Processes

Introduction to Linux processes

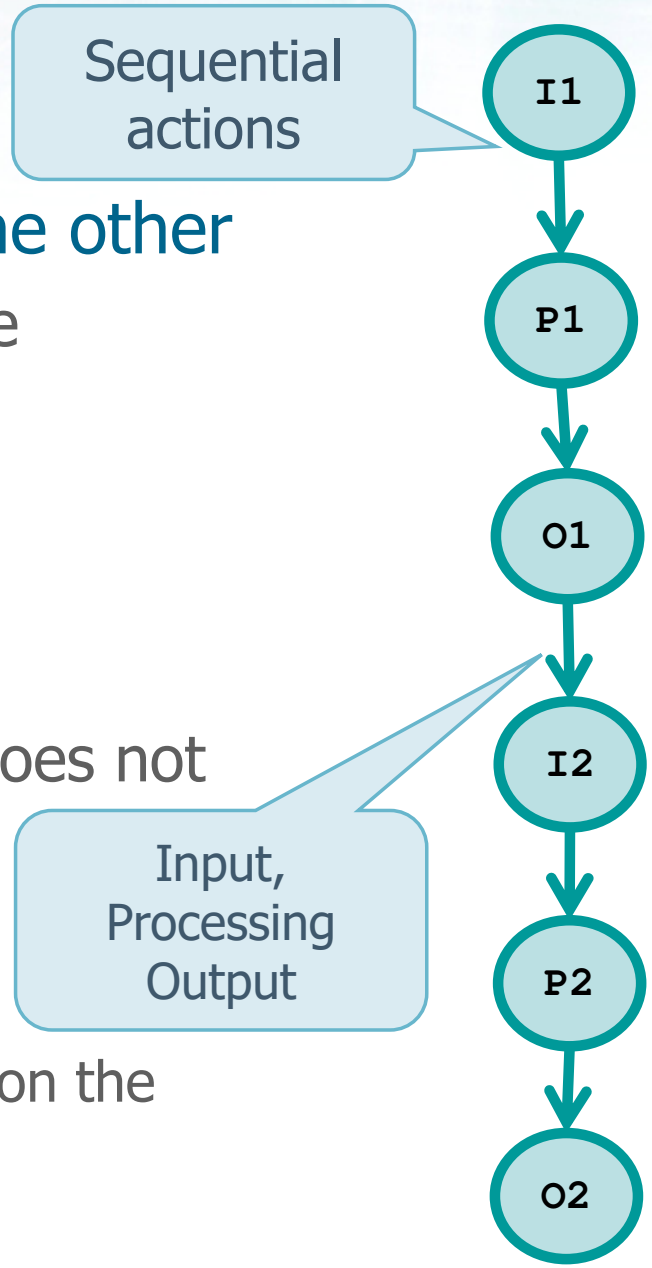
Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

Algorithms, Programs, and Processes

- Algorithm
  - Logical process which, in a finite number of steps, allows the resolution of a problem
- Program
  - Formalization of an algorithm by means of a programming language
  - Passive entity, i.e., file (executable) in the hard disk
- Process
  - Abstraction of a running program
  - Active entity
    - Sequence of operations performed by a program running on a given set of data

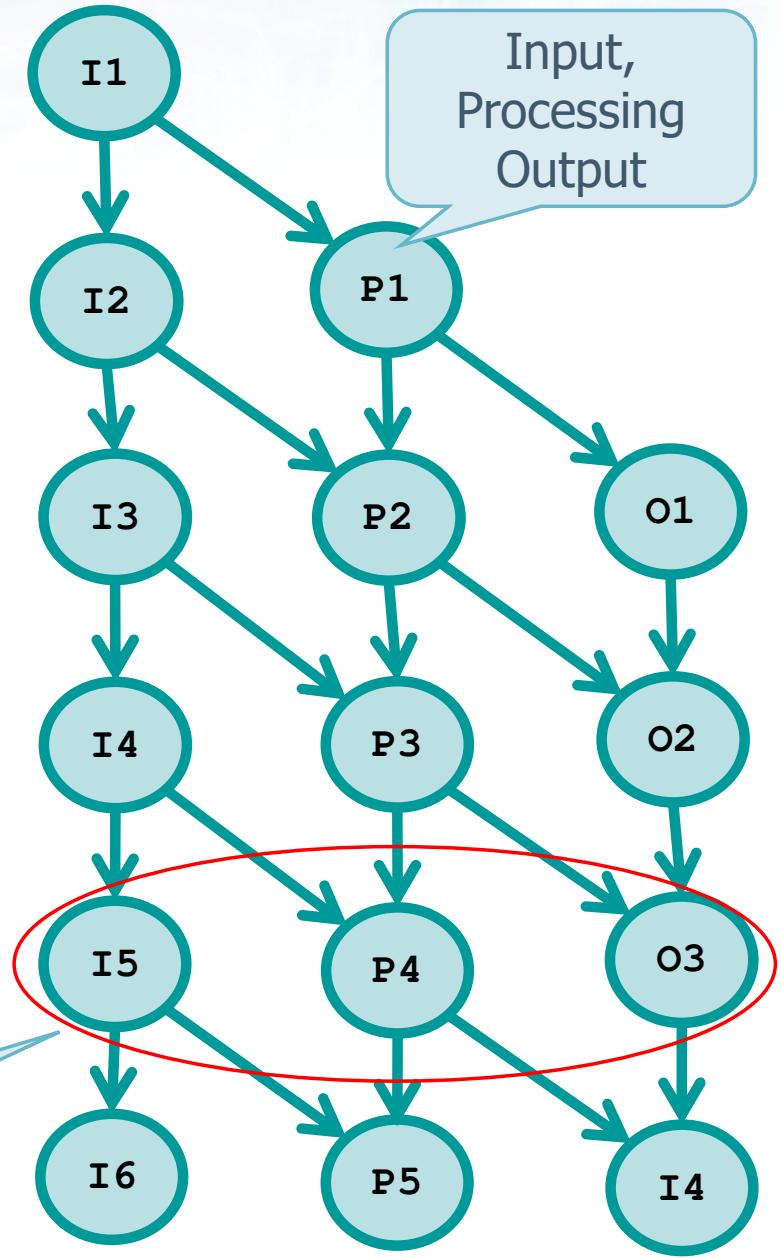
Sequential and concurrent processes

- 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



Sequential and concurrent processes

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



Processes

- Several processes are started at bootstrap
  - Automatic
    - Daemon Processes
    - E-mail applications
    - Various control activities, virus scan and others
    - …
  - On user request
    - Line printer management
    - WEB server (with parallel requests for outside)
    - …

Started at bootstrap, terminated at shut-down. Execute support activities.

Processes

- Process identity and process control
  - ID & system calls: `pid`, `getpid`, `getppid`, etc.
- Process creation
  - The creating process is called the **parent process**, the process created is called the **child process**
  - It is possible to create a **process tree**.
  - System calls: `fork`, `exec`, `system`, etc.
- Process synchronization and termination
  - System calls: `wait`, `waitpid`, `exit`.

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 :

```
sprintf(filename, "file-%d" getpid());
```

creates a different process-dependent filename

Process identification

- Some identifiers are reserved
  - The first process, PID=0, is a system process
    - The **swapper**, which is responsible for memory management and process scheduling
    - Executed at the kernel level
  - The second process, PID=1, is **init** a daemon
    - invoked at the end of the bootstrap
    - executed at user level
    - with super-user privileges
    - Becomes the parent of each orphan process, i.e., of a child of a parent process already terminated

Recent OS: "jobs started are not reparented to PID1 (init), but to a custom init -user, owned by the same user of the process …"

Process identification

```
#include <unistd.h>

pid_t getpid();     // Process ID
pid_t getppid();    // Parent Process ID

uid_t getuid();     // User ID
gid_t getgid();     // Group ID
```

- 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

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

Identification of a process

```
#include <unistd.h>

uid_t getuid();
gid_t getgid();
```

```
#include <unistd.h>

uid_t geteuid();
gid_t getegid();
```

- To UID and GID are also associated Effective-UID and Effective-GID
  - A process with a UID (GID) can change it identity assuming a different EUID (EGID)
    - Example
      - The command **passwd** allows to change the password of a user; this requires root permissions; as a consequence, the process **passwd** with l'UID of the user assumes the EUID of root to preform the operation

Process creation

- Windows and UNIX/Linux use different procedures
- With Windows API a process is created by means of the system call **CreateProcess**
  - In practice it executes a new process specifying the executable
  - The new process is distinct from the caller
- In UNIX/Linux a new process is generated by means of the system call **fork**
  - It clones/duplicates the current process

Process creation

- In Windows the **CreateProcess** assumes the typical style of Windows API
  - Verbosity, high number of parameters, high typing, etc.

```
BOOL CreateProcess (
  LPCTSTR lpImageName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpsaProcess,
  LPSECURITY_ATTRIBUTES lpsaThread,
  BOOL bInheritHandles, DWORD dwCreate,
  LPVOID lpvEnvironment, LPCTSTR lpCurDir,
  LPSTARTUPINFO lpsiStartInfo,
  LPPROCESS_INFORMATION lppiProcInfo
);
```

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 returns in two different processes, and returns different values to the parent, and to the child.

