Exceptional Control Flow

XD

Peking University

2021

Outline

Process

Exceptions

Concurrency

Private Address

Process Control

What can a process see?

- Continuous time
- A private memory space
- Exclusive use of the CPU

What can a process see?

- Continuous time
- A private memory space
- Exclusive use of the CPU

All the above are just illusions (or abstracts)!

Exceptions

Comprises Interrupts, Traps, Faults, and Aborts.

Interrupt

- Async: Not caused by any instruction
- Hardware triggered: I/O devices, disk controllers and timer chips
- Always returns to the next instruction

Traps and System Calls

- Intentionally triggered exceptions: ask for more access
- Sync: Caused by the current process
- Always returns to the next instruction

From *user mode* to *kernel mode*.

Faults

- Errors that could be correct: Page fault
- Sync
- Might return to the next instruction

Aborts

- Unsaveable Errors: Memory failed
- Sync
- Resulting in the termination of the current process

Remark

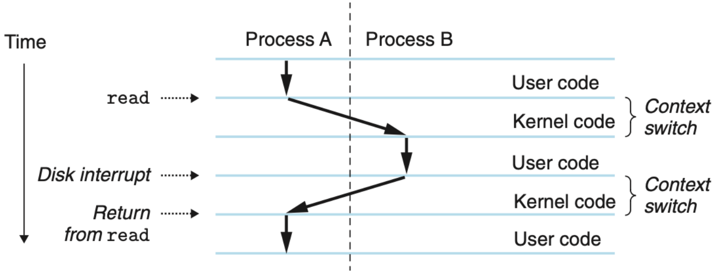
Division by zero could (potentially) be saved, but Linux opt to abort the process when happens.

Concurrency

Some of the processes might seem to be run concurrently.

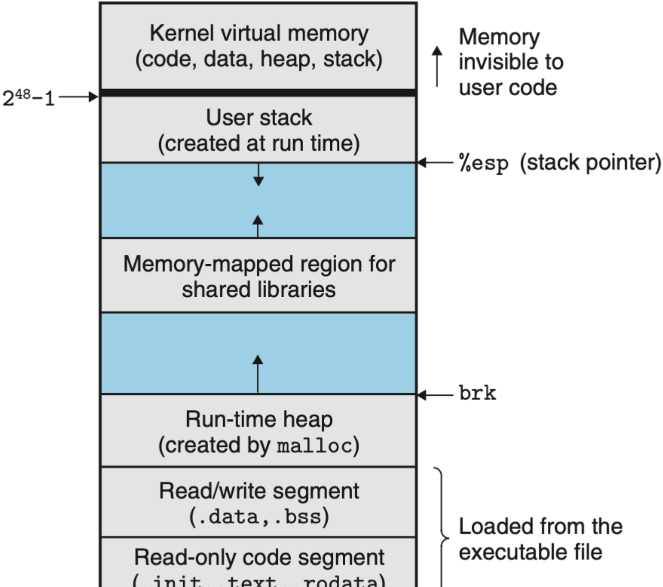
Context Switch

Context switching allows concurrency to be implemented. Here is an example.



Private Address Space

Also an abstract.



Kernel virtual memory
(code, data, heap, stack)

Memory
invisible to
user code

$2^{48}-1$

User stack
(created at run time)

%esp (stack pointer)

Memory-mapped region for
shared libraries

brk

Run-time heap
(created by malloc)

Read/write segment
(.data,.bss)

Loaded from the
executable file

Read-only code segment
(.init,.text,.rodata)

Process Control

Process ID(PID)

- getpid(void): return the current PID
- getppid(void): return the PID of the parent process

Process State

- Running: Running or going to be run
- Stopped: Suspended and won't be scheduled unless receiving SIGCONT
- Terminated: Stopped permanently

Remark

exit(int status) could be used to terminate the current process

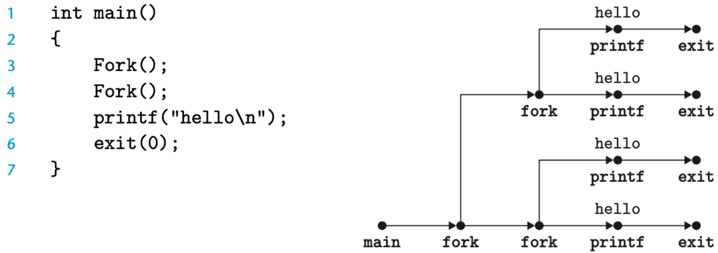
Fork

Fork: opens an *almost identical* child process.
About the child process:

- Start from the return of the fork (Call once, Return twice)
- Concurrent execution: no assumption should be made about the execution order
- Duplicate but separate address spaces
- Shared files: stdout

Fork() returns 0 for child and the child pid for parent.

Fork visualized



```
int main()
{
    Fork();
    Fork();
    printf("hello\n");
    exit(0);
}
```

Figure 8.17 **Process graph for a nested `fork`.**

Reaping Child Processes

Definition

Zombie Process: terminated yet not reaped

Any Orphaned Process would be reaped by init (PID 1, created during the system startup and terminated when shutting off).
To manually ensure a child is reaped, use

- waitpid(int pid, int *statusp, int options)
- wait(int *statusp)

pid

pid determines the wait set of the wait function.
pid ¿ 0: wait set is the child process of which the PID is pid
pid = -1: wait set is all the child processes

Option

Modify the default behavior of waitpid.

