Processes Management

Sistemas de Computadores

2017/2018

Luís Miguel Pinho / Luis Ferreira / Luís Nogueira / Nuno Pereira

Processes

- Instance of execution of a program

- **Program != Process**
  - Program: code + static data
  - Process: dynamic instantiation of code + data + state

- Program and process
  - Process > program: more than code and data
  - Program > process: one program can create several processes

Process

- From the point of view of the operating system, a process is the entity to which resources are given
  - CPU time, memory, files, I/O devices, etc.
- Processes are isolated from each other
  - Sequential execution has well defined interactions
  - By default, no memory sharing between processes

OS process management

- Identify and represent processes internally
- Create/remove processes of user and of system
- Schedule processes access to the CPU
- Provide communication and synchronization mechanisms

Processes identification

- A process is identified through um unique value, names Process ID (PID)

- PIDs are usually given by OS sequentially until the limit of the data type
  - In that case, PID numbers are recycled from older processes that have already finished (lower values first)

Process creation

- A process is created …
  - By action of the OS
  - By action of the user
  - By another process

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork()
```

Process hierarchy

- The process that invoques the fork system call is named the *parent*
- The newly created process is the *child*
- Each one can create more processes, leading to a process tree
    - Each process has exactly one parent and zero or more children

Process hierarchy

```
$ pstree | more
init-+-acpid
|-avahi-daemon---avahi-daemon
|-bonobo-activati---{bonobo-activati}
|-cron
|-cupsd
|-gdm---gdm-+-Xorg
|          `-x-session-manag-+-gnome-panel---{gnome-panel}
|                            |-gnome-settings--+-pulseaudio-+-gconf-helper
|                            |                 |            `-2*[{pulseaudio}]
|                            |                 `-{gnome-settings-}
|                            |-konsole---3*[bash]
|                            |-metacity
|                            |-ssh-agent
|                            `-{x-session-manag}
|-getty
|-konsole-+-2*[bash]
|         |-bash---vim
```

getpid and *getppid*

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid();
pid_t getppid();
```

- *getpid returns the* PID of the process
- *getppid returns the* PID of the parent process

Process creation

- The Child process is a copy of the parent
  - The only differences are the PID (remember it is unique), and certain resources (e.g. signals are not inherited) and stats
  - Child and parent are independent processes and both compete for the CPU time

- After *fork*, both processes execute the same code, in the instruction immediately after the fork function call
  - Order of execution is not known
  - There needs to be a way to distinguish them

fork return values

- -1, in case of error
  - It was not possible to create the child, so a single process still exists
- **The PID of the child** (> 0) is returned in the parent process
  - Parent may need this PID to communicate or synchronize with the child
- The child process receives from fork the value **zero**
  - The fork was not executed by the child
- The different values received by parent and child allows to separate the code each one will execute

Example – fork returned values

```
pid = fork();

if(pid == -1){
   perror(“Fork failed”); exit(1);
}

if(pid > 0)
   printf(“Luke, I’m your father\n”);
else
   printf(“I am the son\n”);

printf(“Who executes this line?”);
```

Process creation

- We can think of *fork* as a cloning function
  - It creates a copy of the original process, with the same code and data
- In the child process, the new variables have the same value as the original ones had in the parent in the instant of execution of the fork
- However, any change made in any of the processes after the fork, is independent and does not affect the other process

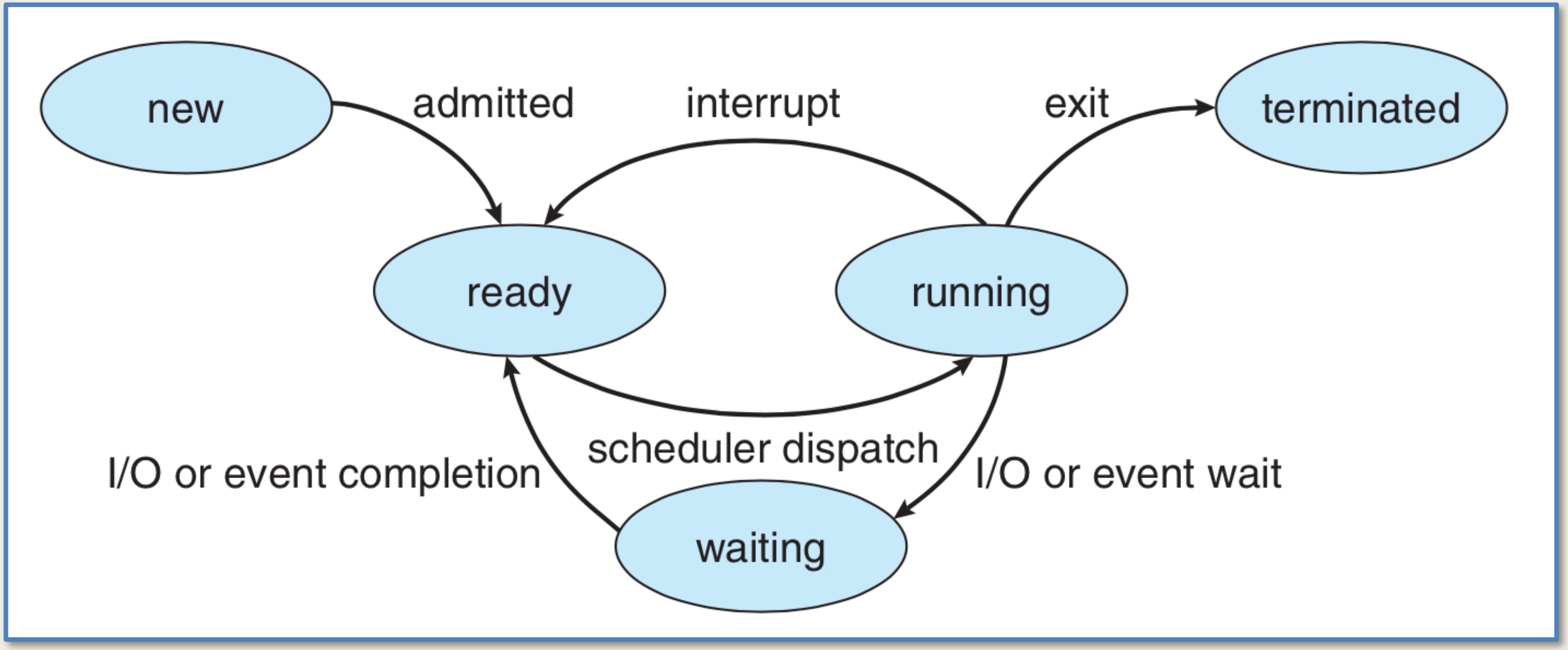
Example – changes after *fork*

```
int x = 10;

pid = fork();

if(pid > 0){
  printf(“Luke, I’m your father\n”);
}
else{
  printf(“I am the son\n”);
  x++;
}
printf(“x = %d\n”, x);
```