Process creation

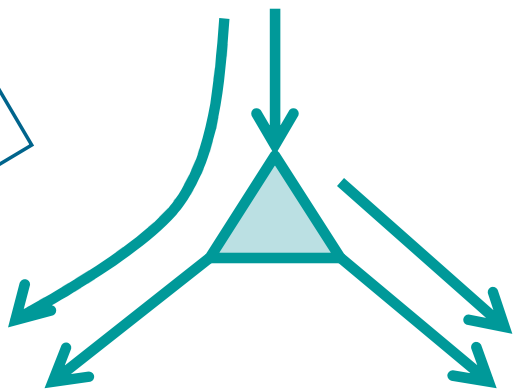
```
#include <unistd.h>

pid_t fork (void);
```

Variants: vfork, rfork, clone

- Returned values
  - If **`fork`** returns without error
    - Child PID in the parent process
    - Zero 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

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



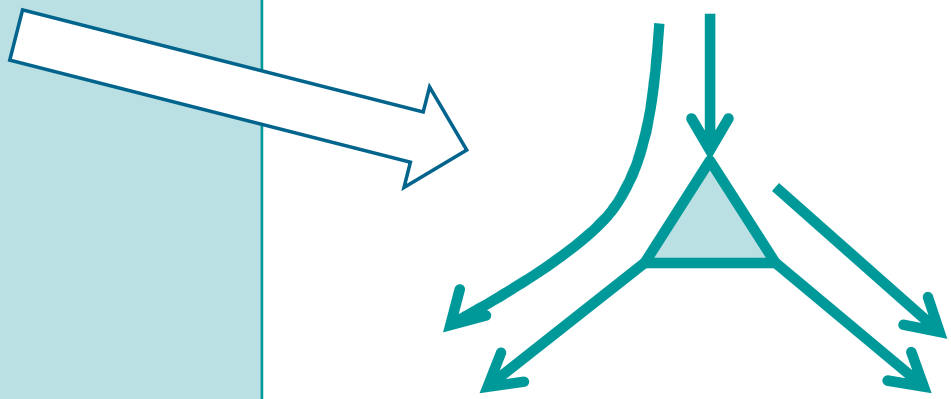
Process (parent)

Parent continues execution pid = child PID

Child continues execution pid = 0

Process creation

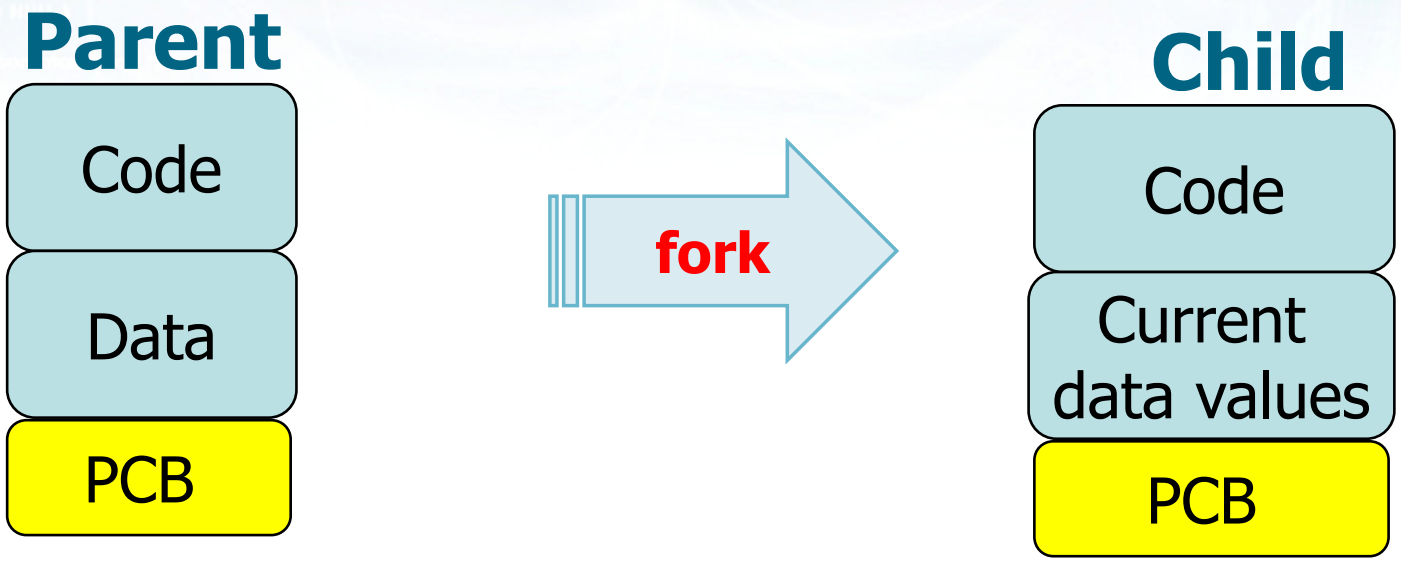
```
#include <unistd.h>
...
pid_t pid;
...
pid = fork();
if (pid>0) {
   // Parent
   ...
   } else {
   // Child
   ...
   }
...
}
...
```



Parent
continues execution
pid = child PID

Child
continues execution
pid = 0

Address space



- Parent and child share their code, but they have a different PCB
- They also share the **value of the data at the time of fork**.
- Parent and child may then change the data values independently

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

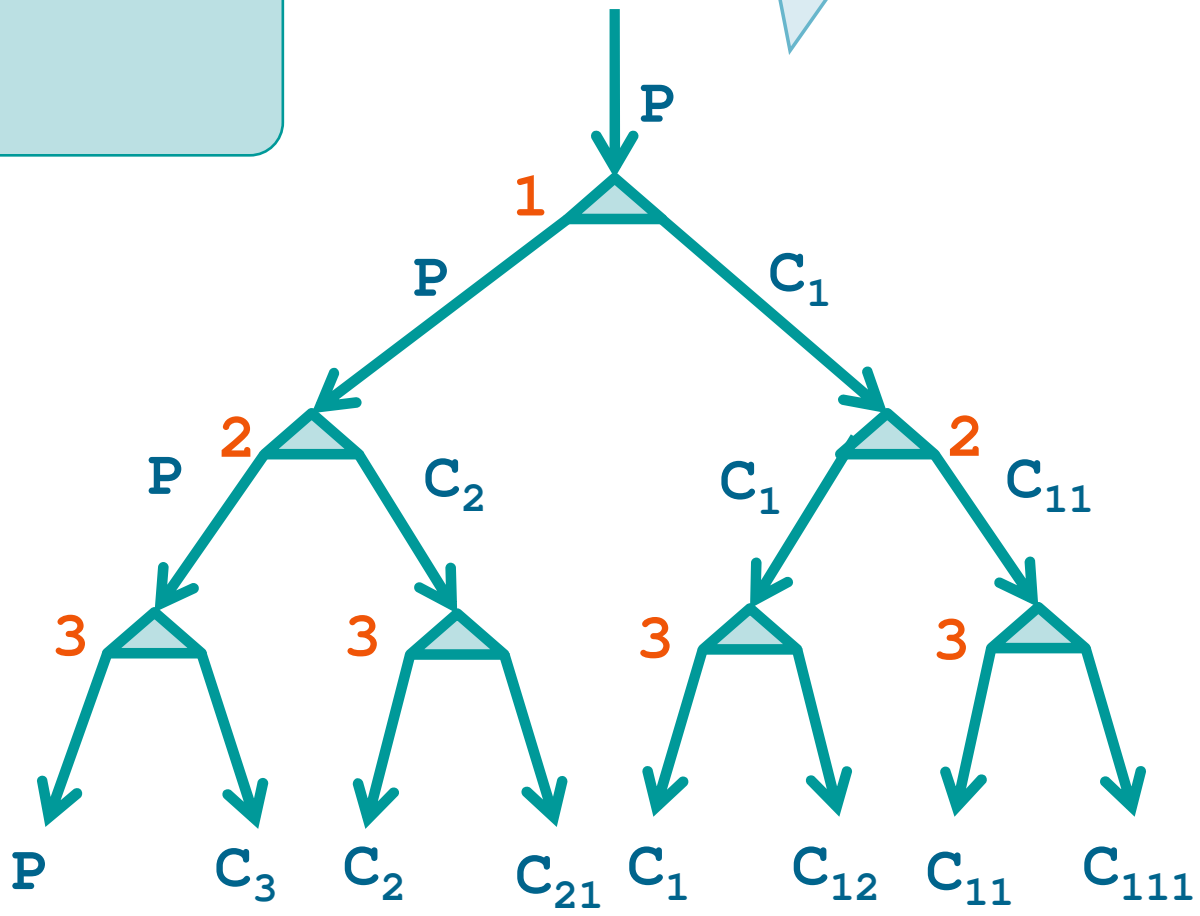
- Given the following program, draw its
  - Control Flow Graph, CFG
  - Process generation graph

```
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
}
```

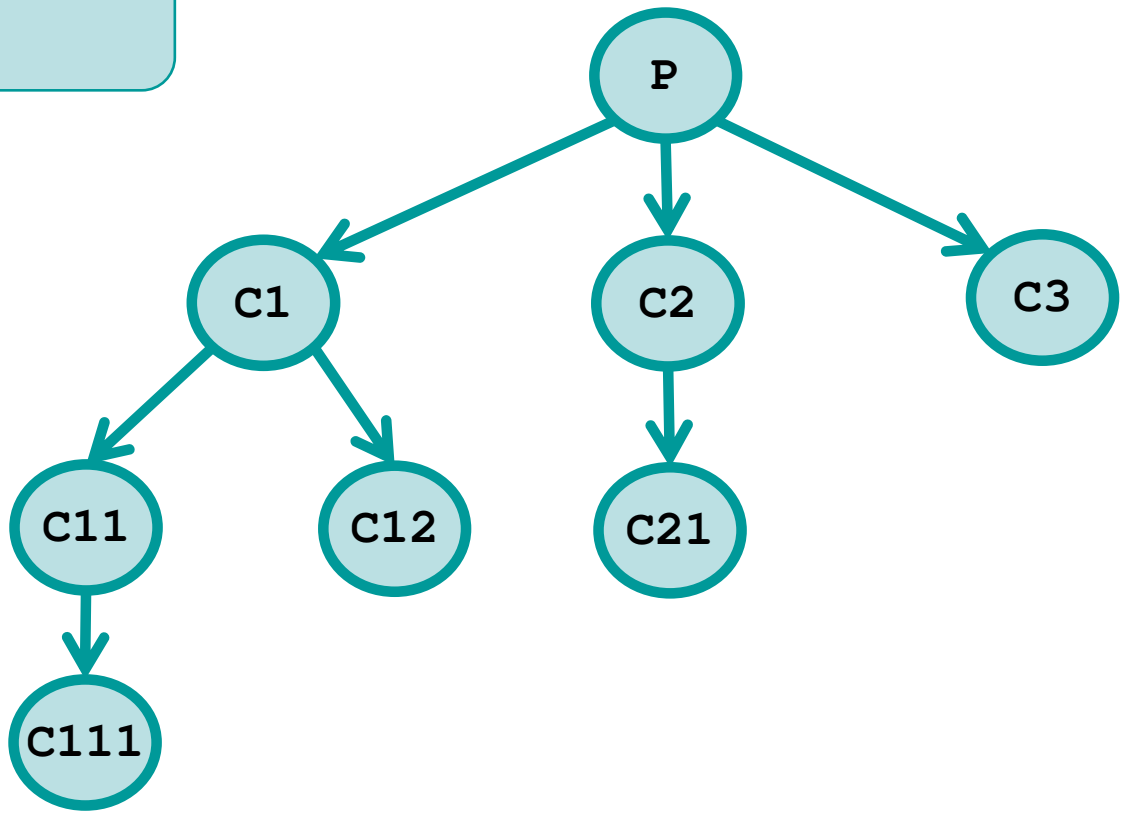
Control Flow Graph (CFG)



