System and Device Programming

UNIX Processes

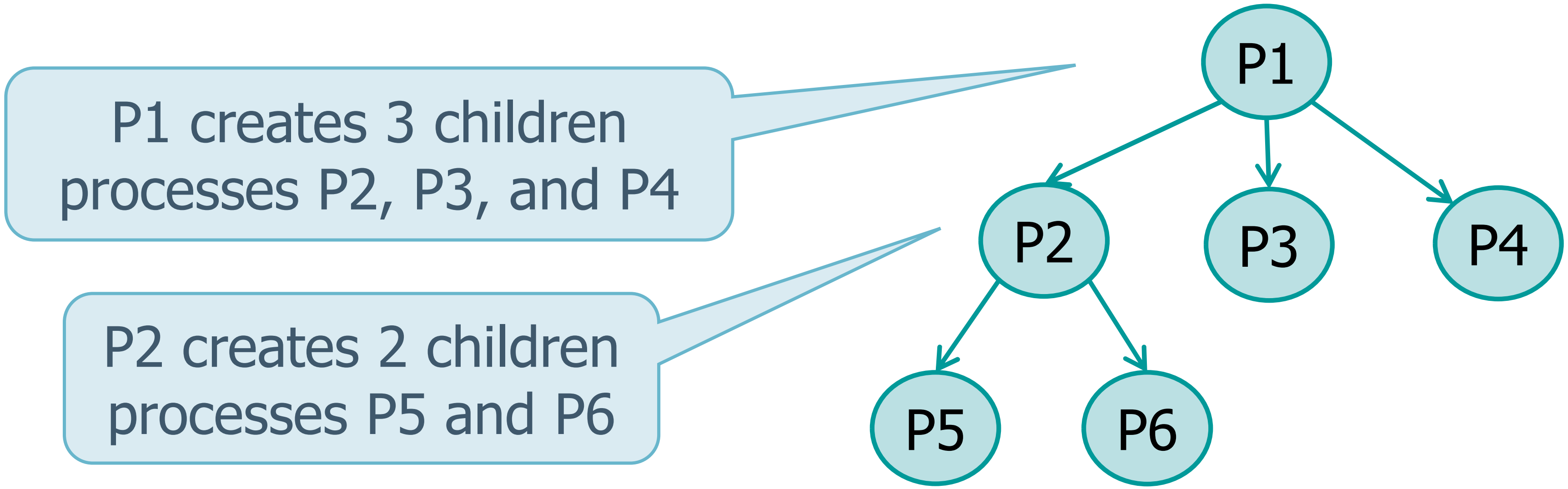
Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

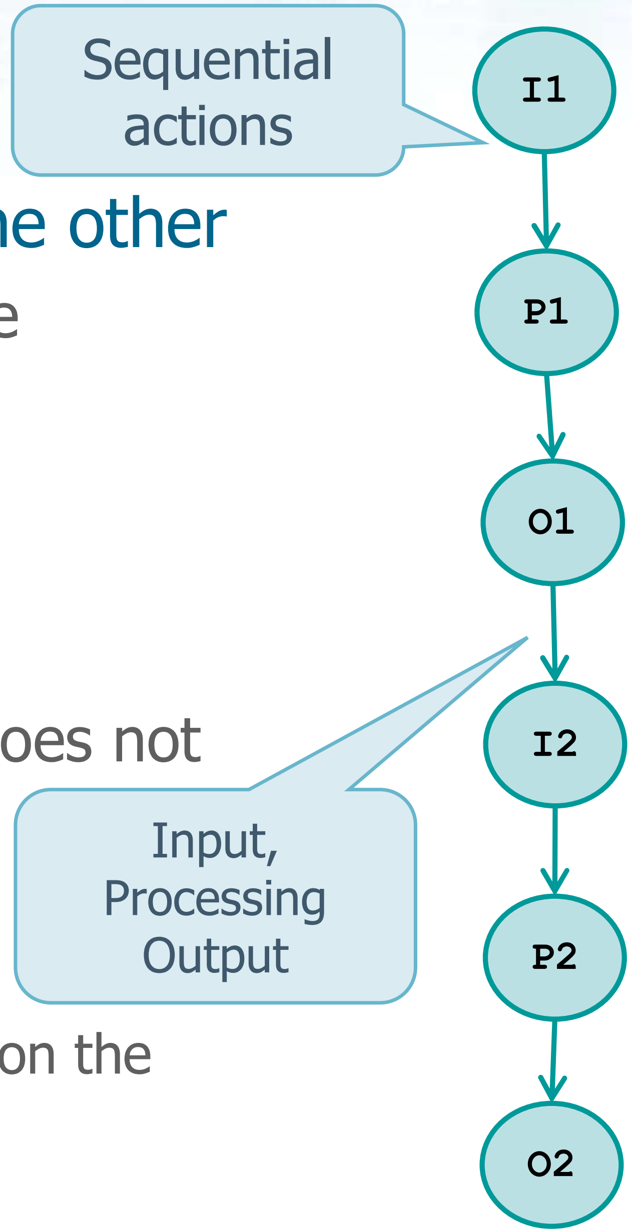
Process

- A process is
  - A running program
    - It includes a program counter, registers, variables, etc.
    - It is an active entity
- An operating system generates a process tree



