A practical course on

Advanced systems programming in C/Rust

Dr. Atsushi Koshiba
Jörg Thalheim

Today’s topic!

Processes

Outline

- What is a process?
  - the concept of a process

- Process creation
  - fork(), exec(), wait()

- Inter-process communication
  - pipe()
  - signals

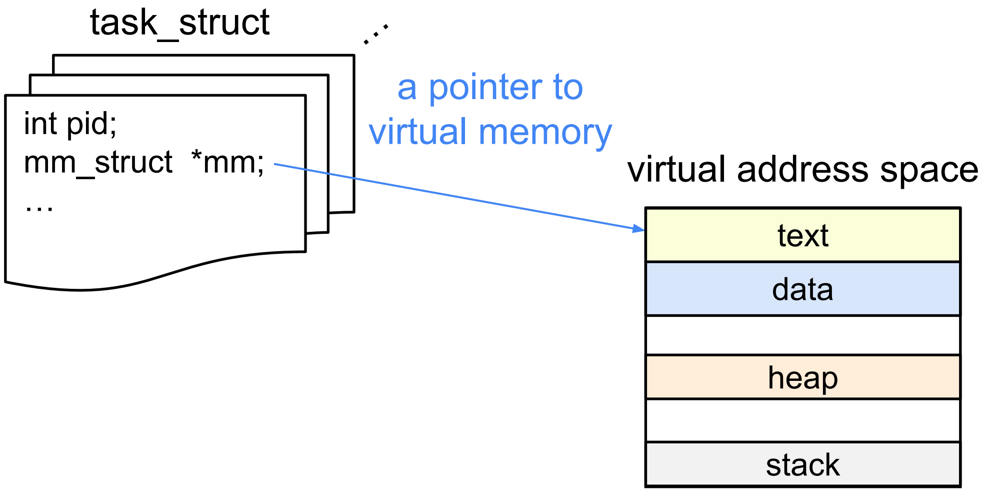
What is a process?

- represents a specific program running in multi-tasking OS (e.g., Linux)
  - OS manages each program as a process
  - each process has an independent (virtual) address space

- Merits
  - Multi-tasking
  - Resource sharing (i.e., CPU, memory, I/O devices)
  - Fault-tolerance – a process crash does not affect other processes

Process management

- Process ID (PID)
  - an identification number gave by OS when the process is created
- Process descriptor
  - a special data structure storing each process info (i.e., task_struct in Linux)
  - pid, state, address space, scheduling priority, file descriptors, …



Process creation

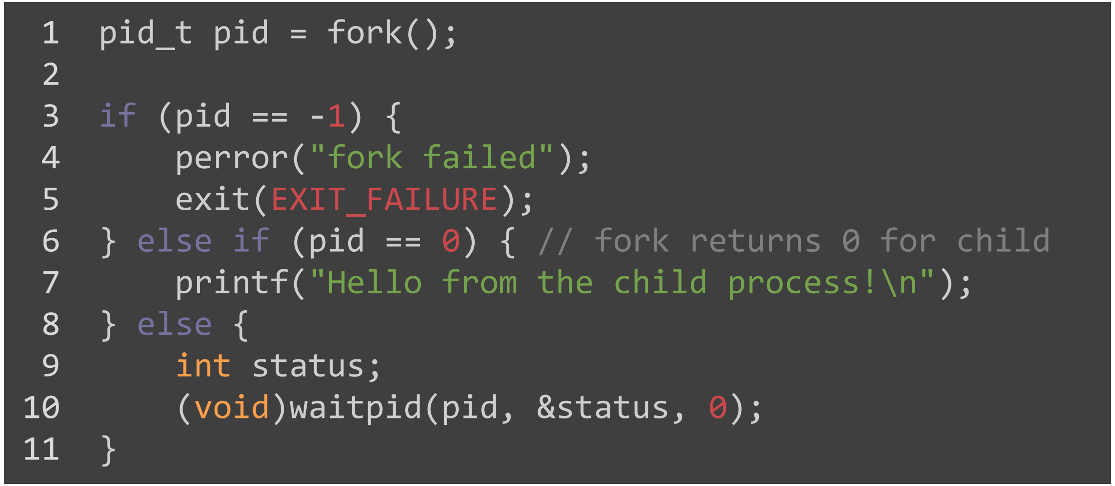
- We can use fork() system call
  - fork() does not ‘create’ a new process, but makes a copy of the caller process
  - The caller process is called *parent*, its duplicate is called *child*

- i.e., booting Linux kernel only creates the ‘init’ process
  - the init process is the parent of all the subsequent processes

- *Note*: ‘process’ and ‘thread’ are a bit different
  - a thread shares the virtual memory of its parent, but a process does not
  - we treat only the process here

fork()

