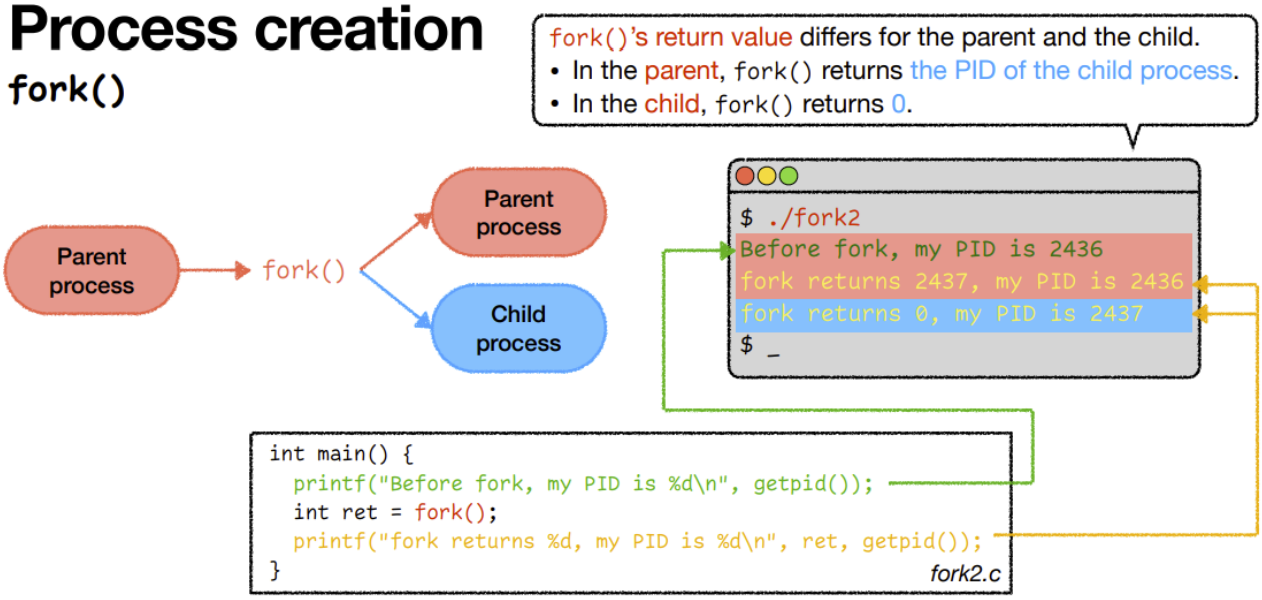
• Process creation: `fork()` (Review)

`fork()` 's return value differs for the parent and the child.

- In the parent, `fork()` returns the PID of the child process.
- In the child, `fork()` returns 0.

Process creation

`fork()`



- `fork()` Behaves like "cloning."

The child inherits (but is independent from) the parent's…

- Program code
  - Both the parent and child share the same code.
- Program counter
  - Therefore, both the parent and the child execute from the same location after `fork()`.
- Memory
  - This includes global variables, local variables, and dynamically allocated memory.
- Opened files
  - If the parent has opened a file, then the child also has the same file opened.

However, the child differs from the parent in a few things…

- Return value of fork()
  - The parent returns the PID of the child, or -1 if `fork()` fails. The child returns 0.
- Process ID
  - The child gets a new PID, which is not necessarily the parent's PID + 1.
- Parent
  - The child process's parent is the parent process, not the grandparent.
- Running time
  - The child's running time is reset to 0.
- File locks
  - The child does not inherit file locks from its parent.

Process creation

fork() behaves like “cloning.”

Challenge: what will be the output?

```
int main() {
  printf("Hello ");
  fork();
  printf("CS202\n");
}
```
fork_buffer.c

The library function `printf()` invokes the `write()` system call.

There is a buffer in the `FILE` structure to reduce the number of system calls.

At `fork()`, the child inherits the buffer.

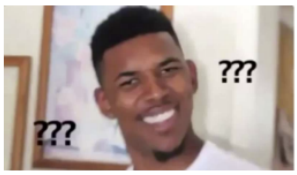
Unbuffered: invoke `write()` immediately.
- `stderr` is unbuffered by default.

Line-buffered: write data to the buffer. Invoke `write()` when a newline character is encountered.
- `stdin` and `stdout` are line-buffered by default.