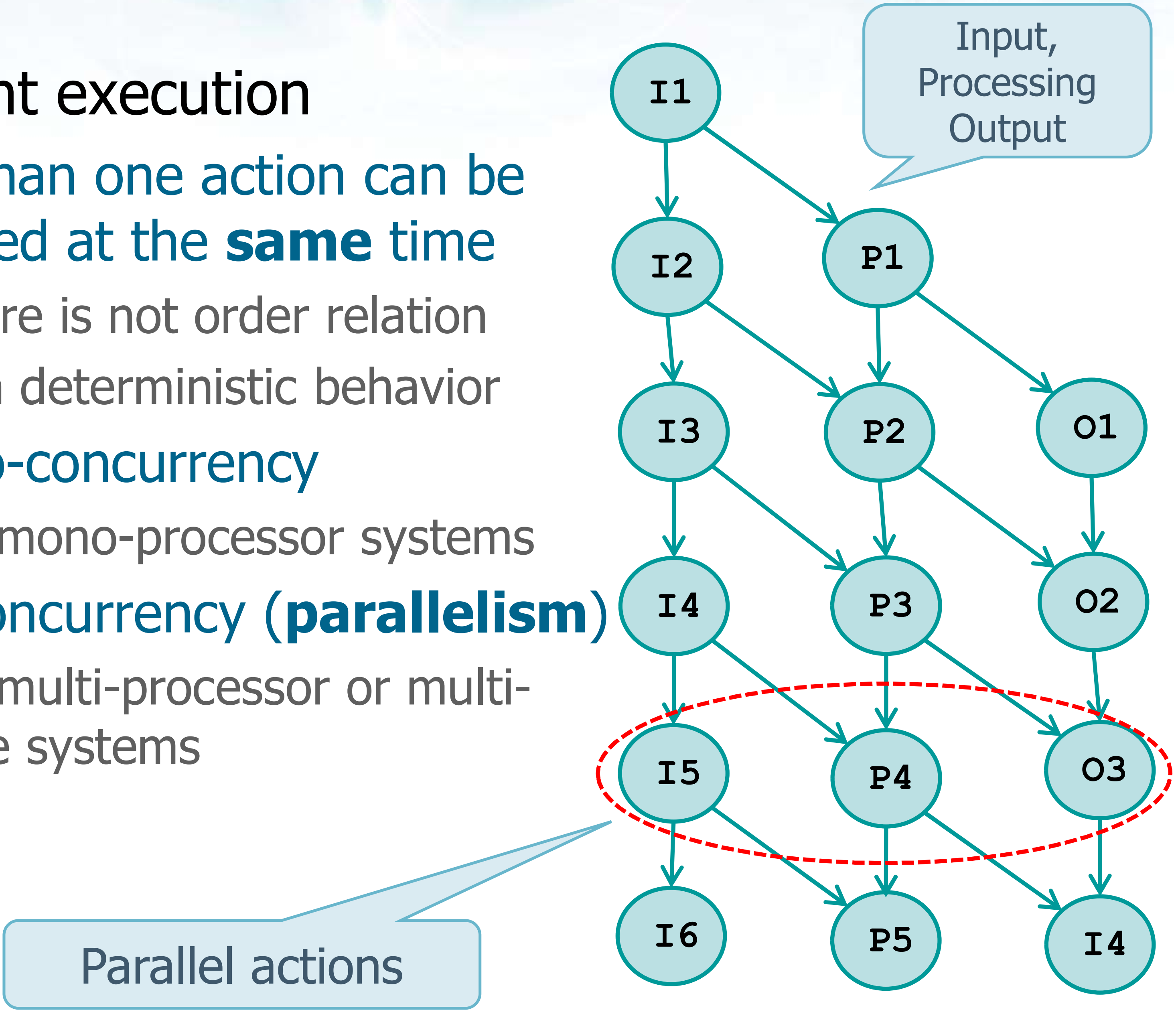
Sequential processes

- Sequential execution
  - Actions are executed one **after** the other
    - A new action begins only after the termination of the previous one
    - They are totally ordered
  - Deterministic behavior
    - Given the same input, the output produced is always the same, it does not depend on
      - The time of execution
      - The speed of execution
      - The number of active processes on the same system



Parallel processes

- Concurrent execution
  - More than one action can be executed at the same **same** time
    - There is not order relation
    - Non deterministic behavior
  - Pseudo-concurrency
    - On mono-processor systems
  - Real concurrency (**parallelism**)
    - On multi-processor or multi-core systems



Process identifier

- Every process has a unique identifier
  - PID or Process Identifier
- The PID is a non negative integer
  - Although a PID is unique, UNIX reuses the numbers of terminated processes
  - PID can be used by concurrent processes for creating unique objects, or temporary filenames
    - For example, to create a different process-dependent filename it is possible to do
      - sprintf(filename, "file-%d" getpid());

Process identification

```
#include <unistd.h>

pid_t getpid();
pid_t getppid();
uid_t getuid();
uid_t geteuid();
gid_t getgid();
gid_t getegid();
```

There is no system call to obtain the PID of a child

- In addition to the PID, there are other identifiers related to a process
  - **getpid** returns the identifier of the calling process
  - **getppid** returns the identifier of the parent process

Process creation

- System call **fork** creates a new **child** process
  - The child is a copy of the parent excluding the Process ID (**PID**) returned by **fork**
    - The **parent** process receives the child PID
      - A process may have more than one child that can identify on the basis of its PID
    - The **child** process receives the value 0
      - It can identify its parent by means of the system call **`getppid`**
  - **fork** is issued **once** in the parent process, but it returns **twice**, in different processes, and returns different values to the parent, and to the child