- when fork() is called, the OS does:
  - replicate the entire virtual address space to a child
  - return a PID of the new child process
- waitpid() syscall
  - **int waitpid(pid_t** *pid*, **int ****status*, **int** *options***);**
  - wait for the exit of the child process specified by *pid*



```c
pid_t pid = fork();

if (pid == -1) {
    perror("fork failed");
    exit(EXIT_FAILURE);
} else if (pid == 0) { // fork returns 0 for child
    printf("Hello from the child process!\n");
} else {
    int status;
    (void)waitpid(pid, &status, 0);
}
```

Monitoring fork()

- fork() is originally implemented by clone() syscall
  - clone will also be later relevant when doing containers

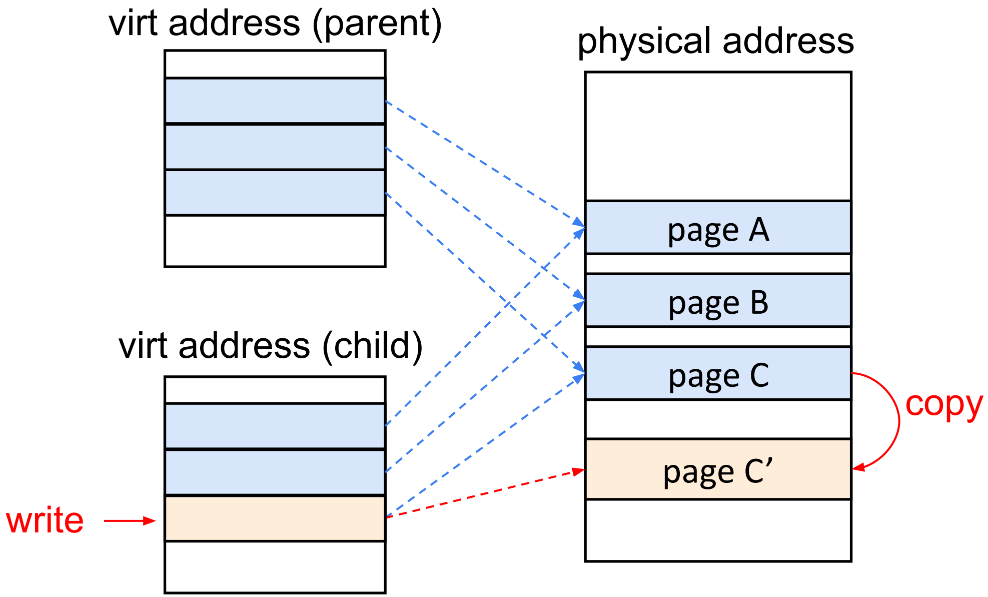
- *strace* can be used for monitoring
  - a tool to trace system calls and signals
  - -e option : specify which events to trace
  - -f option : trace child processes in addition to the parent

```
$ strace -e clone bash -c 'ls -la && echo $?'
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7fd6187c6a10) = 19855
```

PID of a child process

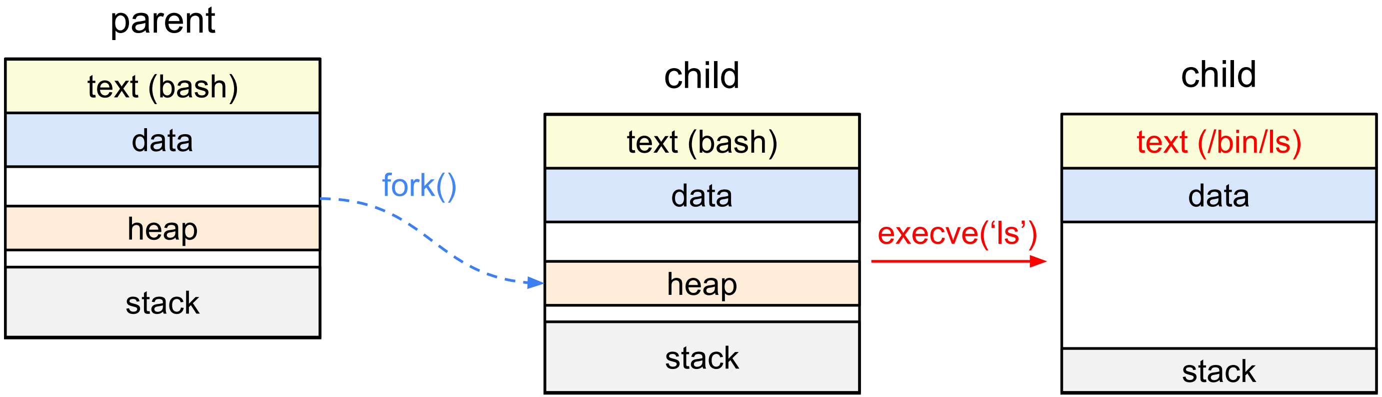
Copy on write

- contents of memory are not copied unless being modified
  - childs share all pages of the parent first
  - when a process modifies a page, a copy of the physical page is created



exec()