Solution

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

Process generation tree



Exercise

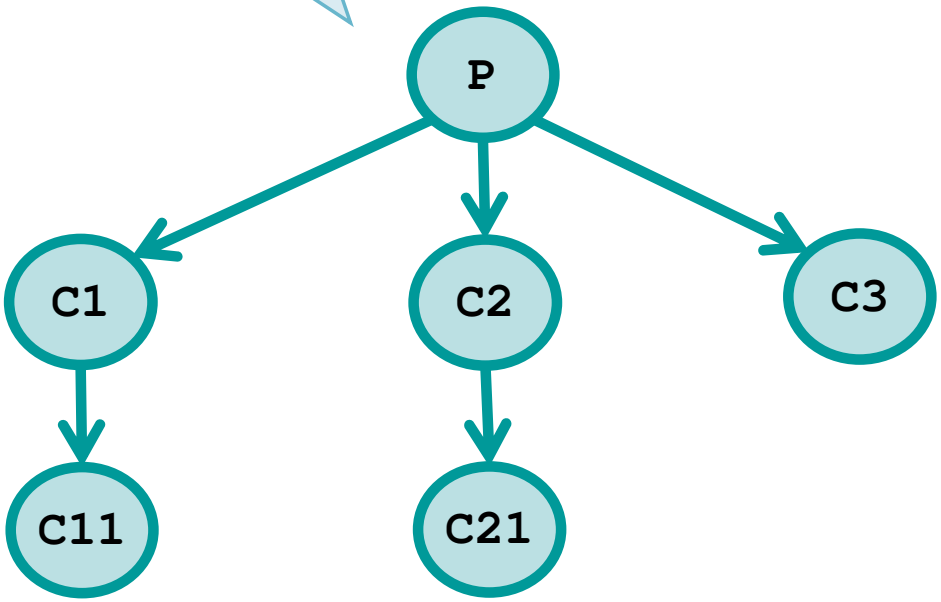
- Given the following program, draw its
  - Control Flow Graph, CFG
  - Process generation graph

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

if (pid != 0)
  fork ();      /* call #2 */

fork ();        /* call #3 */
```

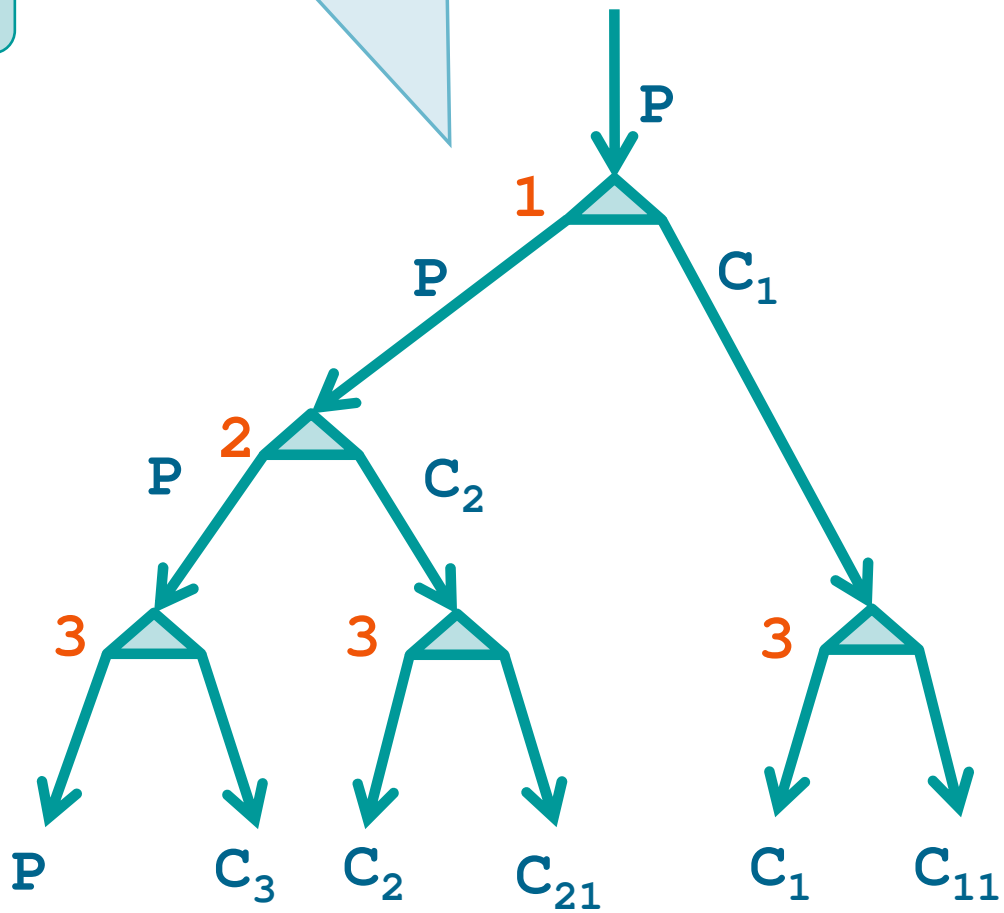
Solution

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

Process generation tree



Control Flow Graph (CFG)



Exercise

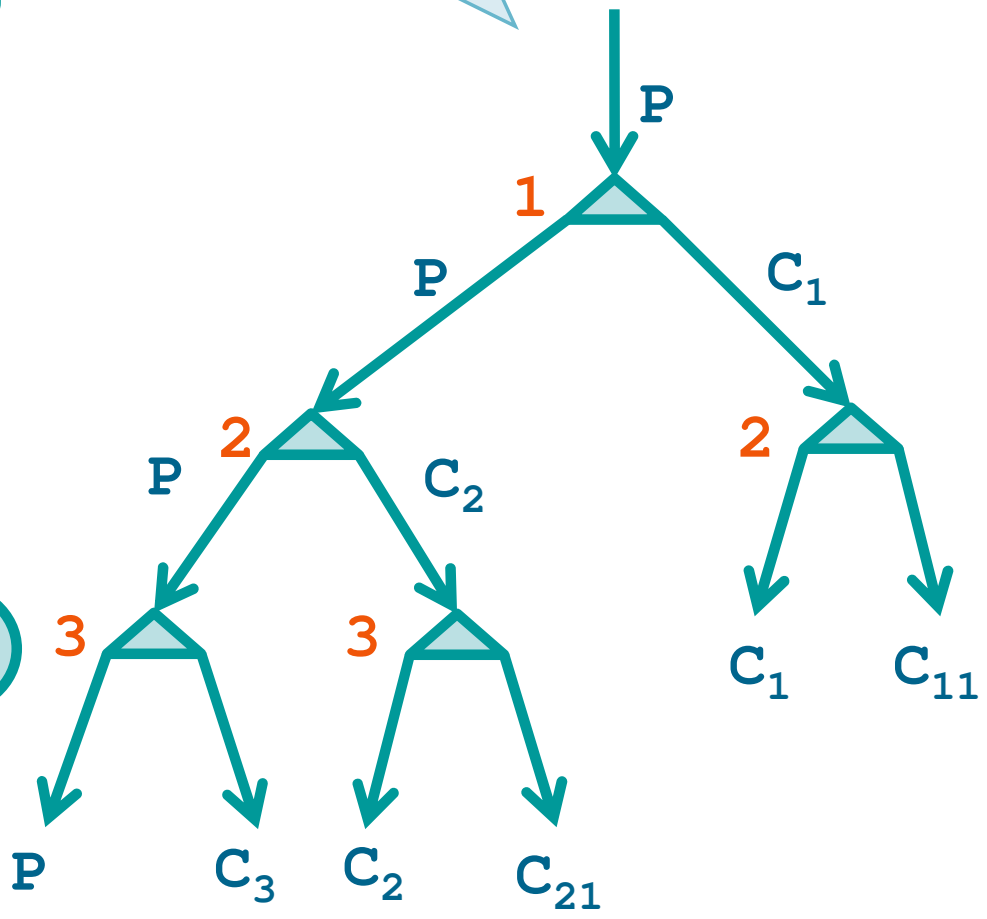
- Given the following program, draw its
  - Control Flow Graph, CFG
  - Process generation graph

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

fork();      /* call #2 */

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

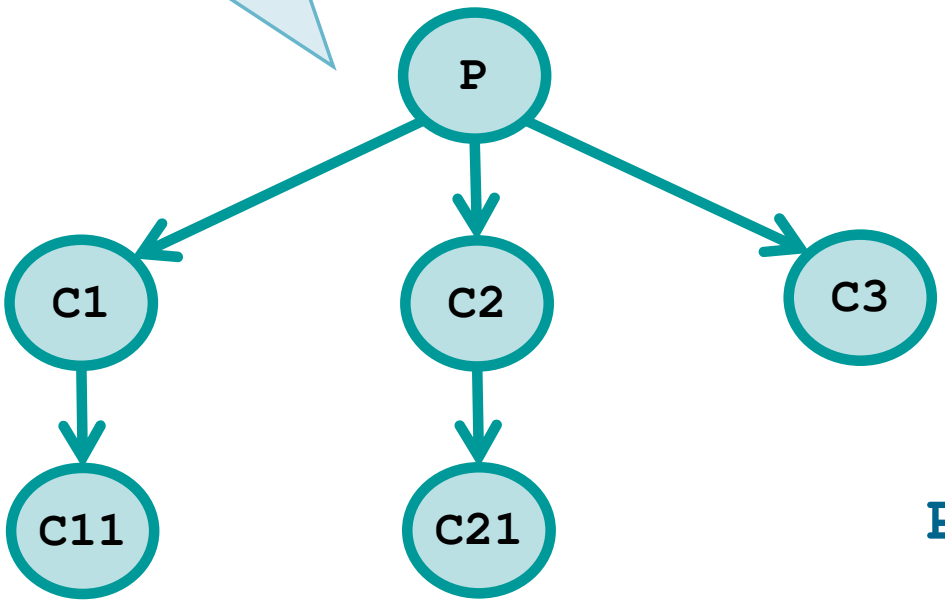
Solution

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

Control Flow Graph (CFG)



Process generation tree



Exercise

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