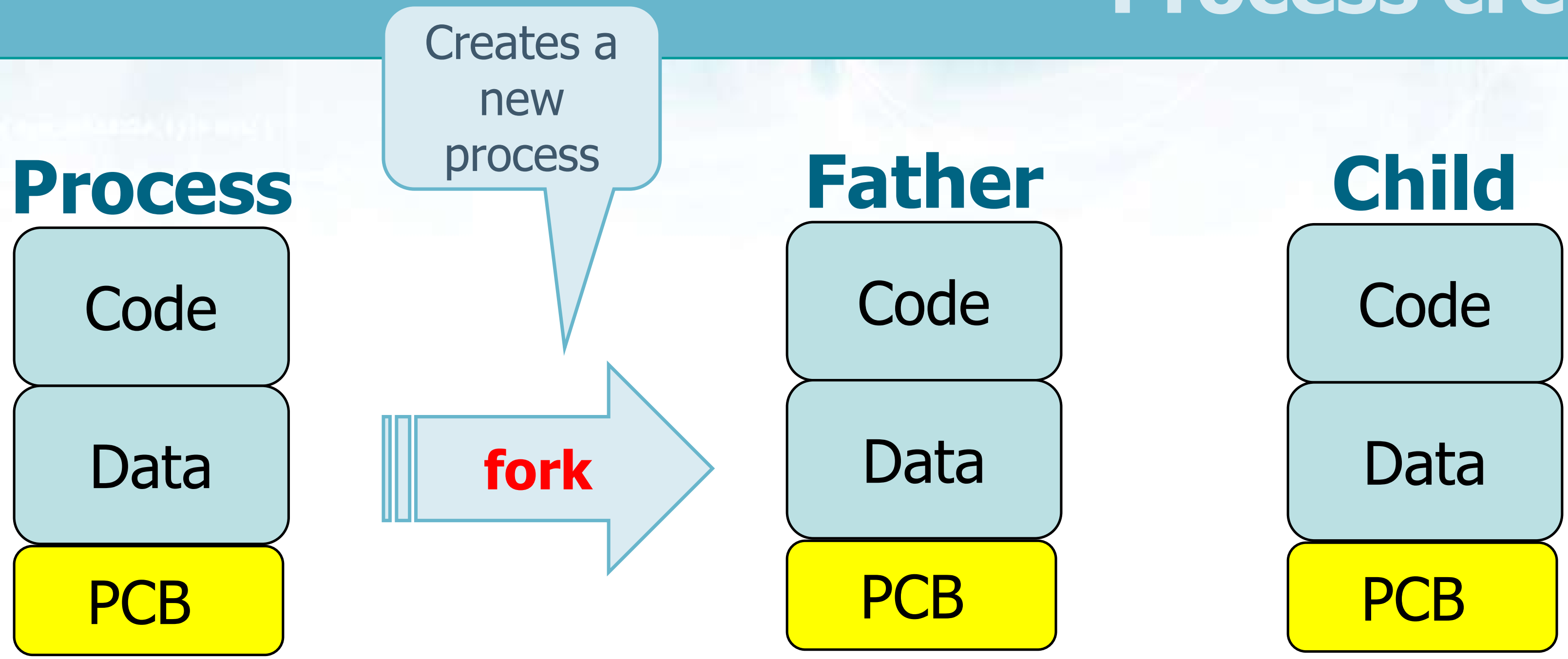
Process creation

```
#include <unistd.h>

pid_t fork (void);
```

Variants: vfork, rfork, clone

- Return value
  - If **fork** returns without error
    - The child PID, in the parent process
    - The value 0, in the child process
  - **fork** returns -1 in case of error
    - Normally because a limit on the number of allowed process has been reached

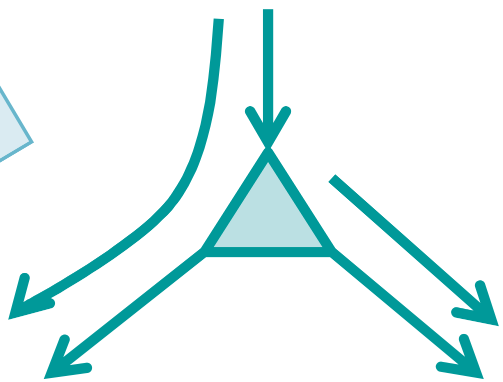
Process creation



- Parent and child
  - Share the code
  - Just after the fork
    - Have an identical copy ot the data
    - May modify (distroy, create, etc.) data independently

Process creation

```
#include <unistd.h>
...
pid_t pid;
...
pid = fork();
switch (pid) {
  case -1:
    // Fork failure
    ...
    exit (1);
  case 0:
    // Child
    ...
  default:
    // Parent
    ...
}
...
```



Example

- Write a concurrent C program that allows
  - Creating a child process
  - Terminating the parent process before the child process
  - Or terminating the child process before the parent process

The system call
unsigned int sleep (unsigned int sec)
places the process in wait for (at least) sec seconds

- Output in both cases the Process Identifier of the terminating process and the Process Identifier of its parent.

Who is the parent's parent?

Who is the child's parent if the parent terminates before the child?

Example

```
#include <unistd.h>
...
printf ("Main   :                   ");
printf ("PID=%d; My parent PID=%d\n",
  getpid(), getppid());
...
pid = fork();
if (pid == 0) {
  sleep (tC);
  printf ("Child : PIDreturned=%d    ", pid);
  printf ("PID=%d; My parent PID=%d\n",
    getpid(), getppid());
} else {
  sleep (tP);
  printf ("Parent: PIDreturned=%d ", pid);
  printf ("PID=%d; My parent PID=%d\n",
    getpid(), getppid());
}
```

tC = atoi (argv[1]);
tP = atoi (argv[2]);

Child

Parent

Example

```
➢ ps
  PID TTY          TIME CMD
 2088 pts/10   00:00:00 bash
 2760 pts/10   00:00:00 ps
```

Shell status
(ps: prints process status)

Child waits 2 secs
Parent waits 5 secs

```
➢ ./u04s01e03-fork 2 5
Main   :                   PID=2813; My parent PID=2088
Child  : PIDreturned=0     PID=2814; My parent PID=2813
Parent: PIDreturned=2814 PID=2813; My parent PID=2088
```