Fully-buffered: write data to the buffer. Invoke `write()` when the buffer becomes full or before the process terminates.

You can call `setvbuf()` to change the buffering strategy or call `fflush()` to force a write.

```
$ ./fork_buffer
Hello CS202
Hello CS202
$ _
```




```
int main() {
  printf("Hello ");
  fflush(stdout);
  fork();
  printf("CS202\n");
}
```
fork_buffer2.c

```
$ ./fork_buffer2
Hello CS202
CS202
$ _
```

> “
>
> Use `fflush()` to force the `write()` system call

Who runs first?

- On a single-processor system, only one process can be executed at one time.
- After `fork()`, does the parent or the child run first?
  - .....We don’t know.
- The OS has a mechanism called process scheduling.
  - It decides which process to run.

We will discuss it later.

However, fork() seems useless…

If a process can only duplicate itself, *how* can we execute other programs?

- The `execve()` system call executes a program.

```
1   int execve(const char *filename, char *const argv[], char *const envp[]);
```

- It replaces the current process image with the new program.

Working with `execve()` is tedious.

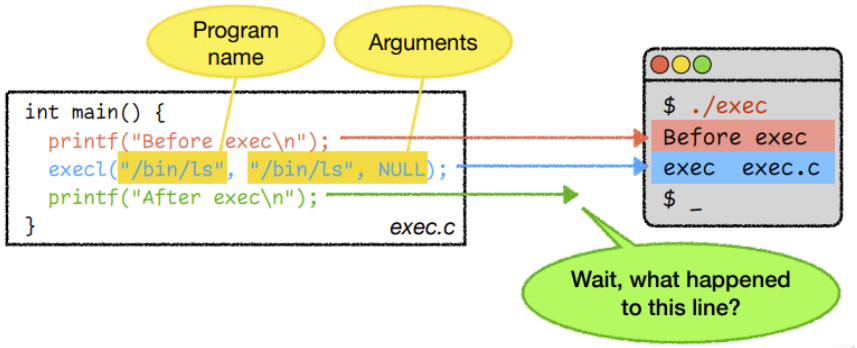
Luckily, we have a family of wrapper functions:

execl	execlp	execle	execv	execvp	execve

Let’s look at a simpler one: `execl()`.

- Process execution: `execve()` and the `exec*()` family of functions

- `execl()`



- Example:

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

int main() {
  printf("Before exec\n");
  execl("/bin/ls", "/bin/ls", NULL);
  printf("After exec\n");
}
[yt2475@linserv1 proc]$ gcc -o exec exec.c
[yt2475@linserv1 proc]$ ./exec
Before exec
envp.c  fork2    fork_buffer     fork_buffer3.c  header.h      my_system.c  pid.c       stupid.c  time1.c
exec    fork2.c  fork_buffer2    fork_buffer.c   loop.c        orphan.c     redirect.c  stupid.s  time2.c
exec.c  fork3    fork_buffer2.c  fork.c          main.c        pause.c      shm.c       system.c  zombie.c
fork    fork3.c  fork_buffer3    fork_fd.c       my_system2.c  pid          sig.c       thread.c
[yt2475@linserv1 proc]$
```

The After Exec line is missing!

- `ececve()`

When `execve()` is called, the process replaces the code that it's executing with the new program and never returns to the original code.

- Nothing after `execve()` will be executed.

The process **discards**…

- Memory: global variables, local variables, and dynamically allocated memory
- Registers: program counter...

The process **preserves**…

- Process ID
- Process relationship
- Running time…

The `exec*()` family of functions

execl	execlp	execle	execv	execvp	execve

Path name or file name

- Default — path name, e.g., "/bin/ls"
- p — file name, e.g., "ls"

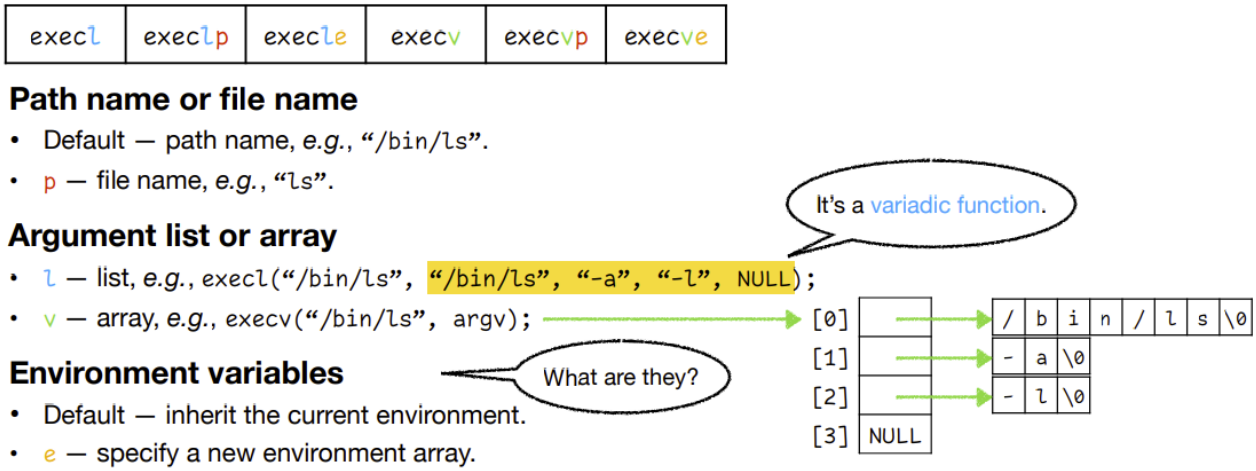
Argument list or array

(It's a variadic function.)

- l— list, e.g., execl("/bin/ls", "/bin/ls", "-a", "-l", NULL);
- v — array, e.g., execv("/bin/ls", argv);

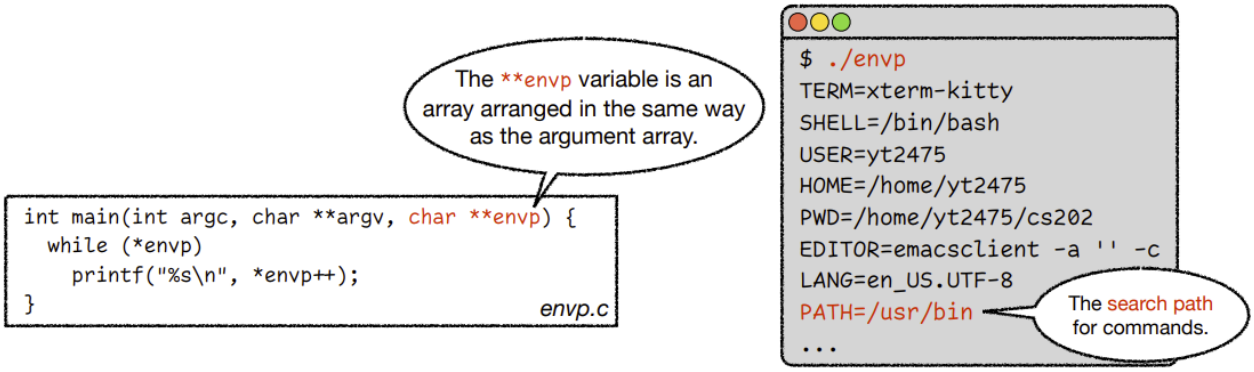
Environment variables

- Default — inherit the current environment.
- e — specify a new environment array



Environment variables

- Environment variables are a set of strings maintained by the shell.
- Many programs will read and make use of environment variables.



> "
>
> PATH tells the shell where to search for a program when you type in a program name, not a path.
>
> There may be multiple paths

- Example

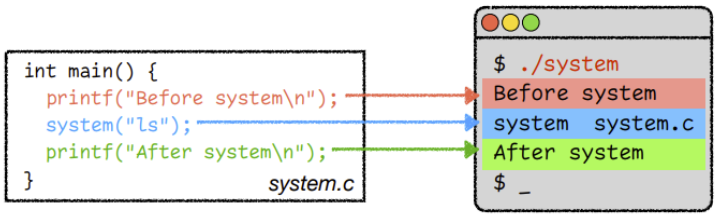
```
[yt2475@linserv1 proc]$ cat envp.c
#include <stdio.h>

int main(int argc, char **argv, char **envp) {
  while (*envp)
    printf("%s\n", *envp++);
}
[yt2475@linserv1 proc]$ gcc -o envp envp
```