```
#include <stdio.h>

int main () {
  int i;
  for (i=0; i<2; i++) {
    printf("i: %d \n", i);
    if (fork())     /* call #1 */
      fork();    /* call #2 */
  }
}
```

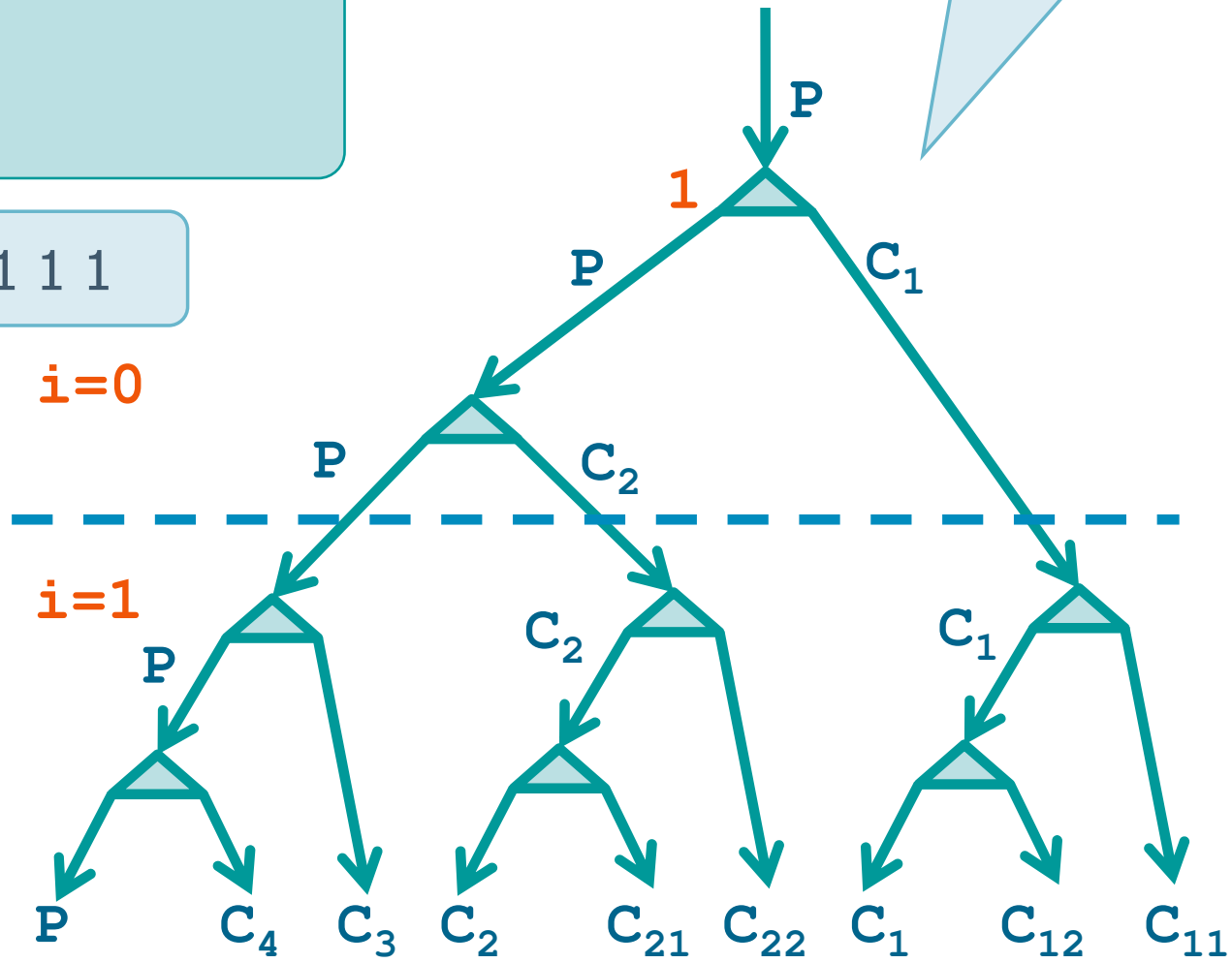
fflush (stdout);
or
setbuf (stdout, 0);

Solution

```
for (i=0; i<2; i++) {
  printf("i: %d \n", i);
  if (fork()) // 1
    fork();  // 2
}
```

Possible output : 0 1 1 1

Control Flow Graph (CFG)

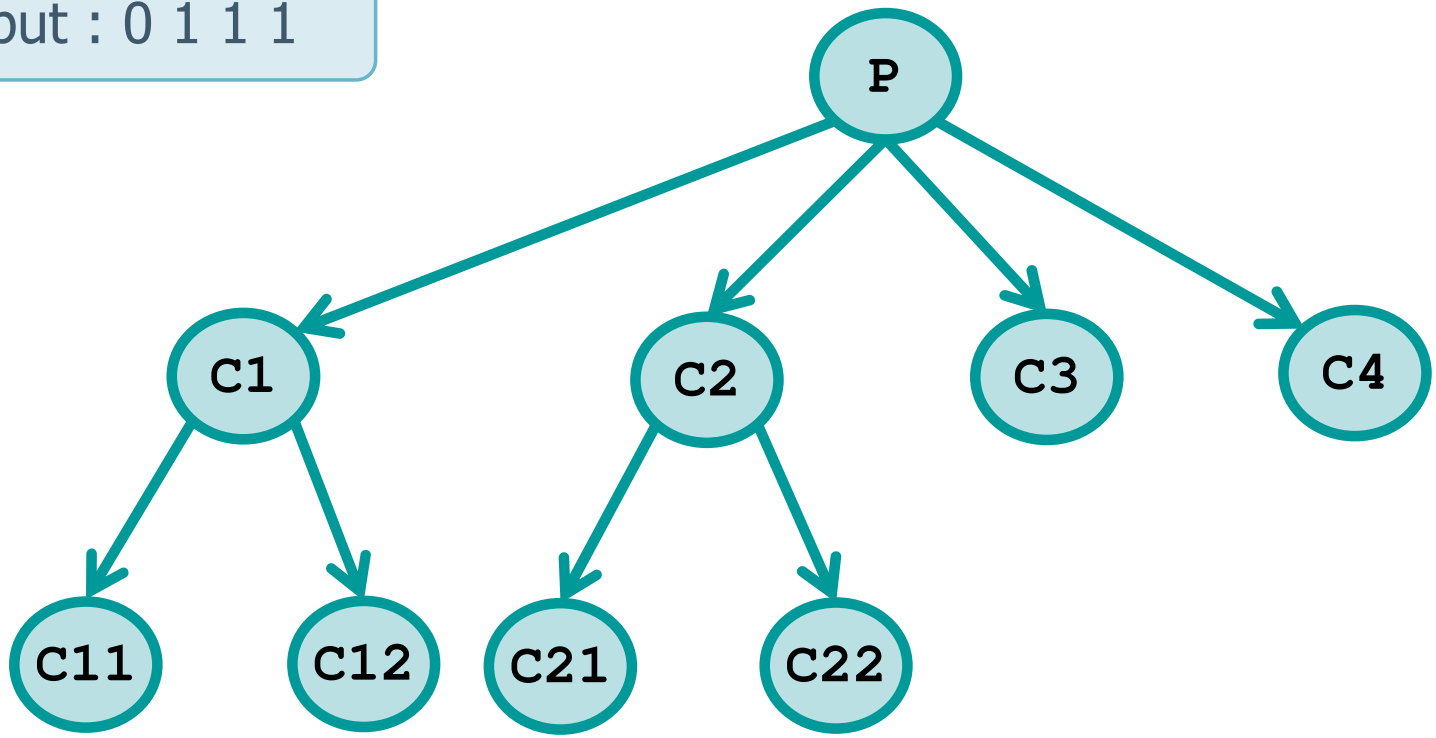


Solution

```
for (i=0; i<2; i++) {
  printf("i: %d \n", i);
  if (fork()) // 1
    fork();  // 2
}
```

Process generation tree

Possible output : 0 1 1 1



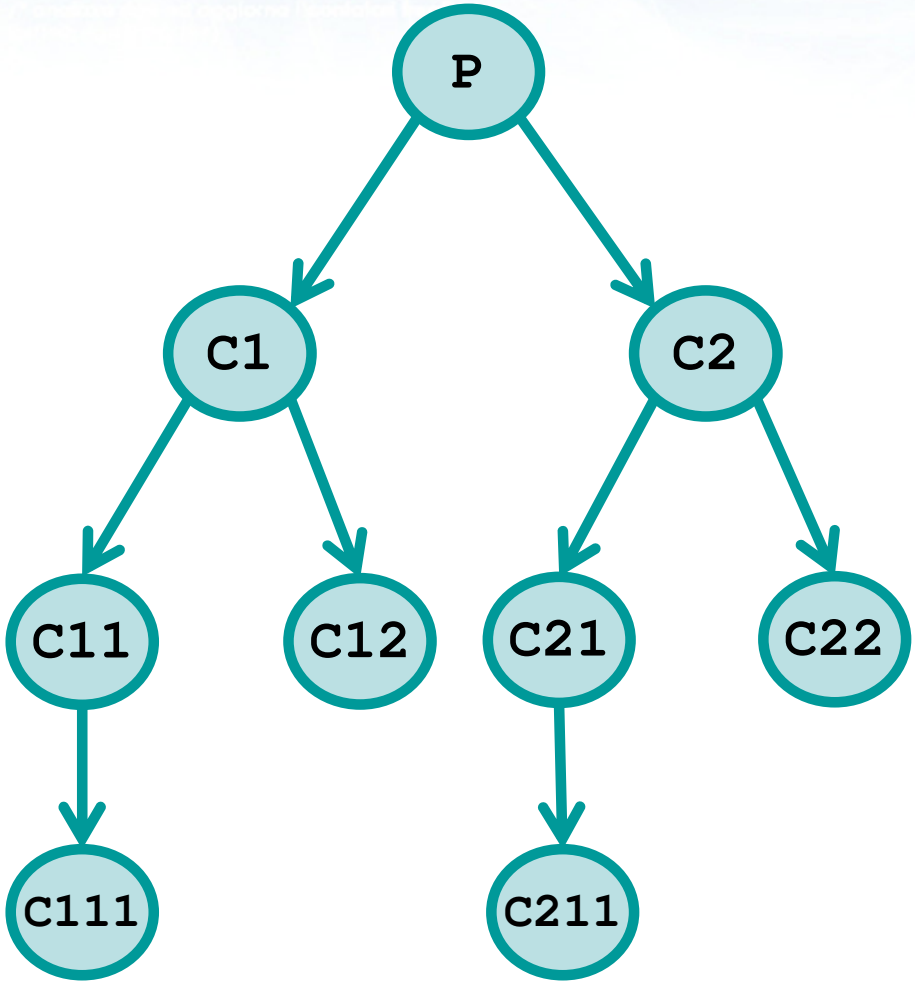
Exercise

- Given the following program, report the output and the process generation tree

```
int main() {
  int a, b=5, c;
  a = fork(); /* #1 */
  if (a) {
    a = b; c = split(a, b++);
  } else {
    fork(); /* #2 */
    c = a++; b += c;
  }
  if (b > c) {
    fork(); /* #3 */
  }
  printf("%3d", a+b+c);
  return 0;
}
```