Notice increasing
PID values

Child waits 5 secs
parent waits 2 secs

```
➢  ./u04s01e03-fork 5 2
Main   :                   PID=2815; My parent PID=2088
Parent: PIDreturned=2816 PID=2815; My parent PID=2088
➢  Child : PIDreturned=0     PID=2816; My parent PID=1
```

init process PID

Exercise

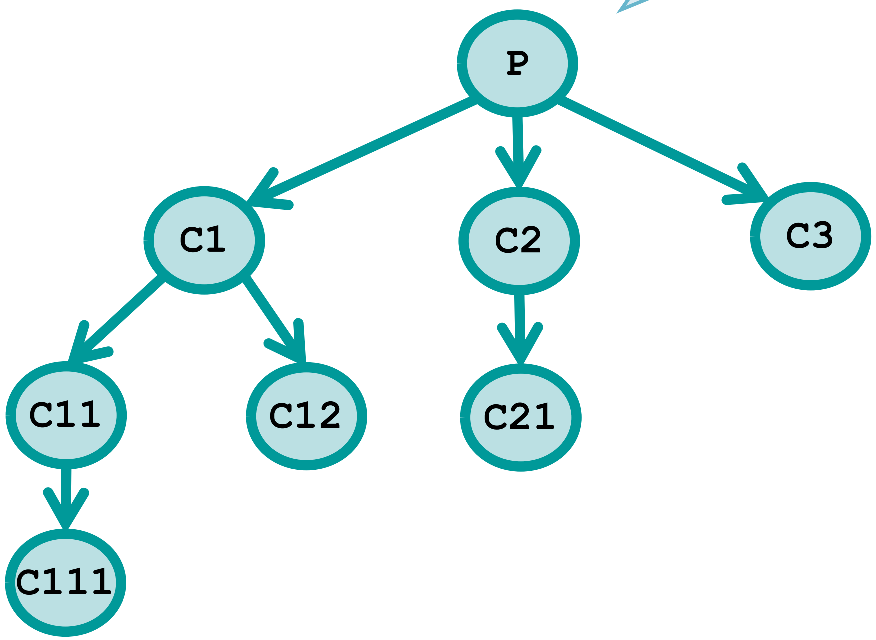
- Given the following program, draw its
  - Control Flow Graph, CFG
  - Process generation graph

```
int main () {
  /* fork a child process */
  fork();

  /* fork another child process */
  fork();

  /* fork a last one */
  fork();
}
```

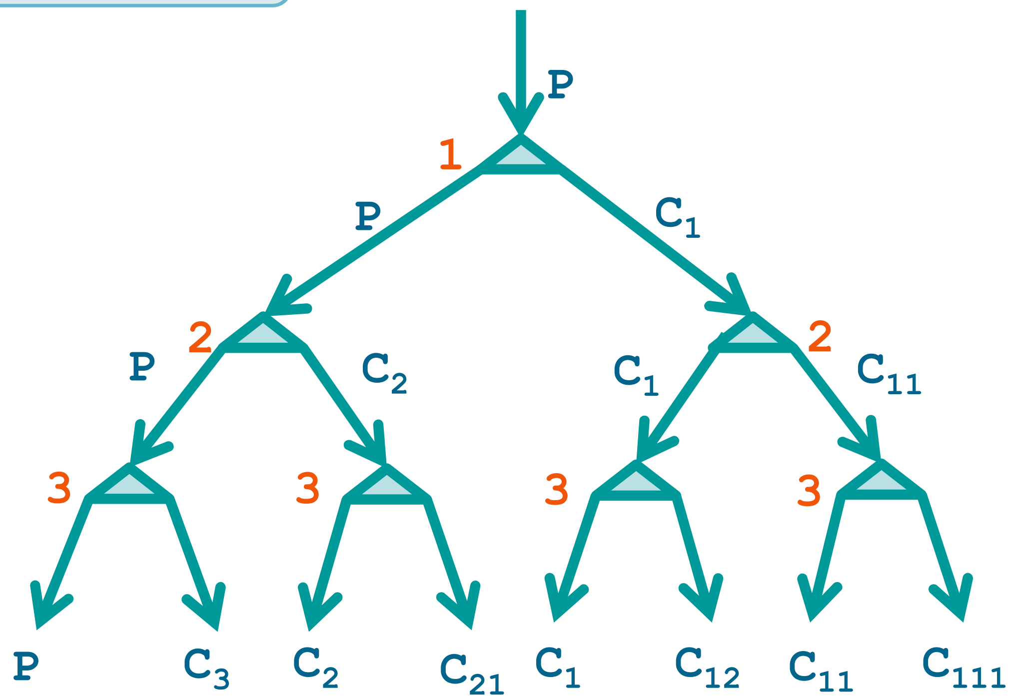
Solution

```
int main () {
  fork ();  // 1
  fork ();  // 2
  fork ();  // 3
}
```

Process generation tree



Control Flow Graph (CFG)



Exercise

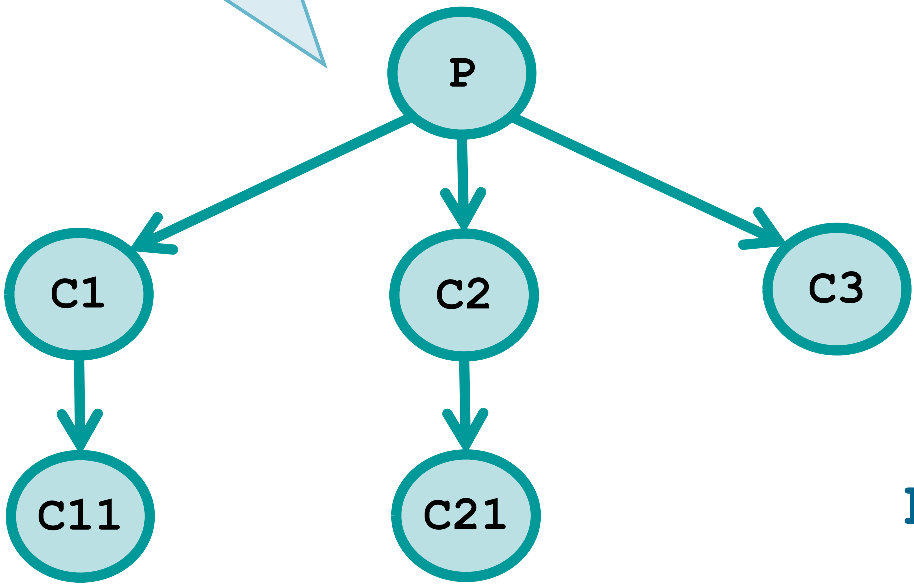
- Given the following program, draw its
  - Control Flow Graph, CFG
  - Process generation graph

```
pid = fork() /* call #1 */

fork();      /* call #2 */

if (pid != 0)
  fork();    /* call #3 */
```

