Processes

Sebastian Altmeyer (Raphael ‘kena’ Poss)

April 5th, 2018

Introduction

What’s going on?

```
$> du -k | sort -n | grep src > disk-use
```

1. Shell reads and parses command line;
2. Shell start 3 new processes, and connects the inputs and outputs as indicated by the pipeline;
3. The new processes then execute “du”, “sort” and “grep”
4. The shell waits until the 3 processes complete.

The shell is ran in a process too!

What is a process?

A process is:

- An **abstraction** for the execution of a task

  This abstraction is used by people to talk about the activities in the computer.

  The properties of the process abstraction include its its **state** and its **process identifier (pid)**.

A process is also:

- The set of **concrete resources** used to execute the task, including:
  - Processor registers: program counter, etc
  - Data: variables and constants
  - Code: Instructions for the processor
  - OS structures: memory layout, buffers, file descriptors

Lifecycle of a process

Process states

Process states

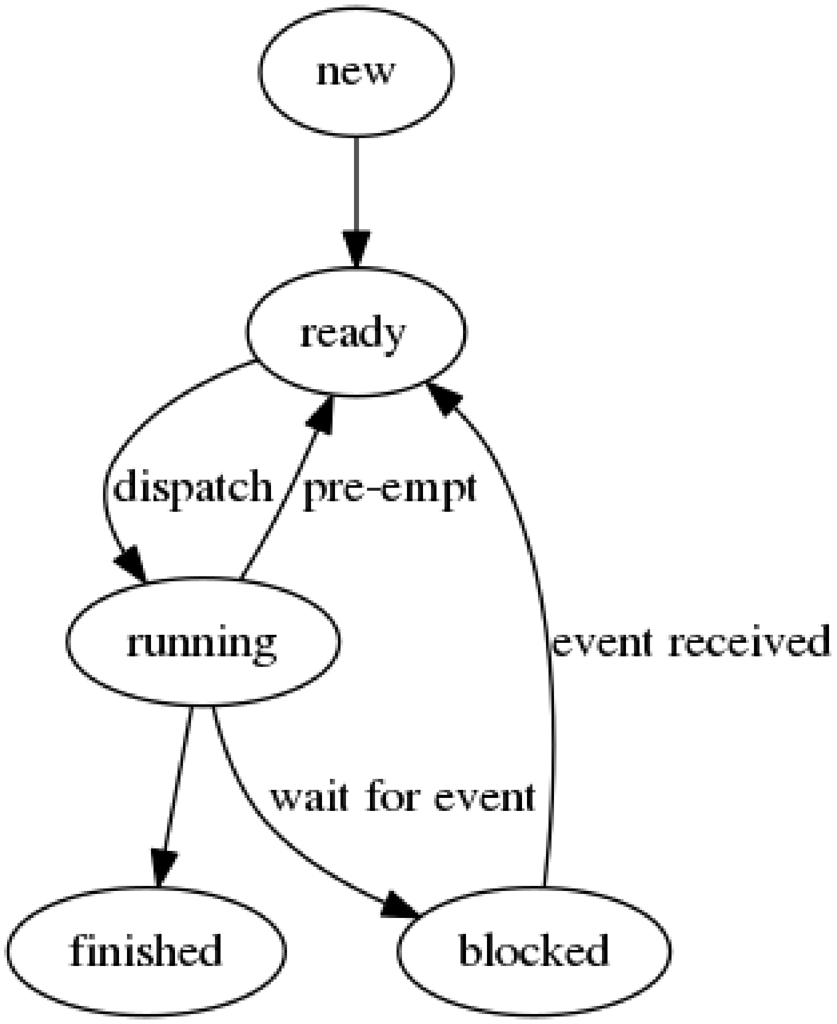

Figure 1:

Process states

- **New** : reserve the required resources, load the program
- **Ready/runnable** :
  - Proces is ready to use CPU time
  - Scheduler decides the *dispatch order*
- **Blocked / waiting** :
  - Waiting on an event (I/O, message, timer, . . .)
- **Running** :
  - Using the CPU, as long as it can and may
- **Finished / defunct**:
  - Cleaning up and releasing resources

The "context switch"