Concurrent execution

- Only one process may, at a given instant, be executing in the core of a CPU…
- … but multiple processes can be loaded to memory and be executed concurrently
- The goal of the OS is to have always a process executing, maximizing resource usage
  - Decision based in metrics of performance and load (scheduling)

States of a process



sleep

```
#include <unistd.h>

int sleep(int seconds);
```

- Suspends execution of the process during (at least) *n* seconds
  - If a process receives a signal while it is sleeping, it may waken before time

- Sleep can be used to try to obtain a order between processes
  - But it is not guaranteed, so use it only for testing, not to guarantee order and synchronization!

Process termination

- A process terminates when …
  - Executes the last instruction
  - Invokes the *exit* system call
  - voluntarily in an error
  - Involuntarily in an error
  - By action of a user or another process

Process termination

```
#include <stdlib.h>

void exit(int status);
```

- Terminates the process, providing a status value to the operating system
  - A single byte of *status* is used so values are 0 - 255

- Two contants are used, EXIT_SUCCESS and EXIT_FAILURE, to indicate success or failure in the execution

Process termination

- When a process terminates, the parent is informed by the OS using the SIGCHLD signal
- This event is asynchronous – the parent can receive it at any time
  - The parent can ignore the signal (the default behavior) or provide a function to be executed when the signal is received (a handler)

Zombie Processes

- If the child terminates before the parent
  - It becomes a zombie – cannot execute but the OS still has to keep the information of its termination

- OS cannot remove the process from the process table, but can free all process resources
  - Parent has to acknowledge the termination for the entry to be removed
  - Acknowledgement is with *wait/waitpid* functions
  - After acknowledgement, process disappears and is no longer zombie

Orphan Processes

- If the parent terminates before the child
  - Child become *orphan*

- In that case, it is necessary the process to be adopted by another process to maintain hierarchy

- In UNIX, orphan processes are adopted by the *init* process that periodicall invokes *wait* to all its children

wait and *waitpid*

```
#include <sys/wait.h>

pid_t wait(int *status);
```

- *wait* **blocks until any of** the process children terminates
  - Returns the PID of the terminated child, or -1 in case of error
- *status* should be initialized with the address of a variable that will receive the exit value of the child
  - The exit value also includes some extra information by the OS

wait and *waitpid*

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

- *waitpid* **blocks until the specified child** terminates
  - *pid* = -1 means it will wait for any children, as *wait*

- *options* may be 0 (blocks), WNOHANG (does not block) and WUNTRACED (children in the STOPPED state)

wait and *waitpid*

- When invoking *wait* or *waitpid*, the process may:
  - Be blocked until the child terminates
  - Return immediately if the child has already terminated
  - Return immediately with an error, if there are no children

Example – synchronizing with *wait*

```
pid = fork();

if(pid > 0){
   printf(“Luke, I’m your father\n”);
   wait(NULL);
   printf(“The child has terminated”);
}
else{
   printf(“I’m the son\n”);
   Some_code();
   exit(0);
}
```

Process exit information

- The OS also stores some information related to the way a process terminates:
    - The exit value
    - The signal that caused the abnormal termination (if any)
    - If the process originated a *core dump*
    - etc

- There are several macros that can be used to obtain that information from the status obtained in *wait/waitpid*

Process exit information

- WIFEXITED(status)
  - Checks if the child terminated normally
- WEXITSTATUS(status)
  - Exit value (1 byte) given in the *exit* function
- WIFSIGNALED(status)
  - Checks if the child terminated due to a non-treated signal

Process exit information

- WTERMSIG(status)
  - Gets the number of the signal that terminated the child
- WCOREDUMP(status)
  - Checks if a *core dump* was generated
- WIFSTOPPED(status)
  - Checks if the child is in the STOPPED state
- WIFCONTINUED(status)
  - Checks if the process was continued

Example – process exit information

```
  pid_t pid1, pid2;
  int status;

  pid1 = fork();
  if(pid1 > 0){
     pid2 = waitpid(pid1,&status,0);
     if(WIFEXITED(status))
        printf(“Parent: child %d returned
%d\n”, pid2, WEXITSTATUS(status));
  }
  else
     exit(10);
  }
```

Exercise

- Implement a program that creates 2 new proceses, with the following behaviour
  - Child 1 sleeps 5 seconds, prints its PID and terminates wirth value 1
  - Child 2 prints its PID and the parent PID and terminates with value 2
- Independently of the order of execution and termination, parent waits first for child 1 and then child 2, printing their PIDs and exit values
  - But guarantee that both children execute concurrently