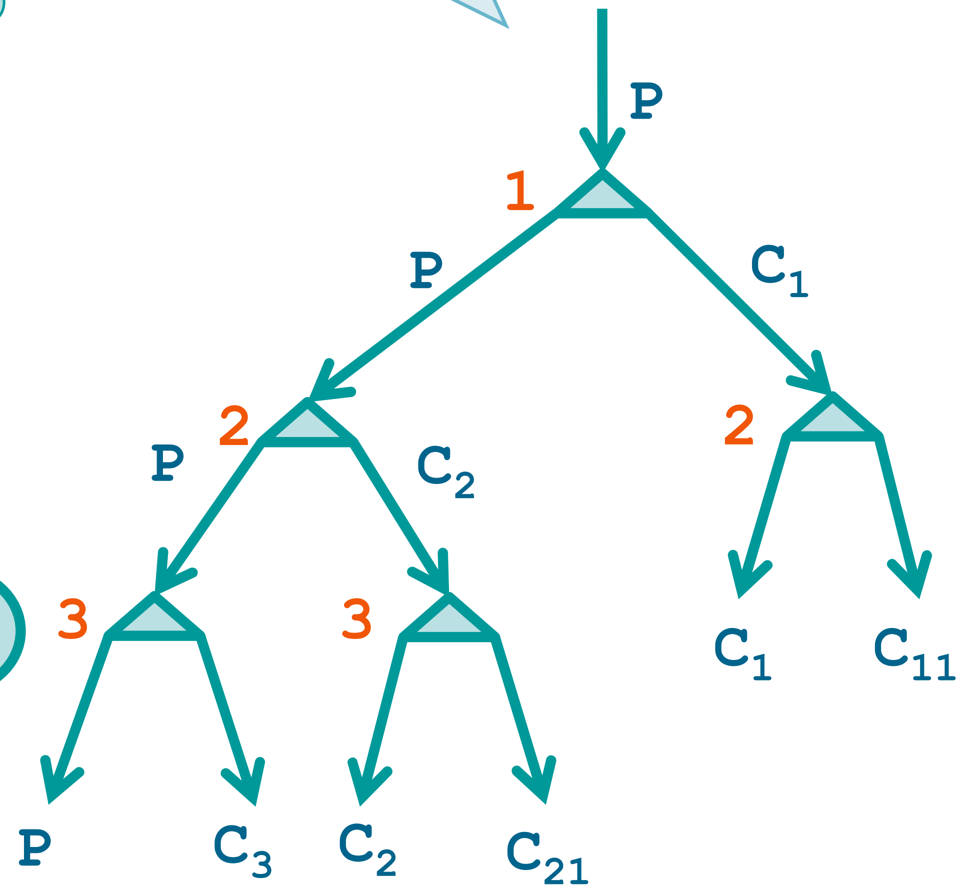
Solution

```
pid = fork()      // 1
fork();    // 2
if (pid != 0)
   fork();        // 3
```

Process generation tree



Control Flow Graph (CFG)



Resources

- The child process is a new entry in the Process Table
- In UNIX/Linux parent and child **share**
  - The source code (C)
  - The open file descriptors (File Description Table)
    - In particular, `stdin`, `stdout`, and `stderr`
      - Concurrent I/O operation implies producing interlaced I/O
  - User ID (UID), Group ID (GID), etc.
  - The root and the working directory
  - System resources and their utilization limits
  - Signal Table

Resources

- In UNIX/Linux parent and child have **different**
  - Return fork value
  - PID
    - The parent keeps its PID
    - The child gets a new PID
  - Data, heap and stack space
    - The **initial value** of the variables is inherited, but the spaces are completely separated
    - The **copy-on-write** technique is used by modern OSs
      - New memory is allocated only when one of the processes changes the content of a variable

Process termination

- A process can terminate in a normal way by issuing
  - A **return** from **main**
  - An **exit** system call
  - An **_exit** or **_Exit**
    - Synonyms defined in ISO C or POSIX
    - Similar effects of **exit**, but different management of stdio flushing etc.
  - A **return** from **main** of the last process thread
  - A **pthread_exit** from the last process thread