- Transition from one process to the next on the same processor/core
- Needed when there are more tasks than processors (multitasking)
- A process on the CPU has information stored in:
  - processor registers, incl program counter and stack pointer
    - Must be saved, so that they can be restored. "Save area"
  - MMU registers
    - Must be replaced with values for the new process
  - Cache
    - Content outdated. Usually cleaned up. Automatically repopulated by the new process.
- **A context switch is expensive!** (between 10ˆ3 and 10ˆ7 times more expensive than a simple addition)

The "process control block" (PCB)

The OS saves in the PCB:

- process identifier
- details about the owner, permissions, parent process, child processes
- content of CPU registers in a "save area" (when the process is not currently running)
- process state (see previous)
- pointers to tables for the MMU
- pointers to information about open files and communication channels
- used CPU time and other accounting information
- pointers to other PCBs (for process lists)

Context switch diagram

(Silberschatz, Figure 3.4)

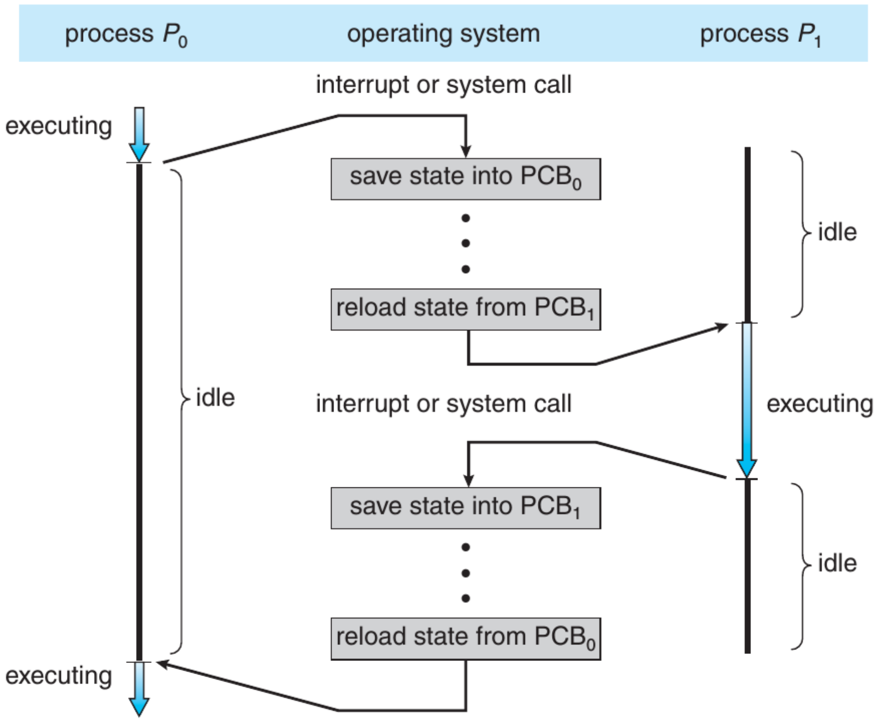
process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
idle
reload state from PCB_1
idle
interrupt or system call
executing
save state into PCB_1
idle
reload state from PCB_0
executing

Figure 2:

Steps of a context switch

1. Mode switch to system
2. Save all registers from old process in PCB
3. Update scheduling information
4. Move PCB to appropriate queue
5. Invalidate CPU cache
6. Choose next process
7. Program MMU with next process tables
8. Load registers from save area of next process
9. Mode switch to the next process

Creating a new process

Two approaches:

- “spawn” (e,g. Windows):
  1. initialise administrative data;
  2. search binary file of program;
  3. reserve memory and other resources;
  4. load code and data from file;
  5. activate proces.
- “fork” (e.g. Unix):
  1. duplicate an existing process and its resources;
  2. (optionally:) search binary file of program;
  3. (optionally:) *replace* code and data using file;
  4. activate process.

A new Unix process: “fork”

Simple definition: **duplicate** an existing process.