```
int split(int a, int b) {
  a++;
  a = fork(); /* #4 */
  if (a) {
    a = b;
  } else {
    if (fork()) /* #5 */ {
      a--;
      b += a;
    }
  }
  return a+b;
}
```

Solution



P	a	b	c	a+b+c
P	5	6	10	21
C1	1	5	0	6
C2	5	6	3	14
C11	1	5	0	6
C12	1	5	0	6
C21	5	6	5	16
C22	5	6	3	14
C111	1	5	0	6
C211	5	6	5	16

Exercise

- Write a concurrent program that
  - Given `n` as its argument
  - Generates `n` children processes
- Each child process outputs its PID and terminates

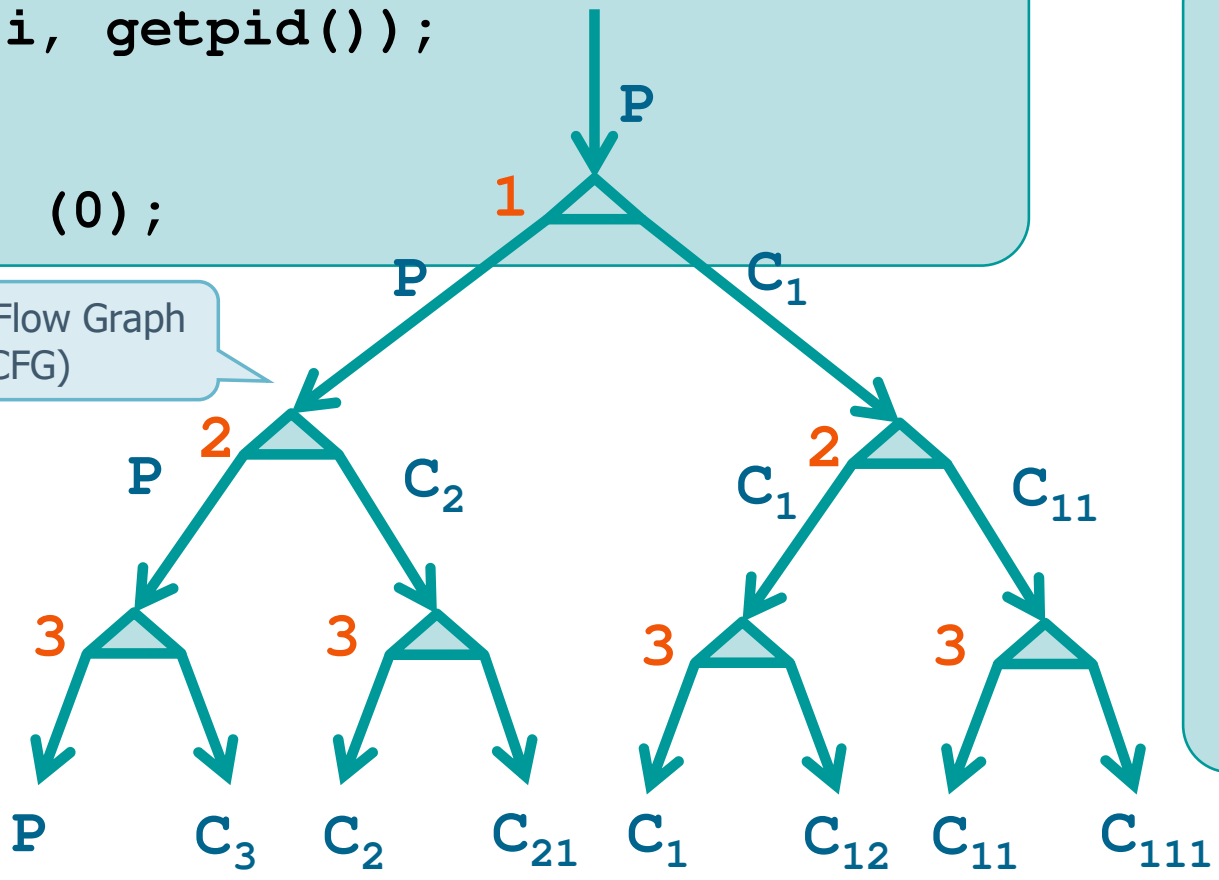
Solution 1

```
int i, n;

scanf ("%d", &n);
for (i=0; i<n; i++) {
  fork();
  printf ("Proc %d (PID=%d)\n",
    i, getpid());
}

exit (0);
```

Erroneous solution 1

```
int i, n;

scanf ("%d", &n);
for (i=0; i<n; i++) {
  fork();
  printf ("Proc %d (PID=%d)\n",
    i, getpid());
}

exit (0);
```

Control Flow Graph (CFG)



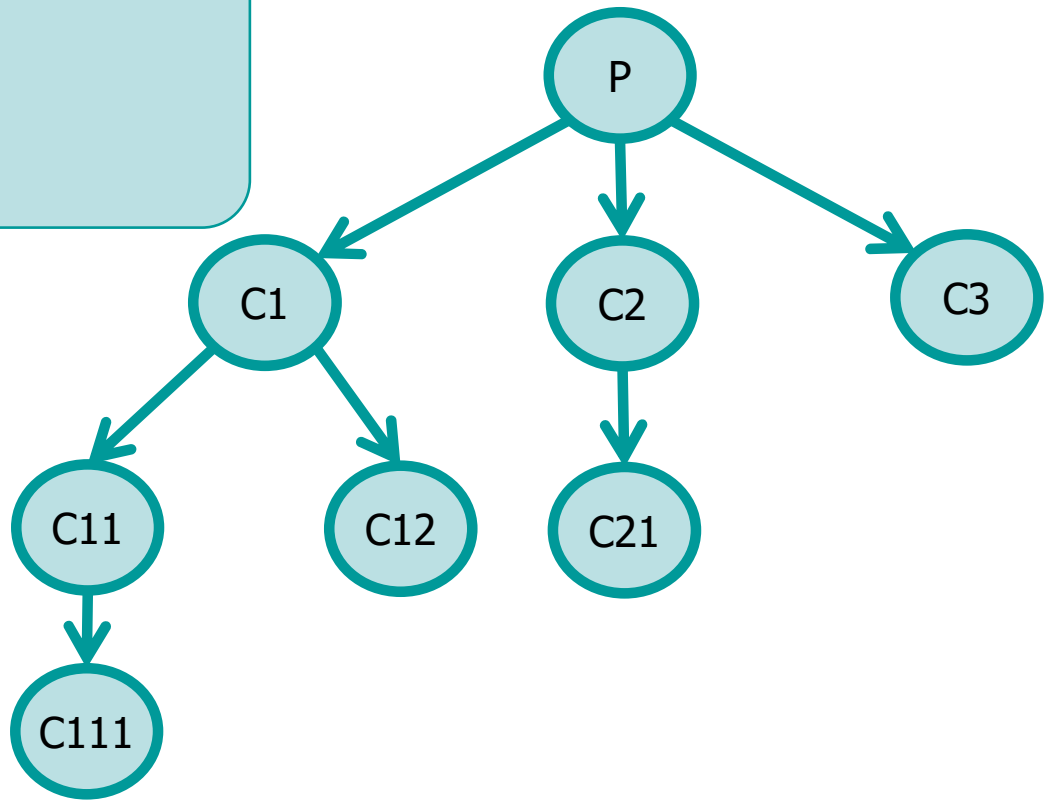
Possible output with n=3

```
Proc 0 (PID=3188)
Proc 1 (PID=3188)
Proc 2 (PID=3188)
Proc 2 (PID=3191)
Proc 1 (PID=3190)
Proc 2 (PID=3190)
Proc 0 (PID=3189)
Proc 1 (PID=3189)
Proc 2 (PID=3189)
Proc 2 (PID=3192)
Proc 2 (PID=3194)
Proc 1 (PID=3193)
Proc 2 (PID=3193)
Proc 2 (PID=3195)
```

Erroneous solution 1

```
int i, n;

scanf ("%d", &n);
for (i=0; i<n; i++) {
  fork();
  printf ("Proc %d (PID=%d)\n",
    i, getpid());
}

exit (0);
```

Process tree with n=3



Generates 7 children processes (in addition to the initial one)

Solution 1 is erroneous

Solution 2

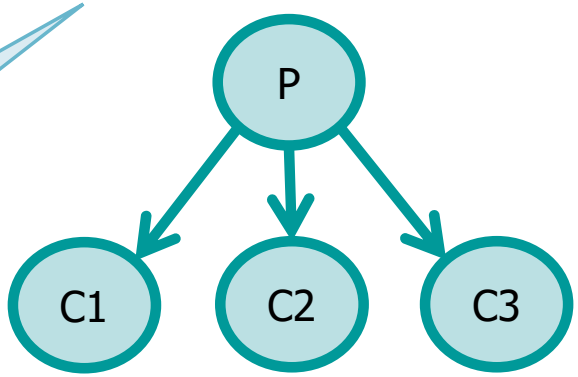
```
int i, n;
…
scanf ("%d", &n);
printf ("Start PID=%d\n",
  getpid());
for(i=0; i<n; i++) {
  if (fork() == 0) {
    printf ("Proc %d (PID=%d)\n",
      i, getpid());
    break;
  }
}
printf ("End PID=%d (PPID=%d)\n",
   getpid(), getppid());

exit(0);
```

Solution 2

```
int i, n;
…
scanf ("%d", &n);
printf ("Start PID=%d\n",
  getpid());
for(i=0; i<n; i++) {
  if (fork() == 0) {
    printf ("Proc %d (PID=%d)\n",
      i, getpid());
    break;
  }
}
printf ("End PID=%d (PPID=%d)\n",
   getpid(), getppid());

exit(0);
```

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

> ./u04s01e06-fork
Start PID=3225
End PID=3225 (PPID=2088)
Proc 2 (PID=3228)
End PID=3228 (PPID=1314)
Proc 1 (PID=3227)
End PID=3227 (PPID=1314)
Proc 0 (PID=3226)
End PID=3226 (PPID=1314)
```

Process tree and output with n=3



Resources

- The child process is a new entry in the Process Table
- The process **resources** can be
  - Completely shared among parent and children processes
    - Same address space
  - Partially shared
    - Address spaces partially overlapped
  - Non shared
    - Separate address spaces

Resources

- 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
  - Etc.