Process termination

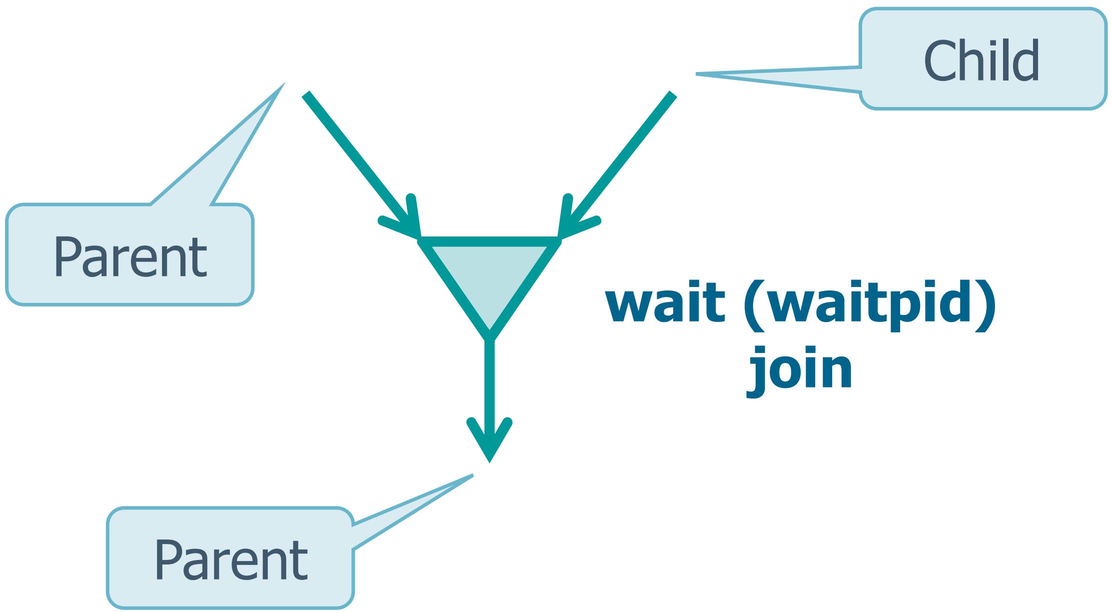
- Three not-normal method for process termination
  - Call of the function **abort**
    - Generates the signal **SIGABORT**, this is a sub-case of the next because a signal is generated
  - If a termination signal, or a signal not caught is received
  - If the last thread of a process is cancelled

System call wait and waitpid

- When a process terminates (normally or not)
  - The kernel sends a signal (**SIGCHLD**) to its parent
  - For the parent this is an asynchronous event
  - The parent process may
    - Manage the child termination (and/or the signal)
      - **Asynchronously**
      - **Synchronously**
    - **Ignore** the event (**default**)

System call wait and waitpid

- A parent process can manage child termination
  - Asynchronously: using a signal handler for signal **SIGCHLD**
    - This approach will be introduced in the section devoted to signals
  - Synchronously: by means of the system calls
    - **wait**
    - **waitpid**



System call wait

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

pid_t wait (int *statLoc);
```

Returned values:
a child's PID on success
-1 on failure

- The system call **wait**
  - Returns an error, **if** the calling process has not children
  - Blocks the calling process, **if** all its children are running (none is already terminated)
  - Returns to the process (immediately) the termination status of a child, **if** at least one of the children has ended (and it is waiting for his termination status to be recovered)

System call wait

- The **statLoc** parameter
  - Is an integer pointer
    - If not NULL collects the exit value of the child
  - The status information are
    - Implementation dependent
    - Recovered using macros defined in **<sys/wait.h>** (**WIFEXITED**, **WIFSIGNALED**, etc.)

Exit status of the child process

