Processes

Advanced Control (exec)

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

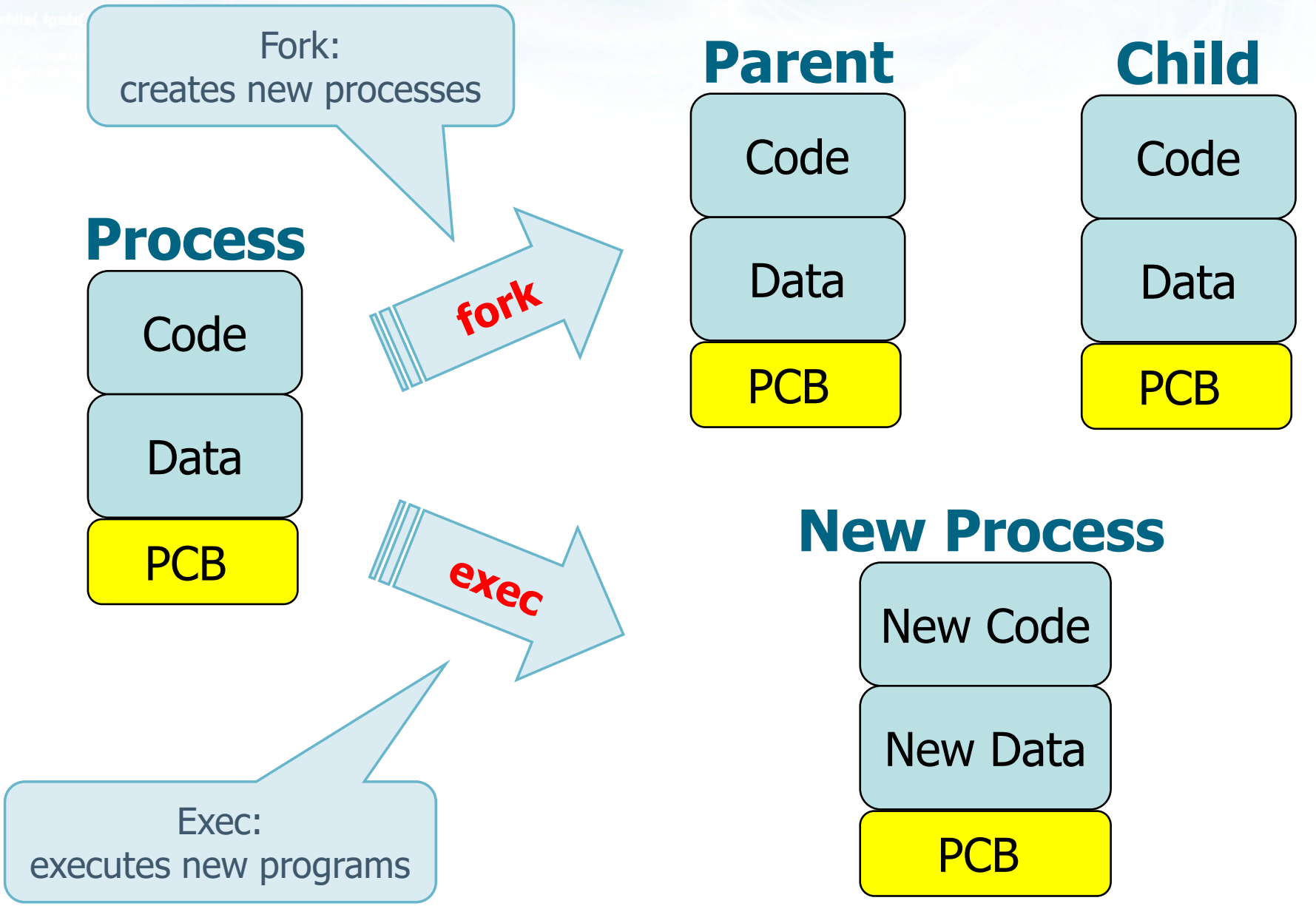
fork and exec system calls

- System call **fork** creates a new process duplicating the calling process.
- There are two main applications of this mechanism
  - Parent and child execute **different code sections**
    - Example: a network server duplicates itself at each client request, and the child serves the request while the parent waits for a new client request
  - Parent and child execute **different code**
    - Example: a command interpreter (shell)
    - Uses the family of **exec** system calls
      - This function is used by many others system call

exec system call

- System call **exec substitutes** the process code with the executable code of another program
- The new program begins its execution as usual (from main)
- In particular **exec**
  - Does not create a new process
  - Substitutes the calling process image (i.e., its code, its data, the stack and the heap) with the image of another program.
  - The process PID does not change
    - fork → duplicates an **existent process**
    - exec → executes a **new program**