```
XDG_SESSION_ID=12956
HOSTNAME=linserv1.cims.nyu.edu
LESS_TERMCAP_md=
SELINUX_ROLE_REQUESTED=
LESS_TERMCAP_me=
TERM=xterm-kitty
SHELL=/bin/bash
HISTSIZE=100000
SSH_CLIENT=216.165.22.146 39966 22
LIBRARY_PATH=/usr/local/stow/gcc-12.2-nvptx/lib64
NCARG_FONTCAPS=/usr/lib64/ncarg/fontcaps
SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib64/qt-3.3
OLDPWD=/home/yt2475
LESS_TERMCAP_ue=
QTINC=/usr/lib64/qt-3.3/include
SSH_TTY=/dev/pts/9
QT_GRAPHICSSYSTEM_CHECKED=1
HISTFILESIZE=100000
USER=yt2475
LD_LIBRARY_PATH=/usr/local/stow/gcc-12.2-nvptx/lib64
LS_COLORS=
TECHOME=/opt/tecplot
CPATH=/usr/local/stow/gcc-12.2-nvptx/include
```

• Process creation and execution: `System()`

- Can we just run a program and be able to come back?
  - Yes, there is a convenient library function: `system()`.



  - It is implemented using `fork()` and `exec()` to call `/bin/sh -c command`.

> “
>
> So System itdelf doesn't do anything
>
> It just put another command as an argument to the shell, an then calls the shell to run the program.

- Implementing `system()`

- Can we do it ourselves?

```c
int my_system(const char *command) {
  if (fork() == 0) {
    execl("/bin/sh", "/bin/sh", "-c", command, NULL);
    exit(-1);
  }

  return 0;
}

int main() {
  printf("Before system\n");
  my_system("ls");
  printf("After system\n");
}
```

my_system.c

- What's the **problem** of it?
  - If child runs first - as we desire

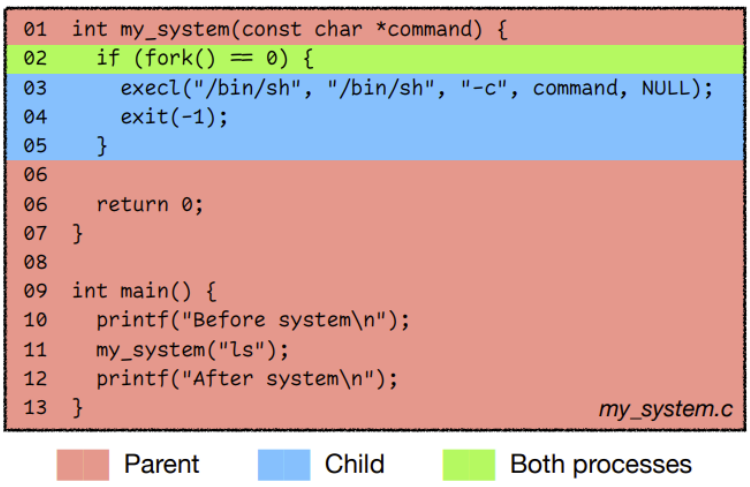
```
$ ./my_system
Before system
system  system.c
After system
```

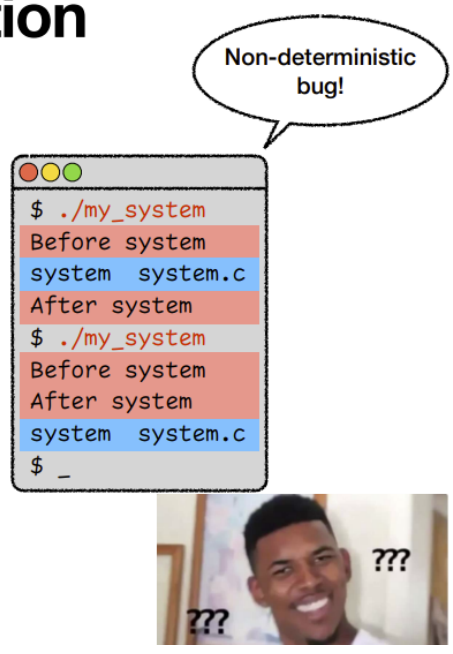
  - If the parent runs first - Problem!