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
    - **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

Example

```
char c, str[10];

c = 'X';
if (fork()) {
  // parent (!=0)
  c = 'F';
  strcpy (str, "parent");
  sleep (5);
} else {
  // child (==0)
  strcpy (str, "child");
}

fprintf(stdout, "PID=%d; PPID=%d; c=%c; str=%s\n",
  getpid(), getppid(), c, str);
```

Output

```
PID=2777; PPID=2776; c=X; str=child
PID=2776; PPID=2446; c=F; str=parent
```

Process termination

- **Five standard** methods for process termination
  - **`return`** from **`main()`**
  - **`exit`** system call
  - **`_exit`** or **`_Exit`**
    - Synonyms defined in ISO C or POSIX
    - Similar effects of **`exit`**, but different management of stdio flushing etc.
  - **`return`** from **`main()`** of the last process thread
  - **`pthread_exit`** from the last process thread

Process termination

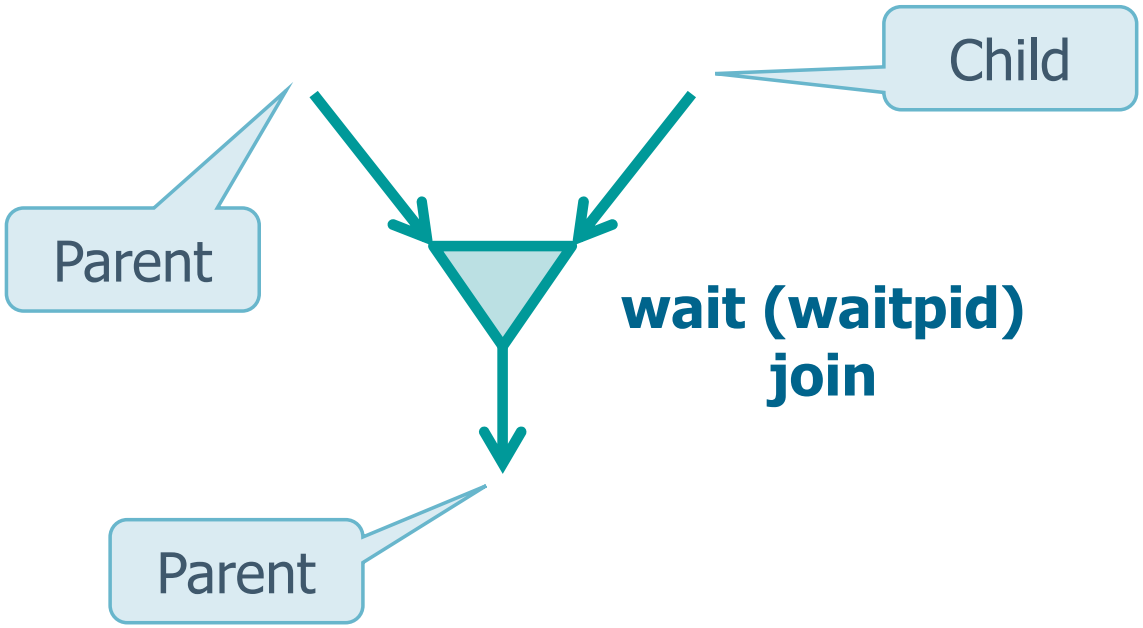
- Three not-normal method for process termination
  - Call of the function **`abort`**
    - Generates the signal **`SIGABORT`**, this is a sub-case of the next because a signal is generated
  - If a termination signal, or a signal not caught is received
  - If the last thread of a process is cancelled

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 system calls
    - **wait**
    - **waitpid**



System call wait ()

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

pid_t wait (int *statLoc);
```

- A call of the system call **wait** by means of a process
  - Returns an error if the calling process has not children
    - A process without children is not supposed to do a wait
    - In this case the returned value is -1

System call wait ()

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

pid_t wait (int *statLoc);
```

- A call of the system call **wait** by means of a process
  - Blocks the calling process if all its children are running (none is already terminated)
    - wait will return as soon as one of its children terminates

System call wait ()

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

pid_t wait (int *statLoc);
```

- A call of the system call **wait** by means of a process
  - 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)
    - When a process ends and the parent does not do a wait, its termination status remains pending
    - Some resources associated with the process remain blocked

System call wait ()

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

pid_t wait (int *statLoc);
```

- The **statLoc** parameter

  Exit status of the child process

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

System call wait ()

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

pid_t wait (int *statLoc);
```

- WIFEXITED(statLoc) is true if wait terminates correctly. In this case WEXITSTATUS(statLoc) catches the 8 LSBs of the parameter passed to a exit (_exit or _Exit)

- Returned values
  - The PID of a terminated child on success
  - -1 on error

Example

```
...
pid_t pid, childPid;
int statVal;
...
pid = fork();
if (pid==0) {
   // Child
   sleep (5);
   exit (6);
} else {
...
```

Example

```
...
  // Parent
  childPid = wait (&statVal);
  printf("Child terminated: PID = %d\n", childPid);
  if (WIFEXITED(statVal))
    // WIFEXITED: True if correctly terminated
    // WEXITSTATUS: Takes the 8 returned LSBs (exit)
    printf ("Exit value: %d\n",
      WEXITSTATUS (statVal));
    else
      printf ("Abnormal termination\n");
  }
  exit(25);
}
...
```

Prints 6

echo $?
(in a shell) prints 25

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.

Zombie 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 ..."

Orphan processes

- If the parent terminates before executing the **wait**, the child process
  - Becomes an **orphan**
  - The orphan processes, in order not to remain in this state, are inherited by the **init** process (the one with PID=1) or by a **user custom init** process
  - Orphan processes and processes inherithed by init will no longer become zombie processes

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);
```

- The parameter **pid** allows waiting for
  - Any child (waitpid==wait) (pid = -1)
  - The child whose PID=pid (pid > 0)
  - Any child whose GID is equal to that of the calling process (pid = 0)
  - Any child whose GID=abs(pid) (pid < -1)

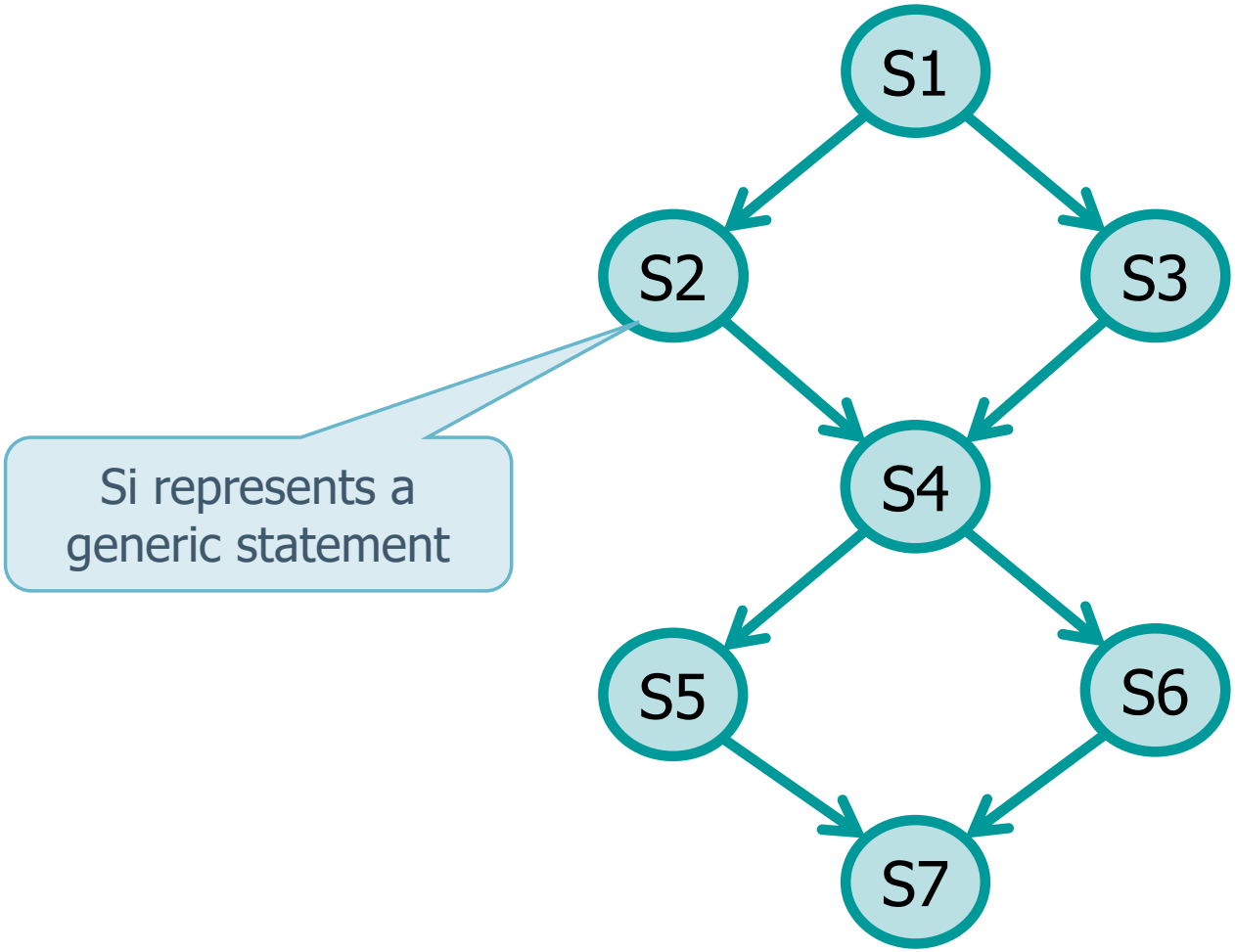
System call waitpid ( )

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

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

- The **options** parameter allow additional controls
  - Default is 0, or is a bitwise OR of constants
    - **WNOHANG**, if the child specified by PID is running, the caller does not block (**not blocking** version of **wait**)
    - **WCONTINUED** and **WUNTRACED** allow to know the status of a child in particular conditions