- how to execute a different program?
  - fork() just copies an entity of the parent process
- execve() syscall
  - **int execve (const char ****pathname***, char *const** *argv[]***, char *const** *envp[]***)**
  - resets virtual memory based on an executable file specified in *pathname* (i.e. elf file)
  - For scripts, OS reads shebangs from the file (#!) a setup's specified executable
    - #!/bin/sh



Inter-process communication (IPC)

- Introduce two IPCs
  - pipe : like a File I/O
  - signals : like an interrupt

- file descriptor (fd): index into a per-process file descriptor table
  - each process has three standard fds: stdin(0), stdout(1), stderr(2)
  - opened fds are inherited from the parent to a child
    - unless O_CLOEXEC is specified as flag

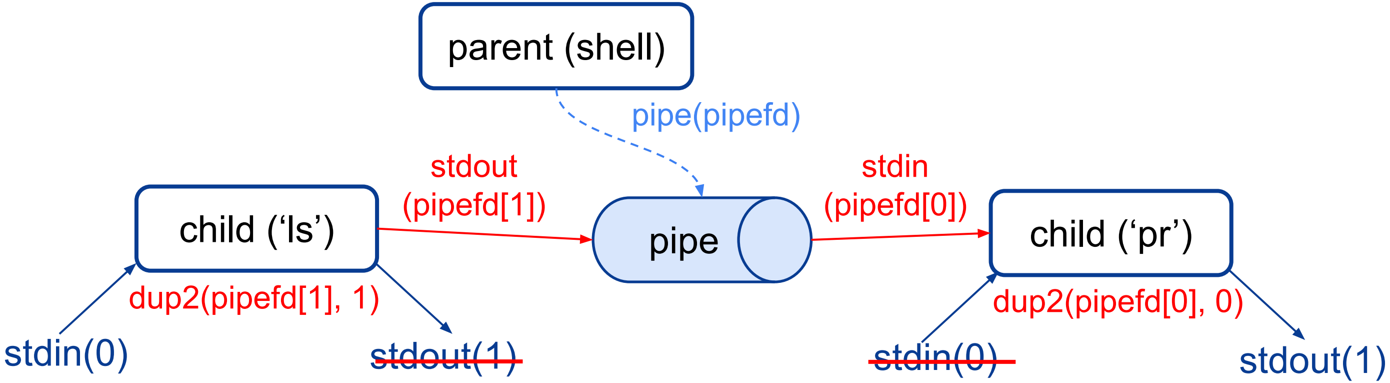
- dup2() syscall
  - **int dup2(int** ***oldfd*****, int** ***newfd*****)**
  - duplicate *oldfd* using the number specified by *newfd*
  - if *newfd* was previously open, it is silently closed

pipe()

- **int pipe2(int** pipefd[2], **int** flags**)**
  - produces a pipe, unidirectional byte stream
  - returns pipefd: pipefd[0] is a read end, pipefd[1] is a write end of the pipe
  - pipefds are not mapped to a file, but a physical page (VFS)

- dup2() is used to connect two processes via the pipe
  - e.g.,

```
$ ls | pr
```



Signals

- notify an asynchronous event to processes
  - e.g., a segmentation fault (SIGSEGV), an exit of child processes (SIGCHLD)
  - Signals are managed by a bitmask field (*long signal*) in task_struct
    - when SIGSEGV (11) is sent, the 11th bit of *signal* is set to 1
  - a process can block a specific signal
    - Only SIGSTOP (pause) and SIGKILL (terminate) cannot be blocked

- Signal handlers
  - When a process receives a non-blocked signal, a corresponding handler is invoked
  - Handlers are set by a process; otherwise, the OS performs the default action
  - e.g., waitpid() handles SIGCHLD

How to send signals

- kill() syscall
  - **int kill(pid_t** *pid*, **int** *sig*)
  - send the signal *sig* to a process specified by *pid*
  - some signals can be sent from a terminal (e.g., ctrl-c)

- examples of standard signals
  - other signals are listed by `$ kill -l`
  - real-time (RT) signals behave a little different

number	name	default action	terminal key-combo
2	SIGINT	terminate a process	Ctrl-c
9	SIGKILL	terminate a process (unblockable)	
17	SIGCHLD	child stopped or terminated	
19	SIGSTOP	pause a process (unblockable)	Ctrl-z

Thank you for listening!

see you in the Q&A session

References

- Linux virtual memory
  - https://www.slideserve.com/cleave/virtual-memory
- Linux processes and signals
  - https://www.bogotobogo.com/Linux/linux_process_and_signals.php
- Linux man-pages
  - https://man7.org/linux/man-pages/man2/