```
$ ./my_system
Before system
After system
system  system.c
```

Process creation and execution

Implementing `system()`





How to let the child run first?

- Unfortunately, we cannot control the OS's process scheduling to this extent.

But, is this our real issue?

- We should pause the parent process while the child is running!

So, our real issue is…

- How to suspend the execution of the parent process?
- How to wake up the parent after the child is terminated?

Implementing `system()` with `wait()`

```
01  int my_system(const char *command) {
02    if (fork() == 0) {
03      execl("/bin/sh", "/bin/sh", "-c", command, NULL);
04      exit(-1);
05    }
06    wait(NULL);
06    return 0;
07  }
08
09  int main() {
10    printf("Before system\n");
11    my_system("ls");
12    printf("After system\n");
13  }
```

Can we remove this `exit()`?

my_system.c

The `wait()` system call suspends execution of the calling process until one of its children terminates.

- What if it has no running children?
  - Nothing happens. It returns immediately
- If passing a non-NULL argument, it will be filled with the child's status, such as the exit code.
  Passing NULL means "I don't care."
- `waitpid()` is a more general version.
  - Wait for a particular child.
  - Wait for a stopped/resumed child.

See "man 2 wait" for more details.

> "
>
> What's the purpose of line 4: `exit(-1)` ?
>
> If line 3: `execl()` succeed, line e will never be executed
>
> If line 3 failed, it won't execute the new program. It'll return -1 that needs to be handled.
>
> If line 3 failed and you don't have line 4.....

```c
#include <unistd.h>

int my_system(const char *command) {
  if (fork() == 0) {
    execl("/bin/shh", "/bin/sh", "-c", command, NULL);
    // exit(-1);
  }
  printf("wait() returns %d\n", wait(NULL));
  return 0;
}

int main() {
  printf("Before system\n");
  my_system("ls");
  printf("After system\n");
}
[yt2475@linserv1 proc]$ gcc -o my_system2 my_system2.c
[yt2475@linserv1 proc]$ ./my_system2
Before system
wait() returns -1
After system
wait() returns 36814
After system
```

Note that the `execl()` failed, and the child process keeps running, go through the if statement, and printed `wait()`.

Since the child doesn't have children, `wait()` returns -1.

Then child returns to the main and also prints After system.

If we add the line 4, it'll print whenn `execl()` fails:

```
[yt2475@linserv1 proc]$ ./my_system2
Before system
wait() returns 36993
After system
```

Reading man pages

Man pages (i.e., manual pages) are of vital importance for programmers working on Linux and such.

They include important information…

- Header files to be included.
- Compiler flags to be added.
- Input arguments.
- Return values.
- Error conditions.

Man pages are divided into sections.

Example: "`man wait`" is the same as "`man 1 wait`".

- It means the shell command "`wait`".
- Use "`man 2 wait`" to read the manual page of the system call `wait()`.

See "`man man`" for the description of each section.

Section	Description
1	Executable programs or shell commands.
2	System calls (functions provided by the kernel).
3	Library calls (functions within program libraries).
...	...

- Example: `man 2 wait`

```
WAIT(2)                          Linux Programmer's Manual                          WAIT(2)

NAME
       wait, waitpid, waitid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status);

       pid_t waitpid(pid_t pid, int *status, int options);

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       waitid():
           _SVID_SOURCE || _XOPEN_SOURCE >= 500 || _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED
           || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L

DESCRIPTION
       All  of  these  system calls are used to wait for state changes in a child of the calling process, and
 Manual page wait(2) line 1 (press h for help or q to quit)
```

```
                    SUMMARY OF LESS COMMANDS

      Commands marked with * may be preceded by a number, N.
      Notes in parentheses indicate the behavior if N is given.
      A key preceded by a caret indicates the Ctrl key; thus ^K is ctrl-K.

  h  H                 Display this help.
  q  :q  Q  :Q  ZZ     Exit.
 ---------------------------------------------------------------------------

                           MOVING

  e  ^E  j  ^N  CR  *  Forward  one line   (or N lines).
  y  ^Y  k  ^K  ^P  *  Backward one line   (or N lines).
  f  ^F  ^V  SPACE  *  Forward  one window (or N lines).
  b  ^B  ESC-v      *  Backward one window (or N lines).
  z                 *  Forward  one window (and set window to N).
  w                 *  Backward one window (and set window to N).
  ESC-SPACE         *  Forward  one window, but don't stop at end-of-file.
  d  ^D             *  Forward  one half-window (and set half-window to N).
  u  ^U             *  Backward one half-window (and set half-window to N).
  ESC-)  RightArrow *  Left  one half screen width (or N positions).
```