- called by an existing process;
- makes a copy of the “process image”, including all its resources;
- returns the value 0 to the child process, and the process ID of the child to the parent.

```
pid = fork();
if (pid) {
   /* pid != 0 - code for parent */
   if (pid < 0) {/* error */} else {/* success */}
} else {
   /* pid == 0 - code for child */
}
```

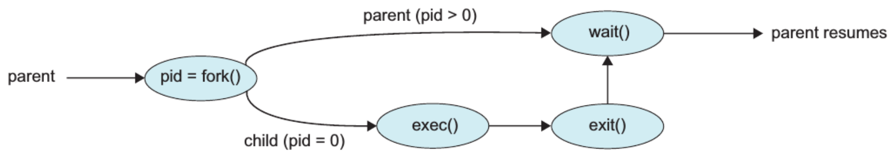

Figure 3:

(Silberschatz, Figure 3.10)

Executing another program: "exec"

Simple definition: **replace** the task of an existing process by another program.

- multiple related functions: `execve`, `execl`, `execle`, `execlp`, `execv`, `execvp`, `execvP`
- they all use at least 3 arguments:
  - path to binary file of program;
  - arguments for the new program's `main`;
  - "environment"
- `exec` often (but not always) follows a `fork`
- in your assignment: use `execvp`

Summary: fork + exec

- Fork
  - New process
  - Same program
- Exec
  - Same process (incl. all input and output channels!)
  - New program
- More information:

  See eg.: Doeppner 1.3.2 and following

  http://www.ibm.com/developerworks/aix/library/au-unixprocess.html

Typical shell structure

1. Read command line
2. Analyse comand line

3. for each command, if the command is “built-in”, run it within the shell; otherwise fork + exec and wait
4. return to step 1.

The Unix process tree

- First process: PID = 1: `init`
- `init` forks and execs startup programs
- each dead process must be waited upon by another process to clean up its resources:
  - parent processes often, but not always, wait on their children
  - `init` also waits on *orphan* processes

Fork + exec - is that expensive?

- Starting a new program from the shell:
  - First `fork` - make a copy of the shell process
  - Then usually replace it with `exec`
- Fork uses **copy-on-write** (COW)
  - Pages in memory are marked read-only
  - A copy is made only at the first attempt to write by one of the two processes,
  - The text (code) area is never copied

Example running processes

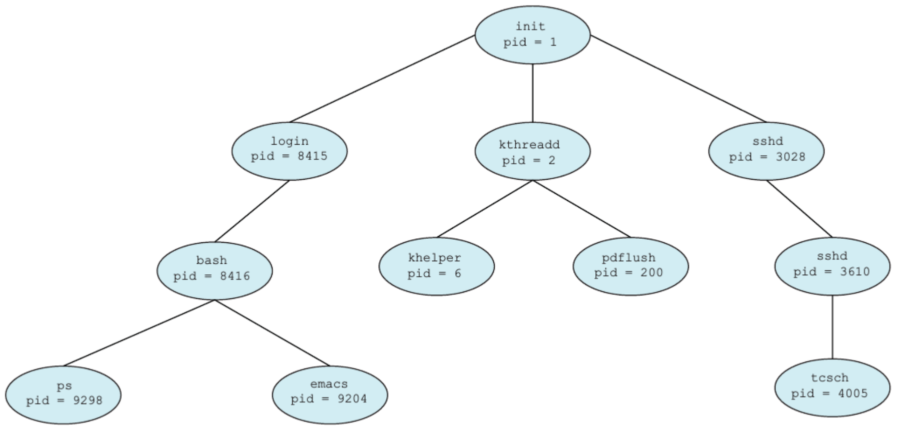

Figure 4:

(Silberschatz, Figure 3.8)

Communication between processes

Mechanisms

6 mechanisms:

- main mechanisms:
  - **arguments + environment** during `exec`
  - **file descriptors**: uniform interface for files, I/O devices, network, pipes
  - **signals**
- also available:
  - shared memory
  - message queues
  - semaphores

File descriptors

- Quite uniform interface for all sorts of I/O

  (Files, devices, network, pipes)

- The file descriptor itself is an integer, an *index* in a table of functions managed by the OS

  http://en.wikipedia.org/wiki/File_descriptor

- APIs:

  - Open/Connect: `open`, `socket`, `accept`, `pipe`, `socketpair`,...
  - Read/write: `read`, `write`, `recv`, `send`, ...
  - Configure/position: `lseek`, `fcntl`, `ioctl`, ...
  - Close/terminate: `close`

Unix `read/write`

- Works on **bytes**, not characters
- `ssize_t read(int fd, void *buf, size_t nbyte)`
  - returns the number of bytes actually read;
  - tries to read `nbyte` bytes, but may also return less for a socket, pipe or at EOF;
  - `O_NONBLOCK` also enables reading 0 bytes (see `fcntl`).
- `ssize_t write(int fd, void *buf, size_t nbyte)`
  - returns the number of bytes actually written;

    - tries to write `nbyte` bytes
    - `O_NONBLOCK` may change that
- see man pages read(2) en write(2)

What about `printf` / `scanf`?