```
pid_t wait (int *statLoc);
```

System call waitpid

- To use **wait** for a specific child, you need to
  - Control the PID of the terminated child
  - Possibly store the PID of the terminated child in the list of terminated child processes (for future checks/searches)
  - Make another wait until the desired child is terminated
- If a parent needs to wait a specific child it is better to use **waitpid**, which
  - suspends execution of the calling process until a child, specified by *pid* argument, has changed state
  - **waitpid**() has a not blocking form (not default)

System call waitpid

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

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

- If a parent needs to wait a specific child it is better to use **waitpid** than **wait**

System call waitpid

- Parameters
  - **pid** allows waiting for
    - Any child (waitpid==wait) if pid = -1
    - The child whose PID=pid, if pid > 0
    - Any child whose GID is equal to that of the calling process , if pid = 0
    - Any child whose GID=abs(pid) if pid < -1
  - **statLoc** has the same meaning analyzed for wait
  - **options** allow additional controls
    - Non-blocking form of **wait** (use WNOHANG)

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

Zombie processes

- A child process terminated, whose parent is running, but has not executed **wait** is in the **zombie** state
  - The data segment of the process **remains** in the process table because the parent could need the child exit status
  - The child entry is removed only when the parent executes **wait**
    - Many zombie processes may remain in the system if one or more parents do not execute their wait system call.

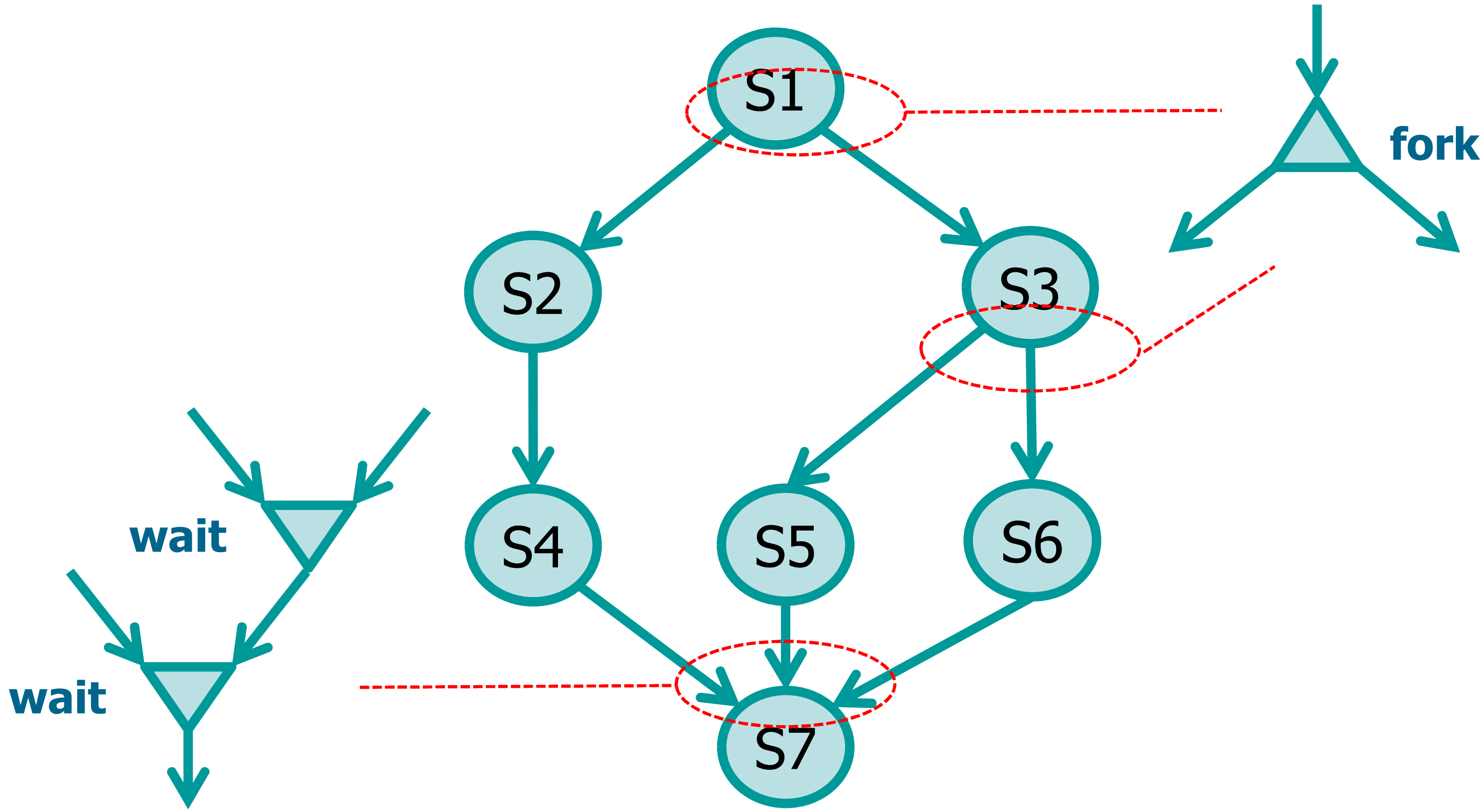
Orphan processes

- It the parent process terminates (without executing **wait**, and the child is still running, the latter is inherited by **init** the process (PID=1)
- The child does not become zombie because the system knows that no one is waiting for its exit status

Remember that in recent OS: "jobs started are not reparented to PID1 (init), but to a custom init -user, owned by the same user of the process ..."

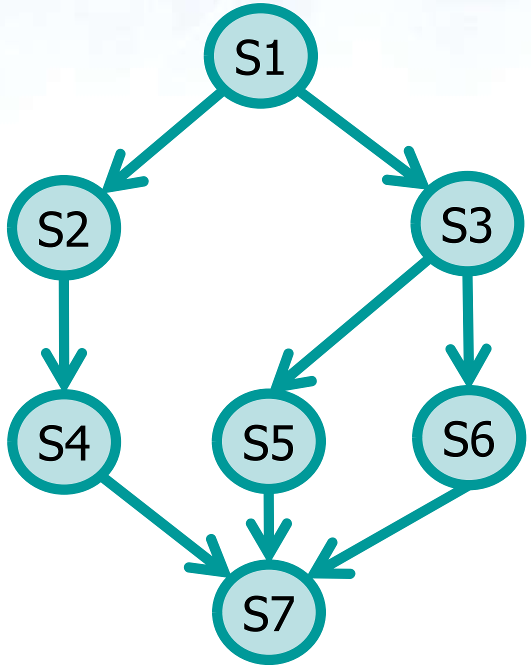
Exercise

- Implement this Control Flow Graph (CFG) by means of the system calls **fork** and **wait**



Solution

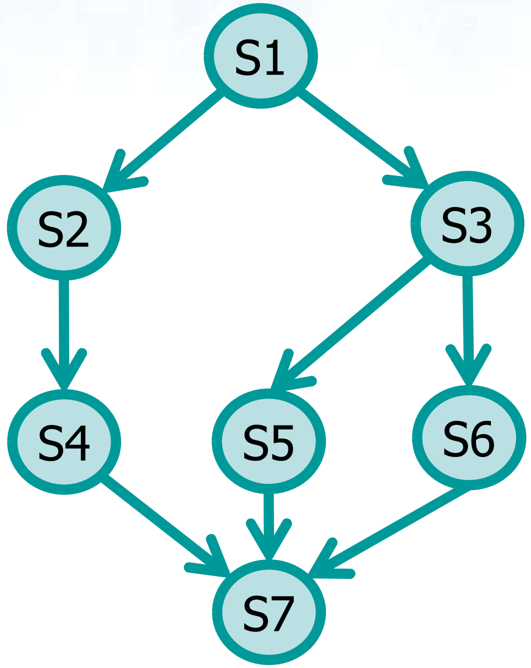
```
int main () {
  pid_t pid;
  printf ("S1\n");
  if ( (pid = fork())==-1 )
    err_sys( "can't fork" );
  if ( pid == 0 ){
    P356();
  } else {
    printf ("S2\n");
    printf ("S4\n");
    while (wait((int *)0)!= pid);
    printf ("S7\n");
    exit (0);
  }
  return (1);
}
```



Check on different terminations (useless in this case and replaceable with waitpid)

Solution

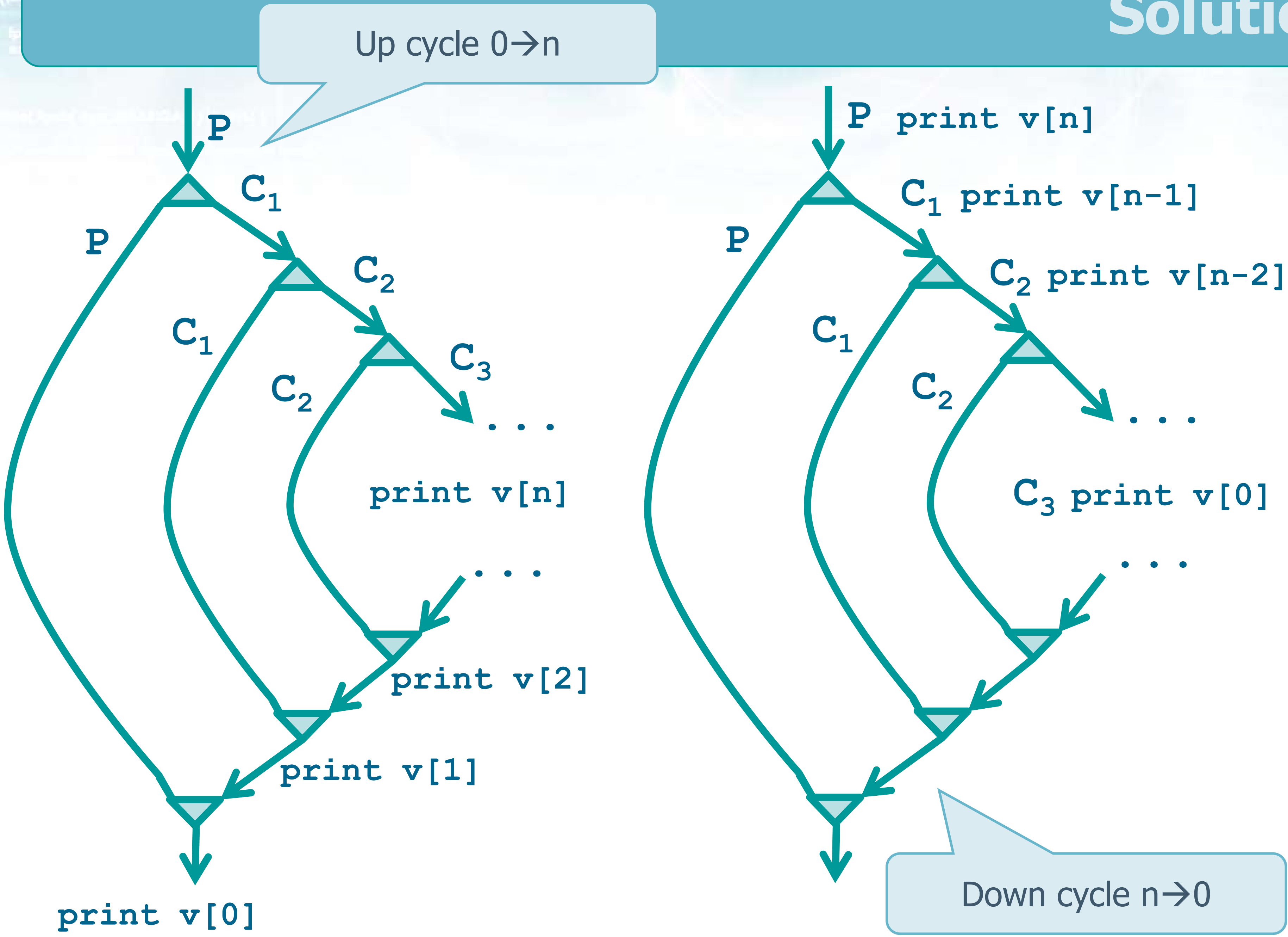
```
P356() {
  pid_t pid;
  printf ("S3\n");
  if ( ( pid = fork() ) == -1 )
    err_sys( "can't fork" );
  if (pid > 0 ){
    printf ("S5\n");
    while (wait((int *)0)!=pid );
  } else {
    printf ("S6\n");
    exit (0);
  }
  exit (0);
}
```



Exercise

- Write a program that
  - Takes as argument an integer value **n**
  - Allocates dynamically an integer vector of dimension **n**
  - Fills the vector with values reads from the terminal
  - Displays the vector content, from the last to the first element, using **n-1** processes, each displaying a single element of the vector
  - Hint
    - Synchronize the processes by means of **wait** system calls, in order to establish the order of display of the elements of the vector

Solution



Up cycle 0→n
P
C1
P
C2
C1
C3
C2
. . .
print v[n]
. . .
print v[2]
print v[1]
print v[0]
P print v[n]
C1 print v[n-1]
P
C2 print v[n-2]
C1
C2
. . .
C3 print v[0]
. . .
Down cycle n→0

Solution

Up-cycle 0→n