```
      int
      main(int argc, char *argv[])
      {
          pid_t cpid, w;
          int status;

          cpid = fork();
          if (cpid == -1) {
              perror("fork");
              exit(EXIT_FAILURE);
          }

          if (cpid == 0) {            /* Code executed by child */
              printf("Child PID is %ld\n", (long) getpid());
              if (argc == 1)
                  pause();                    /* Wait for signals */
              _exit(atoi(argv[1]));

          } else {                    /* Code executed by parent */
              do {
                  w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
                  if (w == -1) {
Manual page wait(2) line 268 (press h for help or q to quit)
```

• Are `fork()` and `execve()` wasteful?

It seems wasteful that the child copies the parent's entire memory at `fork()` and immediately discards it at `exec()`.

In fact, after `fork()`, the child shares the parent's memory copy-on-write.

- As long as nobody modifies the shared memory, no copying is needed.
- Whenever a process wants to modify the memory, the OS makes a copy of that chunk of memory so that the modification can be done privately.

More on that later when we talk about the memory management of the OS.

- Summary

  - A process is created by cloning.
    - `fork()` is the system call that clones processes.
    - Cloning is copying — the new process inherits many things (but not all) from the parent process.
    - (But, where is the first process?)
  - Program execution is not trivial.
    - A process is the entity that hosts a program and runs it.
    - A process can run more than one program.
    - The `exec*()` system call family replaces the program that a process is running.

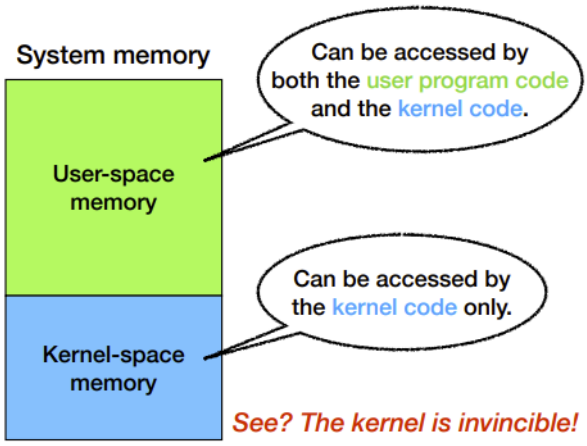
Processes in the kernel

- Kernel-Space Memory vs User-Space Memory

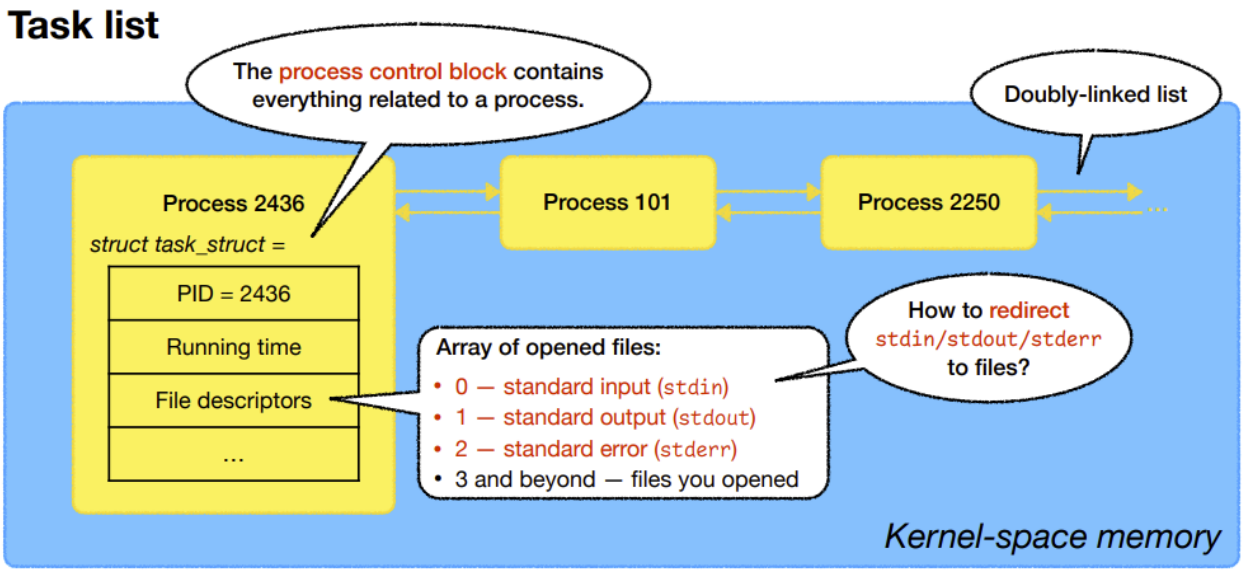
The memory is divided into two parts.

User-space memory stores…
- Program code
- Process's memory
- …

Kernel-space memory stores…
- Kernel code
- Kernel data structures
- Loaded device drivers
- …



- Task list



Redirect using file descriptors

> In Shell, we are able to redirect the input and output to a file:
>
> ```
> [yt2475@linserv1 proc]$ echo Hello world
> Hello world
> [yt2475@linserv1 proc]$ echo Hello world > file.txt
> [yt2475@linserv1 proc]$ cat file.txt
> Hello world
> ```
>
> How to do it in our codes?

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
$ _
```