Address space



exec system call

- **6** versions of exec system call
  - execl, execlp, execle
  - execv, execvp, execve

Type	Action
l (list)	Arguments are a list of strings
v (vector)	Arguments is a vector of strings arguments (char **)
p (path)	The executable filename is looked for in the directories listed in the environment variable PATH
e (environment)	The last argument is an environment vector envp[] which defines a set of new associations strings name=value

exec system call

```
#include <unistd.h>

int execl (char *path, char *arg0, ...,    (char *)0);
int execlp (char *name, char *arg0, ...,   (char *)0);
int execle(char *path, char *arg0,..., (char *)0,
           char *envp[]);
int execv (char *path, char *argv[]);
int execvp (char *name, char *arg[]);
int execve (char *path, char *arg[], char *envp[]);
```

- Returned values
  - **None** on success
  - -1 on error

exec system call

- Arguments
  - Pathname of the executable file
    - Pathname can specify the name of a file, or the name of a file with the related path
    - In the "p" versions of the exec it is sufficient (and better) to specify only the name of the file
      - If the pathname does not contain a path, it is inherited by the environment variable PATH (echo $PATH)
      - If the pathname contains a path, the "p" version of exec is equal to the non-"p" version
  - In the non-"p" version the pathname should include the path (otherwise unknown)

exec system call

- Its argument list
  - In the "l" versions, exec receives a list of parameters (like a main in C)
    - The first argument is the **name** of the process
      - In practice the string argv[0] of the C syntax
    - The other arguments of the list are the arguments for the executable
      - In practice argv[i] with i>0 of the C syntax (i.e., argv[1], argv[2], etc)
  - In the "v" versions the argument is a vector of pointers to the arguments
    - In practice it is a dynamic matrix similar to ** argv
    - Similar, not identical, because it is "NULL terminated"
      - The value argv[i]==NULL indicates the end of the arguments

exec system call

- The optional environment variables
  - In the non- "e" versions, environment variables are inherited from the calling process
  - In the versions "e", environment variables are explicitly specified
    - A second matrix dynamically allocated and NULL-terminated is passed to the function, which is a vector of pointers to strings of characters
    - These strings specify the values of the desired environment variables (e.g., variable=value)

Examples

OK

whereis cp: /bin/cp

User defined name

```
execl("/bin/cp","mycp","./file1","./file2",NULL);
```

OK

Alternative termination

```
execl("/bin/cp","mycp","./file1","./file2",(char*)0);
```

NO

Path is missing

```
execl("cp","File_copy","./file1","./file2",(char*)0);
```

OK

Default path ($PATH)

```
execlp("cp","mycp","./file1","./file2",(char*)0);
```

Example

The program (./**pgrm**) recalls itself if it receives as parameter 1 or 2

```
...
n = atoi (argv[1]);
switch (n) {
  case 1:
    printf("#1:PID=%d;PPID=%d\n", getpid(), getppid());
    sleep (n*10);
    execlp ("./pgrm", "./Pgrm", "2", (char *) 0);
    break;
  case 2:
    printf("#2:PID=%d;PPID=%d\n", getpid(), getppid());
    sleep (n*10);
    execlp ("./pgrm", "myPgrm", "3", (char *) 0);
    break;
  default:
    printf("#3:PID=%d;PPID=%d\n", getpid(), getppid());
    sleep (n*10);
    break;
}
return (1);
```

The path is the same arg0 (its name) changes

Example

Run with n=1

The PID does not change

Shell commands (in blue)

```
> ./pgrm 1 &
[2] 2471
#1: PID=2471; PPID=2045

> ps -aux | grep 2471
scanzio 2471 0.0 0.0 4192 352 pts/2 S 19:29 0:00 ./pgrm 1
#2: PID=2471; PPID=2045

> ps -aux | grep 2471
scanzio 2471 0.0 0.0 4192 356 pts/2 S 19:29 0:00 ./Pgrm 2
#3: PID=2471; PPID=2045

> ps -aux | grep 2471
scanzio 2471 0.0 0.0 4192 356 pts/2 S 19:29 0:00 ./Pgrm 3
[2]+ Exit 1 ./pgrm 1
```