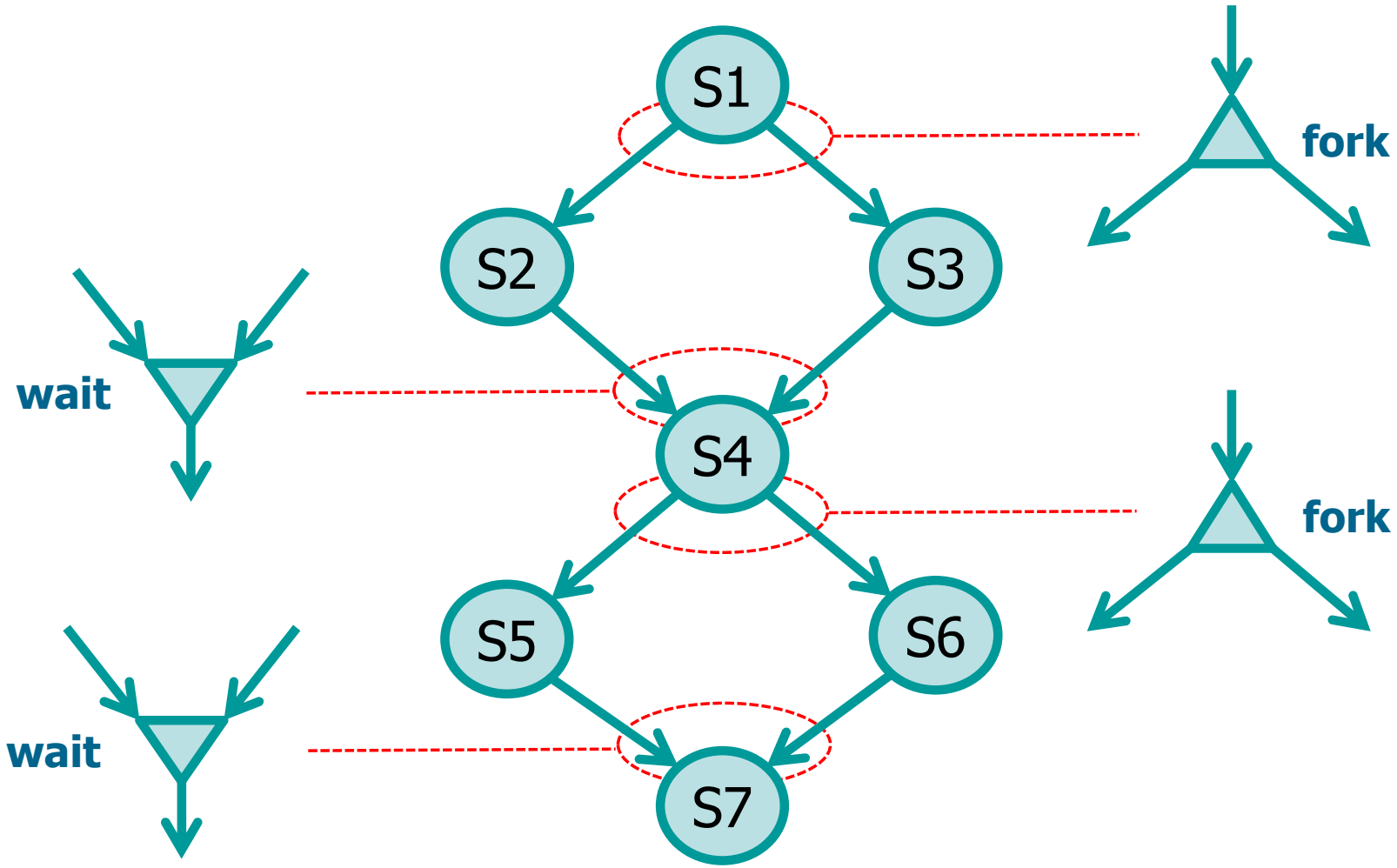
Exercise

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



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");
  pid = fork();
  if (pid == 0) {
    //sleep (2);
    printf ("S3\n");
    exit (0);
  } else {
    //sleep (2);
    printf ("S2\n");
    wait ((int *) 0);
  }
```

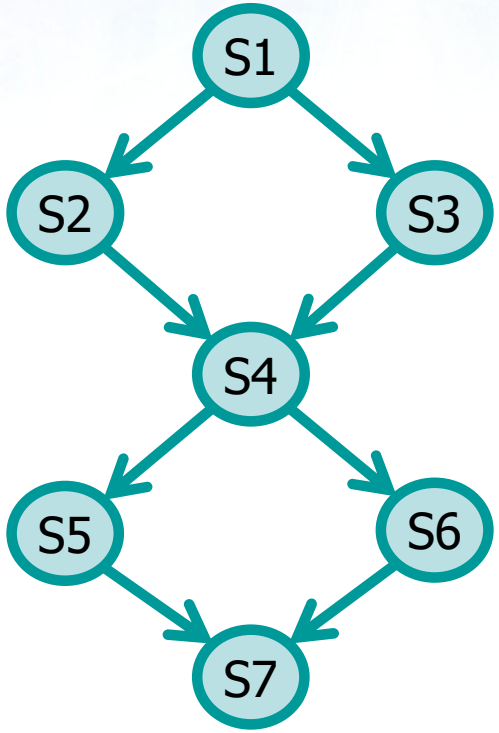
Debug ...

Child

Parent

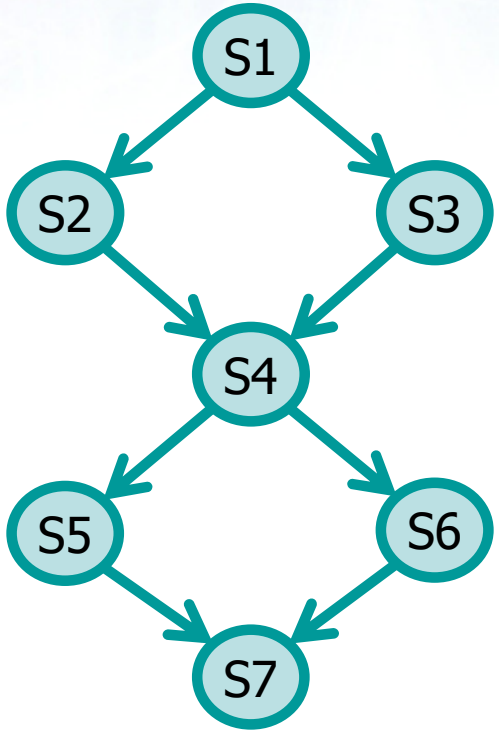
Returned PID ignored

Termination state ignored



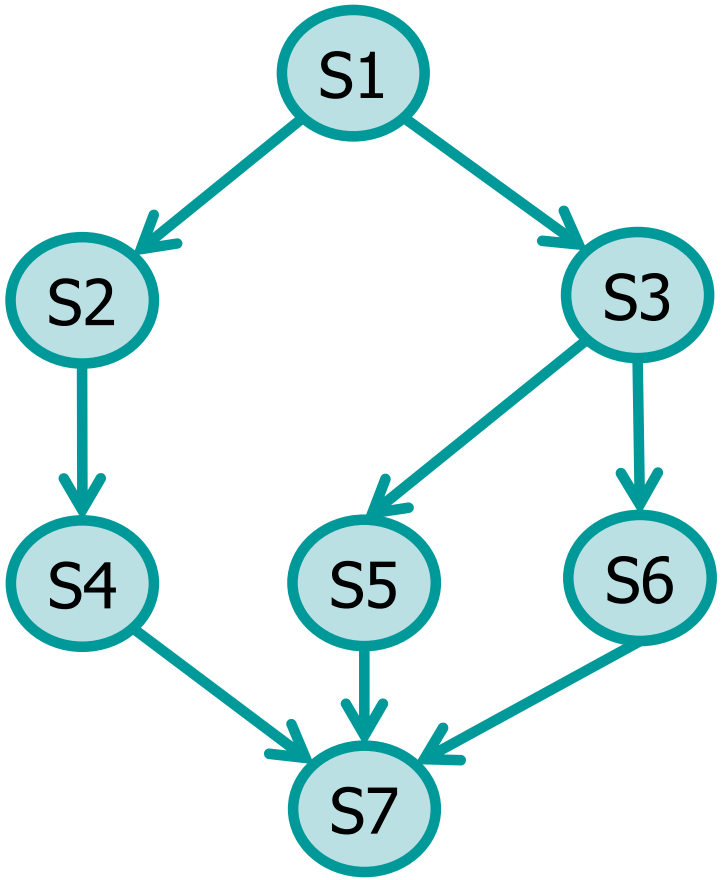
Solution