- 0: halt until a child in the wait set terminates
- WNOHANG: return immediately if none of the child processes in the wait set has terminated yet
- WUNTRACED: halt until a child in the wait set terminates or stops
- WCONTINUED: halt until a child in the wait set is terminated or is resumed from SIGCONT

Remark

Options could be combined, e.g., WNOHANG — WUNTRACED.

Quick Quiz #1

What's the equivalent in the form of waitpid(., ., .) for wait(&status)?

Quick Quiz #1

What’s the equivalent in the form of waitpid(., ., .) for wait(&status)?

waitpid(-1, &status, 0)

Check status

Defined in **wait.h**.

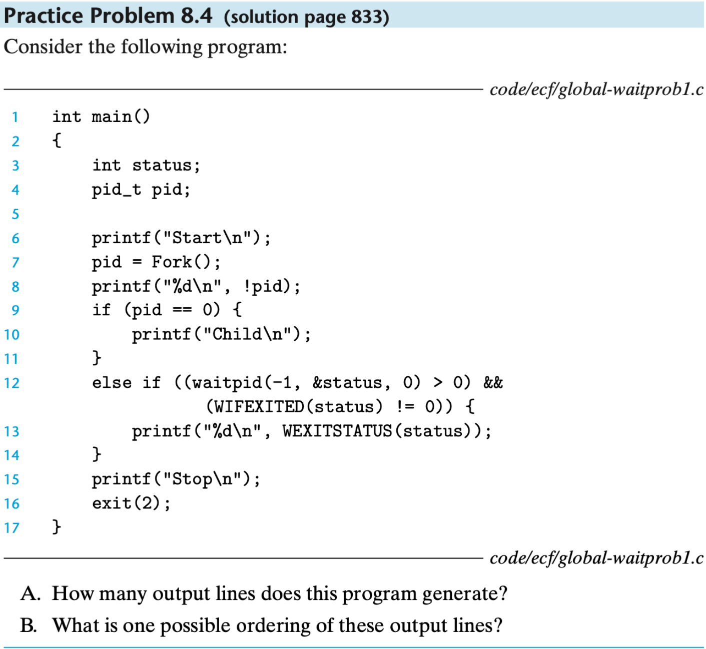
- WINEXITED: true if the child terminated normally
- WEXITSTATUS: the exit status of a normally terminated child; defined only if WINEXITED is true
- WIFSIGNALED: true if the child terminated due to an uncaught signal
- WTERMSIG: number of the signal causing the termination of the child; defined only if WINSIGNALED is true
- WIFSTOPPED: true if the child causing the return has stopped
- WSTOPSIG: number of the signal causing the child to stop; defined only if WIFSTOPPED is true
- WIFCONTINUED: true if the child restart on receipt of a SIGCONT

Error Condition

errno =

- ECHILD: if the process has no children
- EINTR: if waitpid is interrupted by another signal

Quick Quiz #2



Practice Problem 8.4 **(solution page 833)**

Consider the following program:

code/ecf/global-waitprob1.c

```
int main()
{
    int status;
    pid_t pid;

    printf("Start\n");
    pid = Fork();
    printf("%d\n", !pid);
    if (pid == 0) {
        printf("Child\n");
    }
    else if ((waitpid(-1, &status, 0) > 0) &&
                (WIFEXITED(status) != 0)) {
        printf("%d\n", WEXITSTATUS(status));
    }
    printf("Stop\n");
    exit(2);
}
```

code/ecf/global-waitprob1.c

A. How many output lines does this program generate?

B. What is one possible ordering of these output lines?

Putting processes to sleep

- sleep(uint s): sleep for s secs
- pause(): sleep until waken by a signal

Load and Run

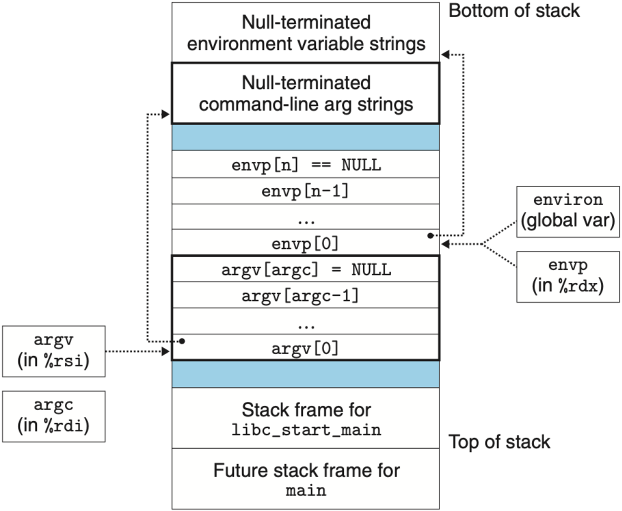
execve(const char* filename, const char* argv[], const char* envp[])

- filename:
- argv: arguments, terminated with NULL
- envp: environment, terminated with NULL

Example

argv[] = {"g++", "-o", "program", "-O2", "xxx.cpp", NULL}
envp[] = {"DEBUG=1", NULL}

Load and Run


Bottom of stack
Null-terminated environment variable strings
Null-terminated command-line arg strings
envp[n] == NULL
envp[n-1]
...
envp[0]
argv[argc] = NULL
argv[argc-1]
...
argv[0]
Stack frame for libc_start_main
Future stack frame for main
Top of stack
environ (global var)
envp (in %rdx)
argv (in %rsi)
argc (in %rdi)