```
int main(int argc, char *argv[]) {
  int i, n, *vet;
  int retValue;
  pid_t pid;
  n = atoi (argv[1]);
  vet = (int *) malloc (n * sizeof (int));
  if (vet==NULL) {
    fprintf (stderr, "Allocation Error.\n");
    exit (1);
  }
  fprintf (stdout, "Input:\n");
  for (i=0; i<n; i++) {
    fprintf (stdout, "vet[%d]:", i);
    scanf ("%d", &vet[i]);
  }
```

Solution

```
  fprintf (stdout, "Output:\n");
  for (i=0; i<n-1; i++) {
    pid = fork();
    if (pid>0) {
      pid = wait (&retValue);
      break;
    }
    // fprintf (stdout, "Run PID=%d\n", getpid());
  }

  fprintf (stdout, "vet[%d]:%d - ", i, vet[i]);
  // fprintf (stdout, "End PID=%d\n", getpid());

  exit (0);
}
```

Soluzione

Down-cycle n→0

```
...
  fprintf (stdout, "Output:\n");
  for (i=n-1; i>=0; i--) {
    fprintf (stdout, "vet[%d]:%d - ", i, vet[i]);
    pid = fork();
    if (pid>0) {
      pid = wait (&retValue);
      break;
    }
    // fprintf (stdout, "Run PID=%d\n", getpid());
  }

  // fprintf (stdout, "End PID=%d\n", getpid());
...
```