- `dup2()` duplicates the file descriptor to file descriptor 1
  - So file descriptor 1 corresponds to "output.txt"
  - ```
    [yt2475@linserv1 proc]$ cat redirect.c
    #include <fcntl.h>
    #include <stdio.h>
    #include <unistd.h>

    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

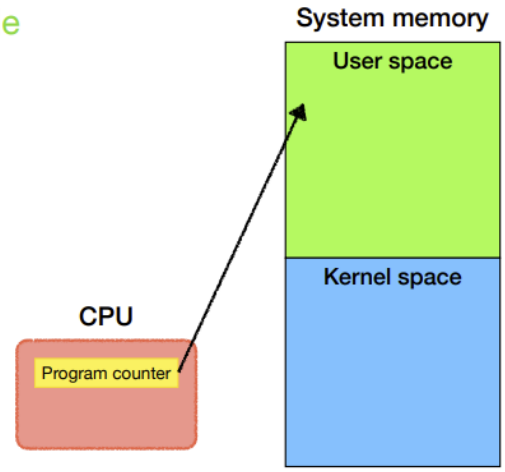
      printf("Hello CS202 again\n");
    }
    [yt2475@linserv1 proc]$ gcc -o redirect redirect.c
    [yt2475@linserv1 proc]$ ./redirect
    Hello CS202
    [yt2475@linserv1 proc]$ cat output.txt
    Hello CS202 again
    [yt2475@linserv1 proc]$
    ```
- It's the same for input. If you want to input from file, use `dup2(fd, 0)`
    ```

```
[yt2475@linserv1 proc]$ cat output.txt
Hello CS202 again
[yt2475@linserv1 proc]$ cat < output.txt
Hello CS202 again
[yt2475@linserv1 proc]$ cat < output.txt > output2.txt
[yt2475@linserv1 proc]$ cat output2.txt
Hello CS202 again
```

- 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()`

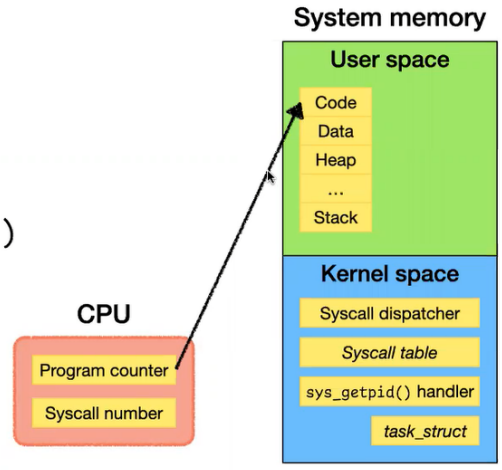
The CPU is running a process in user mode.

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.

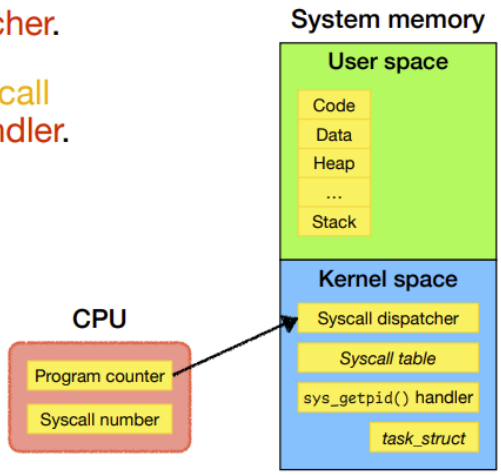


The kernel starts execution at the syscall dispatcher.

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.