The name changes

exec system call

- exec**v**[p]
  - Uses a single argument: a pointer
    - The pointer identifies a vector of pointers to the parameters (i.e., strings)
    - The vector must be properly initialized

```
char *cmd[] = {
   "ls",
   "-laR",
   ".",
   (char *) 0
};
...
execv ("/bin/ls", cmd);
```

Last argument must be the NULL pointer

System call exec ()

- exec[lv]**e**
  - Can provide to the executable a set of environment variables
    - Pointer to a vector of pointers (i.e., strings)
    - Without "e" the environment of the new process is inherited from the calling process

```
char *env[] = {
   "USER=unknown",
   "PATH=/tmp",
   NULL
};
...
execle (path, arg0, ..., argn, 0, env);
...
execve (path, argv, env);
```

Considerations

- Note that during the exec
  - all open file descriptors are mantained (including stdin, stdout, stderr)
  - This allow the process to inherit possible redirections previously set (e.g., by shell)
- Many kernels
  - Implement only system call **execve**
  - The other versions are macros that use this system call

Exercise

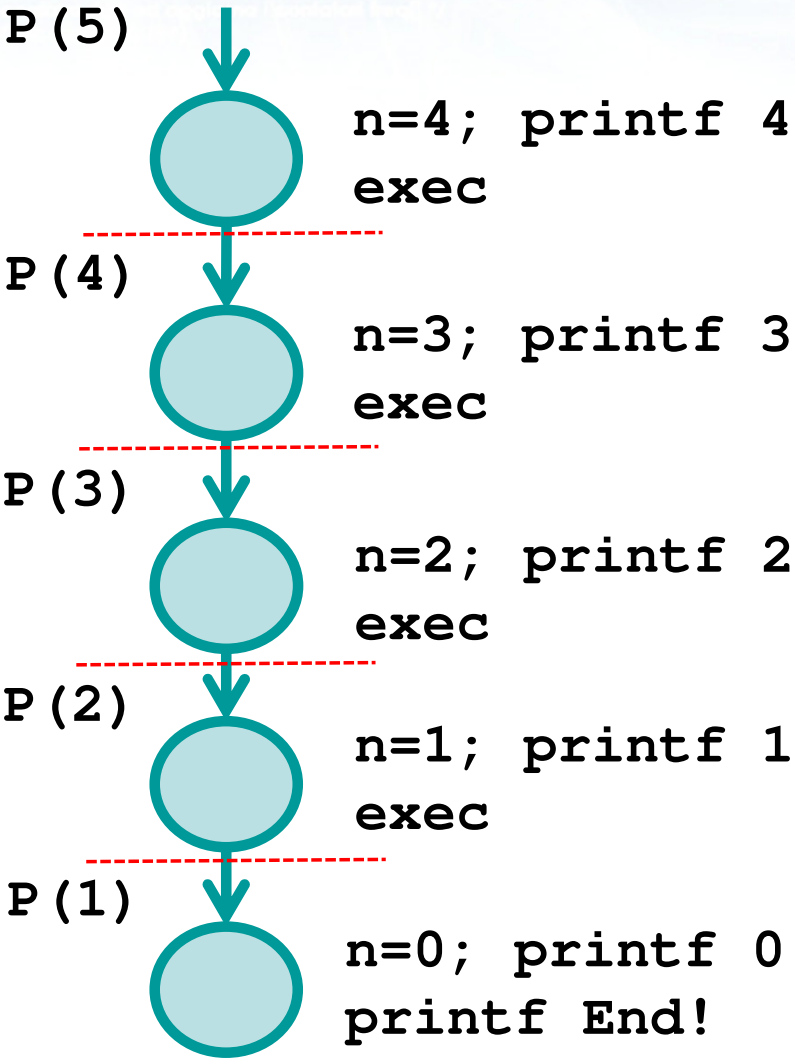
- Draw the process generation tree of the following C code segment
  - executed passing as its argument on the command line string "5"
- What does it display?
- Why?