- `printf` etc use "`FILE *`".
- `FILE` is really a `struct` containing a file descriptor
- when `printf` must write data, it calls `write` behind the scenes.
- `stdin` -> fd 0; `stdout` -> fd 1, `stderr` -> fd 2.
- this is managed by the *standard C library*, not the OS!

POSIX pipe

- Connection between parent and child process (usually)
- Required: two integer file descriptors: `int f_id[2];`
- Parent calls "`pipe(f_id)`" first
- Gets 2 file-ids back. The communication is uni-directional:
    - Read from `f_id[0]`,
    - Write to `f_id[1]`;
- Parent then calls "`fork()`".
    - Then parent writes to `f_id[1]`
    - Child reads from `f_id[0]`

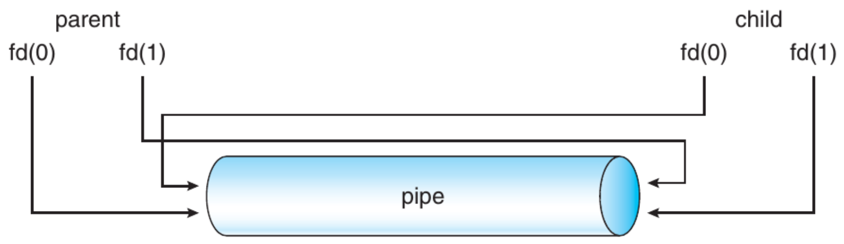

Figure 5:

(Silberschatz, Figure 3.24)

POSIX pipe: Example

```
int main(void)
{
    int     fd[2], nbytes;
    pid_t   childpid;
```

```
        char    string[] = "Hello, world!";
        char    readbuffer[80];

        pipe(fd);

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

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
    }
```

Renumbering file descriptors

- What happens with: `ls >out.txt`?
  - stdout is fd 1, shell opens `out.txt`, `open()` returns eg. fd 3
  - how do we ensure that ls gets `out.txt` as stdout?
- `newfd = dup(oldfd)`
  - After this, `oldfd` and `newfd` refer to the same file. the OS chooses a value for `newfd`.
- `val = dup2(oldfd, newfd)`
  - After this `oldfd` and `newfd` refer to the same file. The program chooses the value for `newfd`. `val` is equal to `newfd` unless an error occurs.

Example with pipes

```
$  bc | grep --line-buffered 4 | grep 2 --line-buffered
## 1. shell creates pipe: f_id[0] and f_id[1]
## 2. shell forks to child1
## 3. child1 connects stdout with f_id[1], closes f_id[0]
## 4. child1 starts bc
## 5. shell closes f_id[1]; creates pipe: f_id[2] and f_id[3]
## 6. shell forks child2
## 7. child2 connects stdin with f_id[0], stdout with f_id[3], closes f_id[2]
## 8. child2 starts grep
## 9. shell closes f_id[0], f_id[3]; forks child3
## 10. child3 connects stdin with f_id[2]
## 11. child3 starts grep
## 12. shell closes f_id[2], waits
```