Exercise

Run with n=5

```
#include <stdio.h>
...
#include <unistd.h>
int main (int argc, char ** argv) {
  char str[10];
  int n;
  n = atoi(argv[1]) - 1;
  printf ("%d\n", n);
  if (n>0) {
    sprintf (str, "%d", n);
    execl (argv[0], argv[0], str, NULL);
  }
  printf ("End!\n");
  return 1;
}
```

Solution




```
int main (int argc, char ** argv) {
  char str[10];
  int n;
  n = atoi(argv[1]) - 1;
  printf ("%d\n", n);
  if (n>0) {
    sprintf (str, "%d", n);
    execl (argv[0], argv[0], str, NULL);
  }
  printf ("End!\n");
  return 1;
}
```

Output

```
4
3
2
1
0
End!
```

Exercise

- Draw the process generation tree of the following C code segment
- What does it display?
- Why?

Exercise

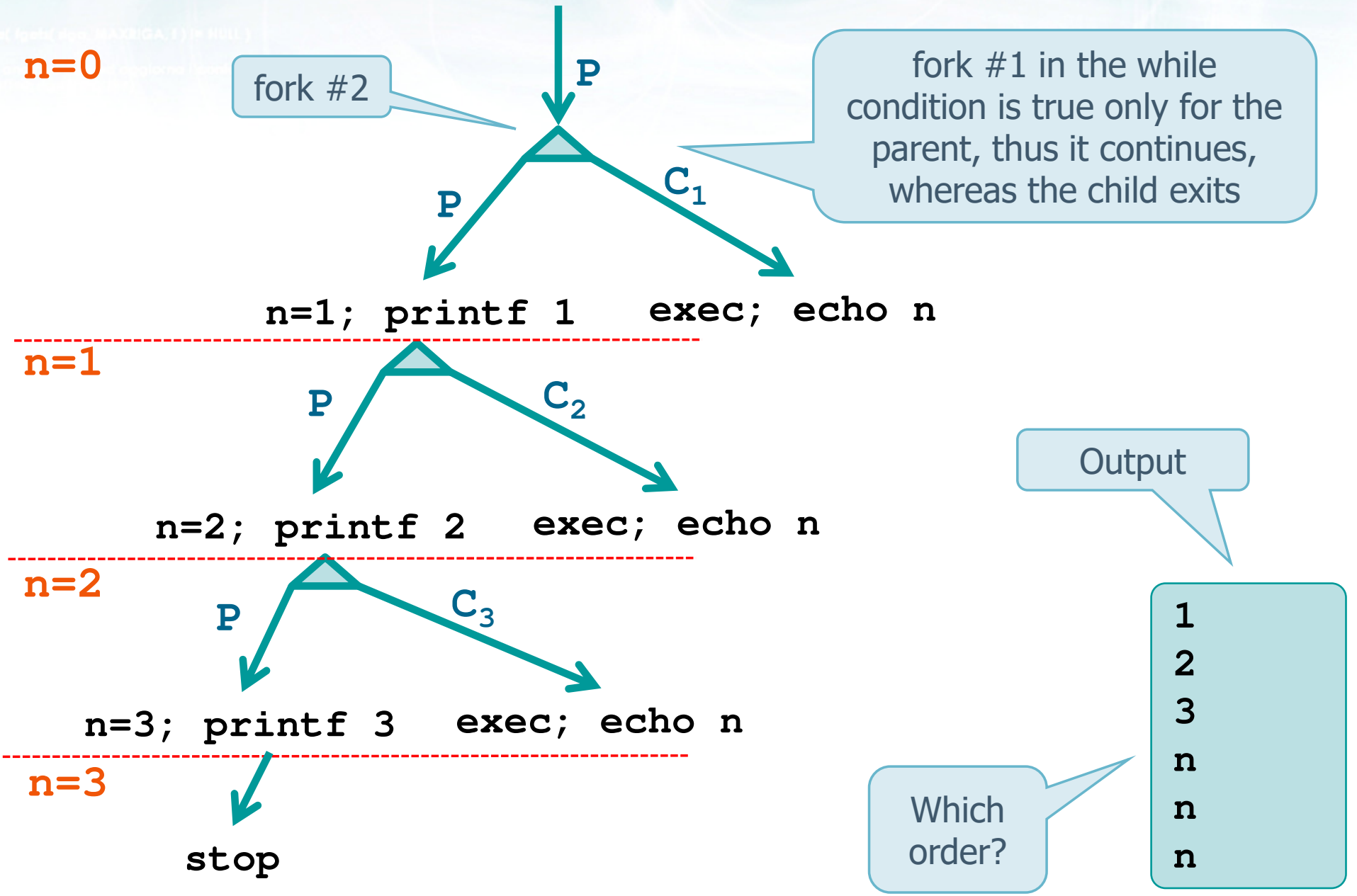
```
#include <stdio.h>
#include <unistd.h>

int main(){
  int n;
  n=0;
  while (n<3 && fork()){
    if (!fork())
      execlp ("echo", "n++", "n", NULL);
    n++;
    printf ("%d\n", n);
  }
  return (1);
}
```

fork #1
If 0 we are in the child; the child ends immediately

fork #2
If 0 we are in the child; the child does exec

shell command to print on stdout

Solution


n=0
fork #2
P
P
C_1
fork #1 in the while condition is true only for the parent, thus it continues, whereas the child exits
n=1; printf 1 exec; echo n
n=1
P
C_2
n=2; printf 2 exec; echo n
n=2
P
C_3
n=3; printf 3 exec; echo n
n=3
stop
Output
1
2
3
n
n
n
Which order?

UNIX shell skeleton

- Command run in foreground
  - <command>

```
while (TRUE) {
  write_prompt();
  read_command (command, parameters);
  if (fork() == 0)
    /* Child: Execute command */
    execve (command, parameters);
  else
    /* Parent: Wait child */
    wait (&status);
}
```

UNIX shell skeleton

- Command run in background
  - <command> **&**

```
while (TRUE) {
  write_prompt();
  read_command (command, parameters);
  if (fork() == 0)
    /* Child: Execute command */
    execve (command, parameters);
/* else */
    /* Parent: DOES NOT wait */
    /* wait (&status); */
}
```

Command execution

- It can be useful to execute a **shell command** from a process
  - For example for appending a date or a hour to a filename or to a file
- System call **system** solves this problem
  - Defined in the standard ISO C and POSIX
    - Although defined by the C standard, it is highly implementation-dependent
    - It is always present in UNIX-like systems

system() system call

```
#include <stdlib.h>

int system (const char *string);
```

Since it is implemented with fork, exec and wait has different termination conditions

- System call system()
  - Forks a shell, which execute the string command, while the parent process waits the termination of the shell command
  - Returned values
    - -1 if fork or waitpid fail (used in its implementation)
    - 127 if the exec fails (used in its implementation)
    - The exit value of the shell that executed the command (with the format of **`waitpid`**)

Example

```
...
system ("date");
...
system ("date > file");
```

Redirection...
see section u04s07

```
...
system ("ls -laR");
...
```

```
char str[L];
...
strcpy (str, "ls -la");
system (str);
...
```

system() implementation

- In initial LINUX versions
  - system was implemented by means of
    - fork, exec and wait
  - They were inefficient
    - while ((lastpid=wait(&status)) != pid && lastpid!=-1);
- Current versions
  - usually use the system calls fork, exec and waitpid

system() implementation

```
int system (const char *cmd) {
  pid_t pid;
  int status;
  if (cmd == NULL)
    return(1);
  if ( (pid = fork()) < 0) {
    status = -1;
  } else if (pid == 0) {
    execl("/bin/sh", "sh", "-c", cmd, (char *) 0);
    _exit(127);
  } else {
    while (waitpid (pid, &status, 0) < 0)
      if (errno != EINTR) {
        status = -1;
        break;
      }
  }
  return(status);
}
```