Fork-pipe structure

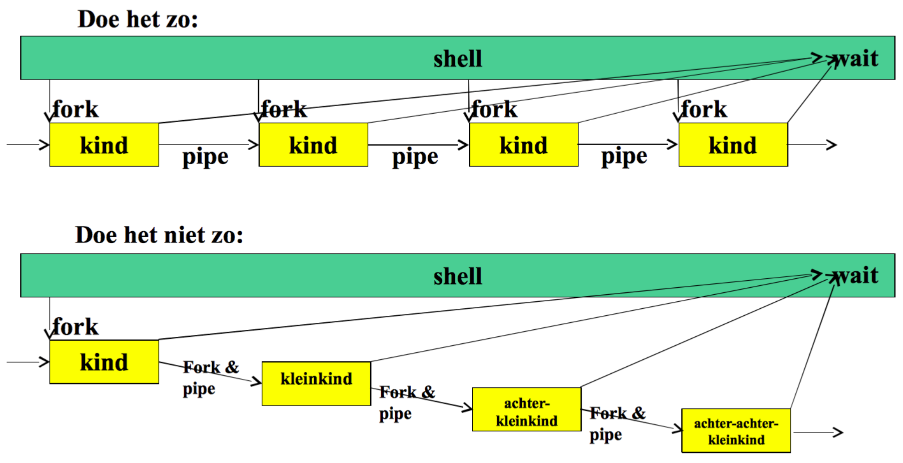

Figure 6:

POSIX Signals

- See also Doeppner 2.2.5 "Deviations"
- Asynchronous signal from OS to process
  - Resembles a hardware interrrupt
  - Can be ignored, blocked or handled

    - Default behavior or configurable signal handler
- Usually the default handler stops the process
  (but not always)

Signal handling

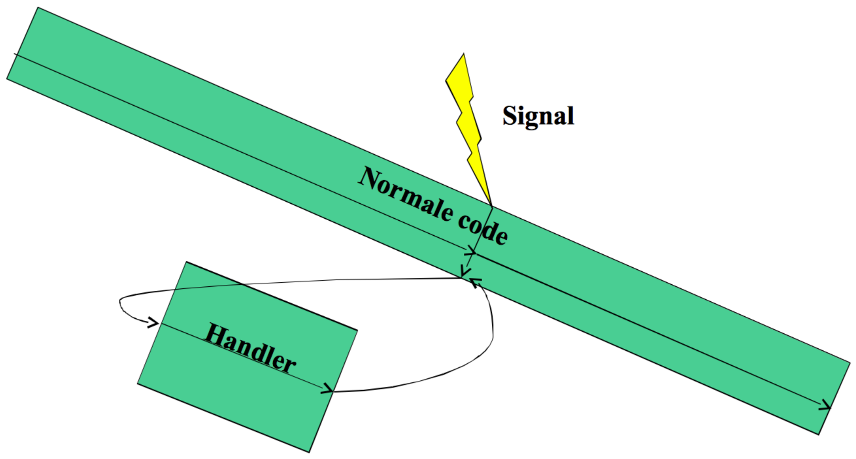

Figure 7:

Example signals

`SIGINT` (default: stop process) Request to terminate the program (Ctrl+C)

`SIGSTOP` (default: suspend process) Request to suspend the program (Ctrl+Z)

`SIGHUP` (default: stop process) Program is trying to use a closed fd

`SIGFPE` (default: stop process) Program performed an invalid arithmetic operation

`SIGBUS`, `SIGSEGV` (default: stop process) Program performed an invalid memory operation

`SIGCHLD` (default: ignored) Something happened to a child process

Example code

```
void finale(int signum) {
   breaking++;
   if (breaking > 2) {
     write(2, "Stopping program\n", 17);
     _Exit(0);
   }
   write(2, "Program terminating\n", 20);
}

/* Code in main() */
sigaction(SIGINT, 0, &newact);
newact.sa_handler = &finale;
newact.sa_flags |= SA_RESTART;
sigaction(SIGINT, &newact, 0);
```

Process management

- Proces management is organized using **signals**.
- Signals are sent:
  - to a process that tries to read from a closed fd;
  - by specific key combos to all processes connected to that keyboard;
  - from one process to another via `kill()`;
  - to a process encountering a fault.

Threads

What are threads?

- "light-weight processes"
- originally: one thread per process
- now: one process, several threads
- Advantages
  - responsivness
  - resource sharing
  - economical (thread creation cheaper than process creation)
  - scalability (exploit multiprocessor systems)
- various different thread-libraries (java threads, pthreads, windows threads)

Threads versus Processes

- Processes

    - own process identifier
    - own memory space, file descriptors, and signals
- Threads
    - same process identifier of creating process, own thread identifier (tid)
    - share memory, file descriptor and signal handling
- be careful with *fork*: do we also copy the threads?

Further reading

- Doeppner, Chapter 1.3.2 - 1.3.5
- Silberschatz, Chapter 3.1 - 3.4