Error in fork

The shell must read from the command line, not from stdin

Interrupted function call

Options: WNOHANG

Exercise

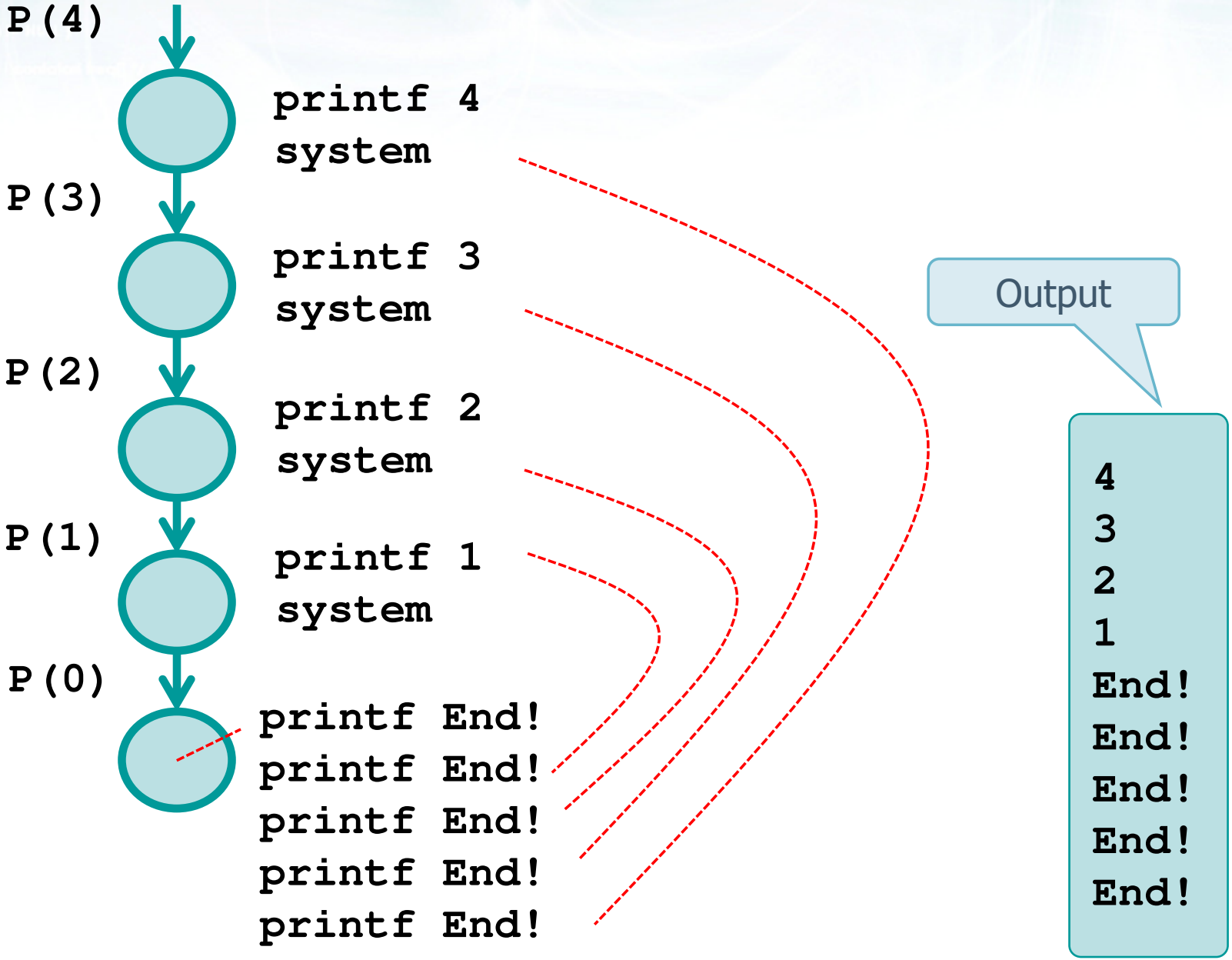
- Draw the process generation tree of the following C program
  - executed passing as its argument on the command line string "4"
- What does it display?
- Why?

Exercise

Run with n=4

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char ** argv){
  int n;
  char str[10];
  n = atoi (argv[1]);
  if (n>0) {
    printf ("%d\n", n);
    sprintf (str, "%s %d", argv[0], n-1);
    system (str);
  }
  printf("End!\n");
  return (1);
}
```

Solution


P(4)
P(3)
P(2)
P(1)
P(0)
printf 4
system
printf 3
system
printf 2
system
printf 1
system
printf End!
printf End!
printf End!
printf End!
printf End!
Output
4
3
2
1
End!
End!
End!
End!
End!

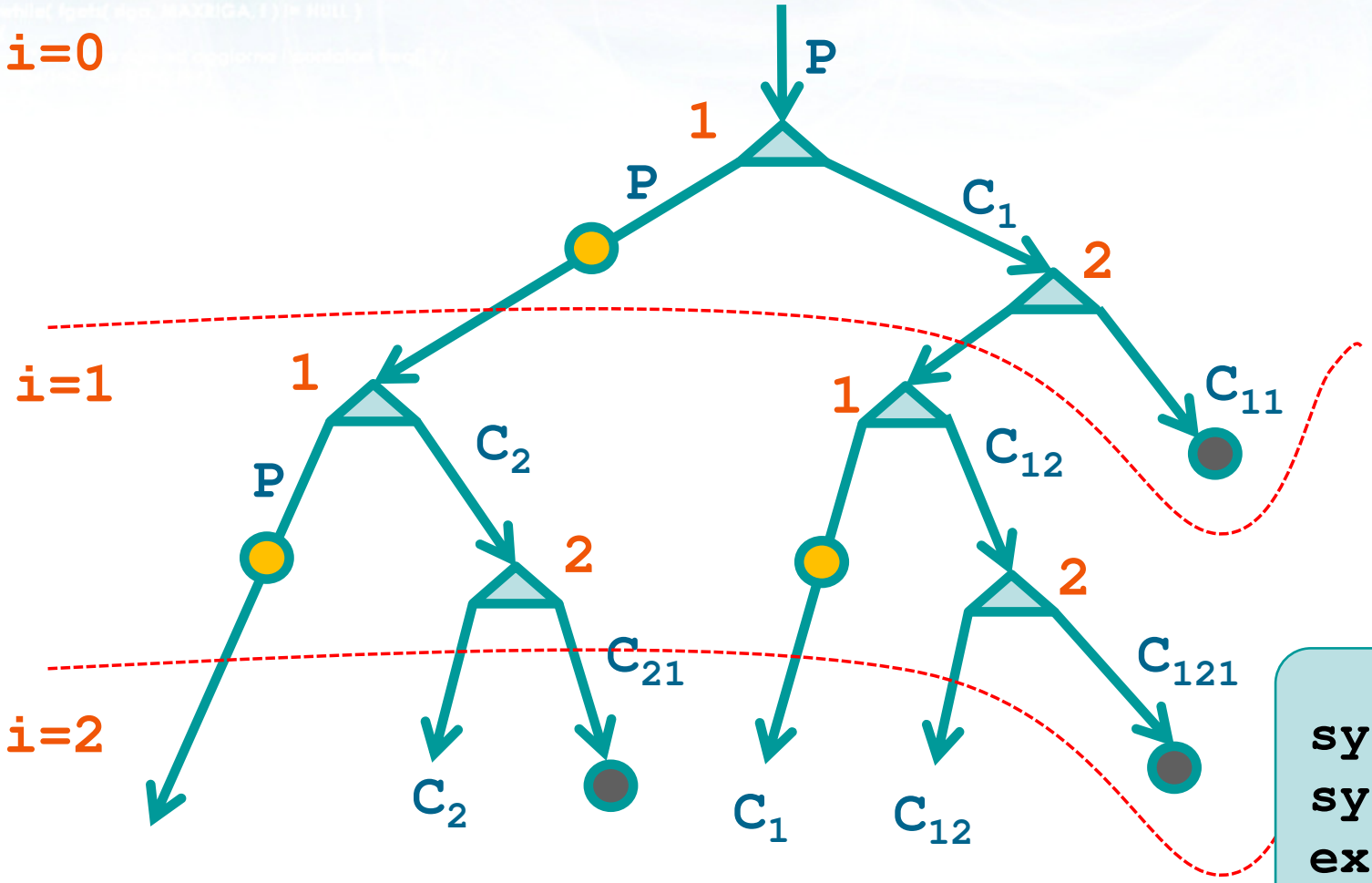
Exercise

- Draw the process generation tree of the following C code segment
- What does it display?
- Why?

Exercise

```
#include ...
int main () {
  char str[100];
  int i;
  for (i=0; i<2; i++){
    if (fork()!=0) {
      sprintf (str, "echo system with i=%d", i);
      system (str);
    } else {
      if (fork()==0) {
        sprintf (str, "exec with i=%d", i);
        execlp ("echo", "myPgrm", str, NULL);
      }
    }
  }
  return (0);
}
```

Exercise



echo system with i=%d

exec echo with i=%d

Which order?

Output

```
system with i=0
system with i=1
exec with i=1
exec with i=0
system with i=1
exec with i=1
```