```
  printf ("S4\n");
  pid = fork();
  if (pid == 0) {
    //sleep (2);
    printf ("S6\n");
    exit (0);
  } else {
    //sleep (2);
    printf ("S5\n");
    wait ((int *) 0);
  }
  printf ("S7\n");
  return (0);
}
```



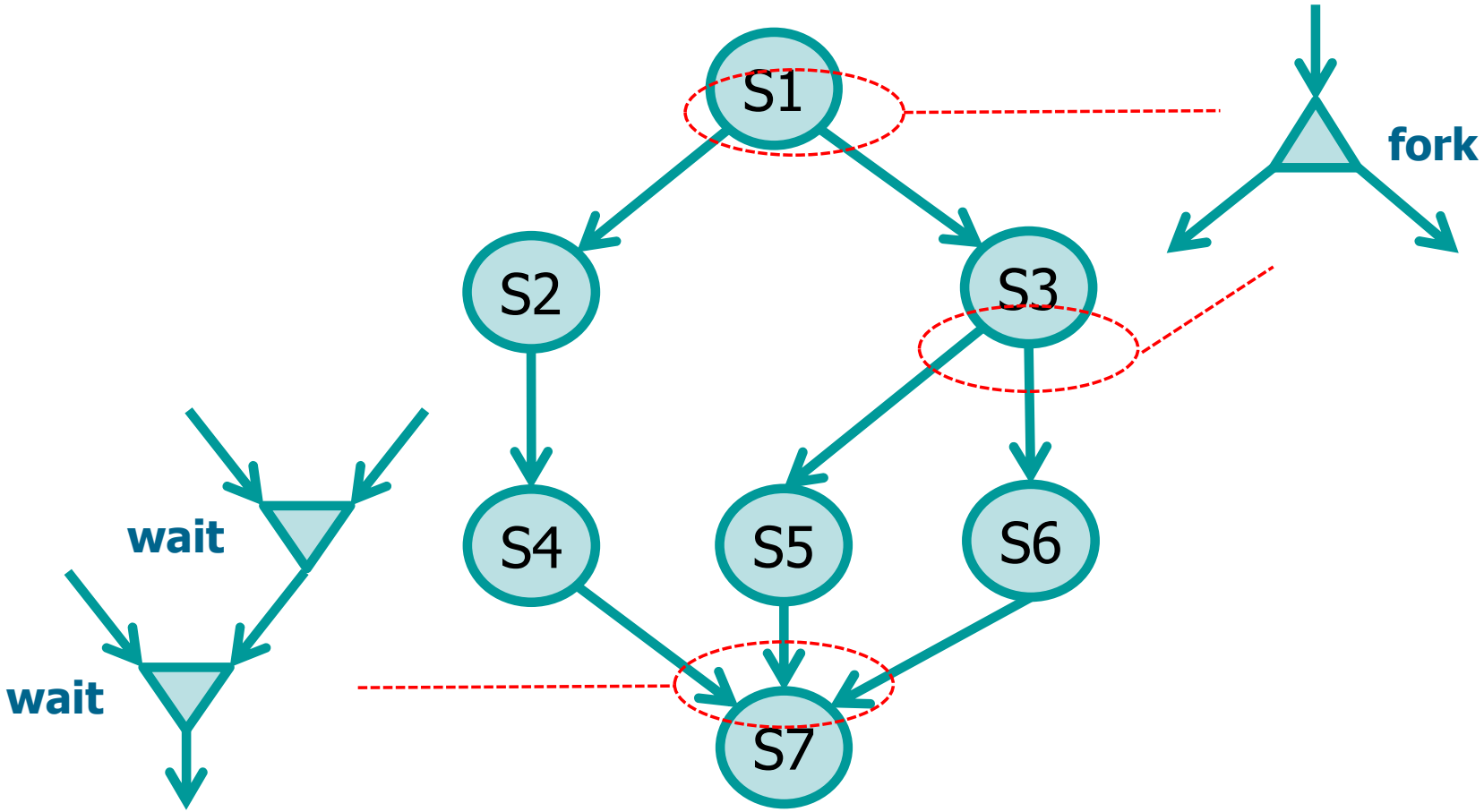
Exercise

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



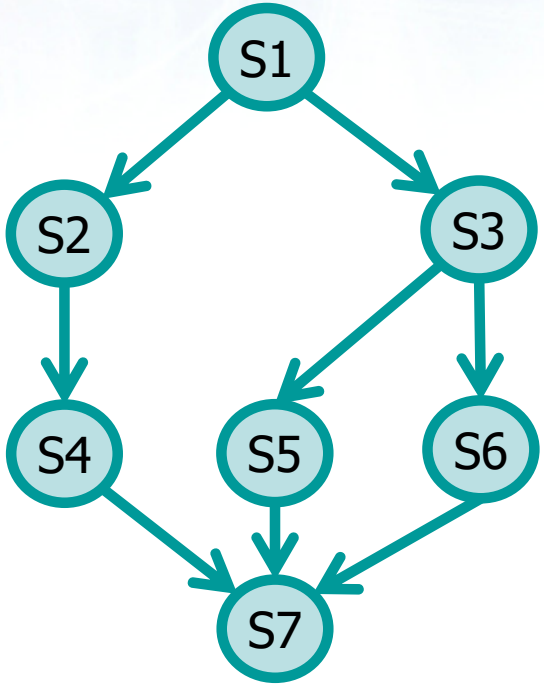
Exercise

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



Solution

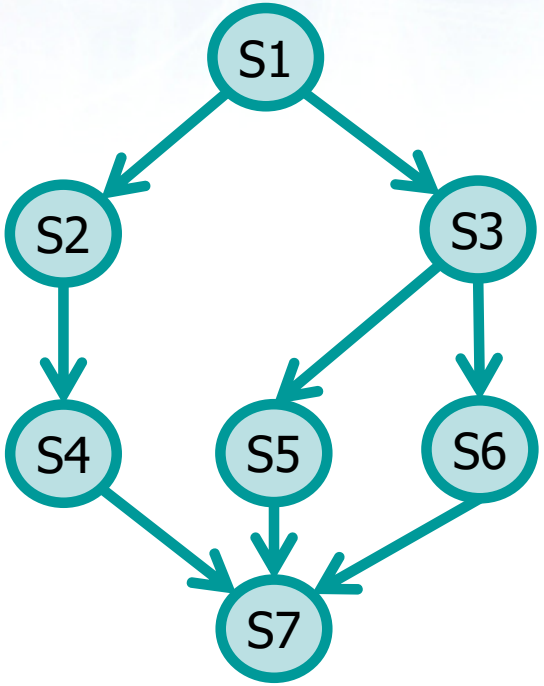
```
int main () {
  pid_t pid;
  printf ("S1\n");
  if ( (pid = fork())==-1 )
    err_sys( "can't fork" );
  if ( pid == 0 ){
    P356();
  } else {
    printf ("S2\n");
    printf ("S4\n");
    while (wait((int *)0)!= pid);
    printf ("S7\n");
    exit (0);
  }
  return (1);
}
```



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

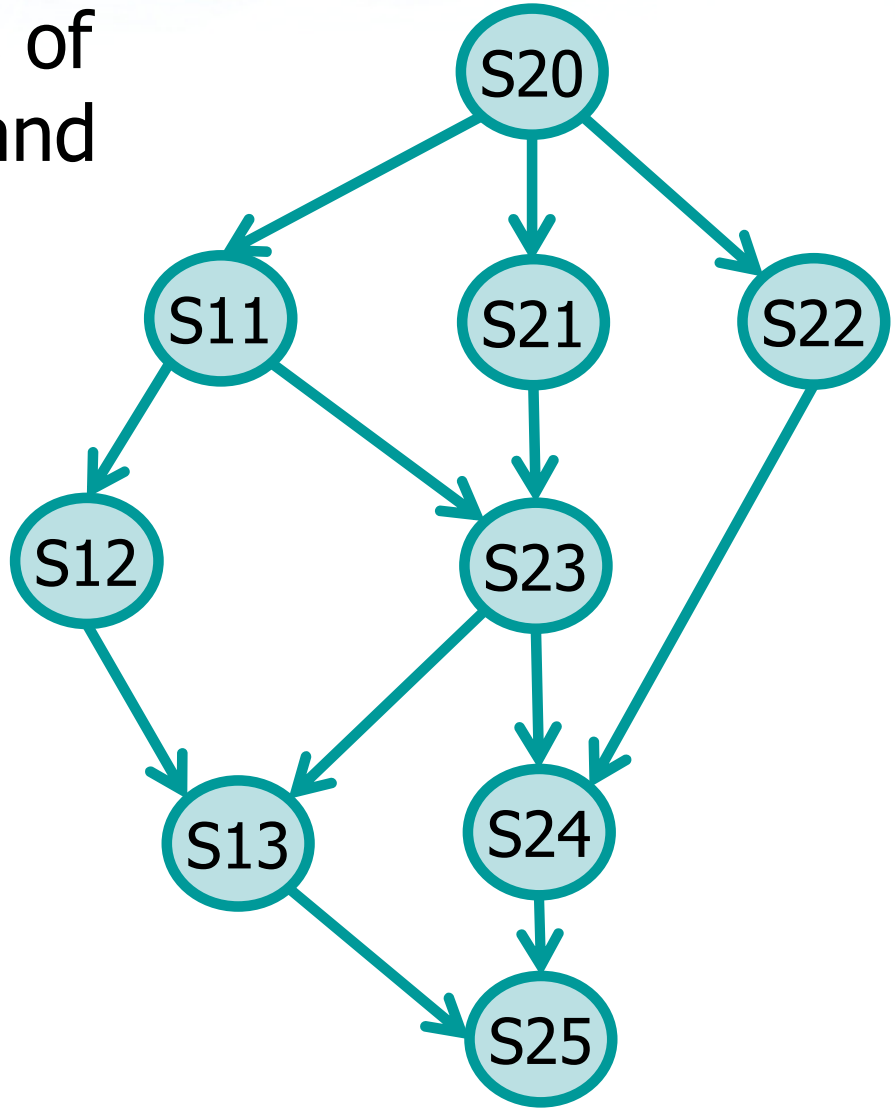
```
int main(int argc, char *argv[]) {
  int i, n, *vet;
  int retValue;
  pid_t pid;
  n = atoi (argv[1]);
  vet = (int *) malloc (n * sizeof (int));
  if (vet==NULL) {
    fprintf (stderr, "Allocation Error.\n");
    exit (1);
  }
  fprintf (stdout, "Input:\n");
  for (i=0; i<n; i++) {
    fprintf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

Solution

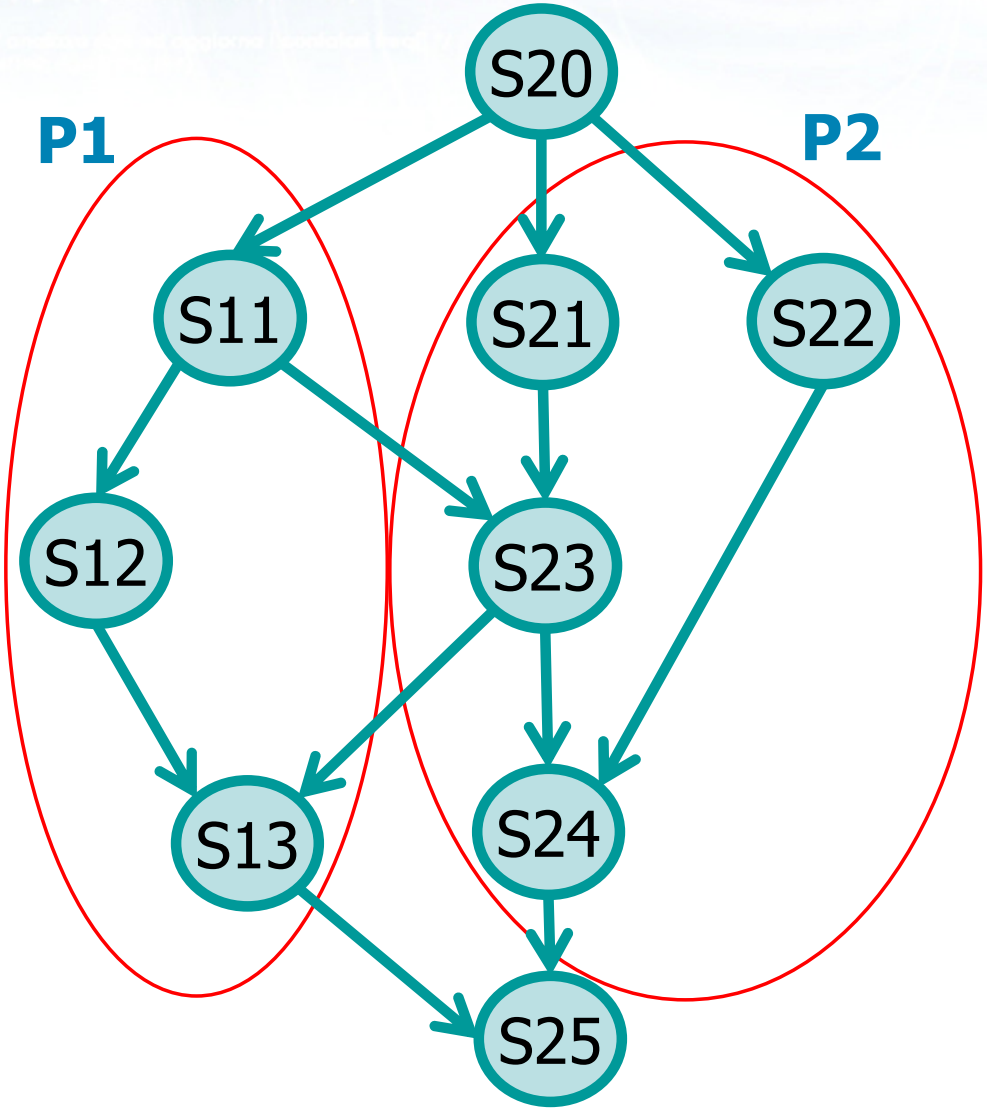
```
  fprintf (stdout, "Output:\n");
  for (i=0; i<n-1; i++) {
    pid = fork();
    if (pid>0) {
      pid = wait (&retValue);
      break;
    }
    fprintf (stdout, "Run PID=%d\n", getpid());
  }

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

  exit (0);
}
```

Exercise

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

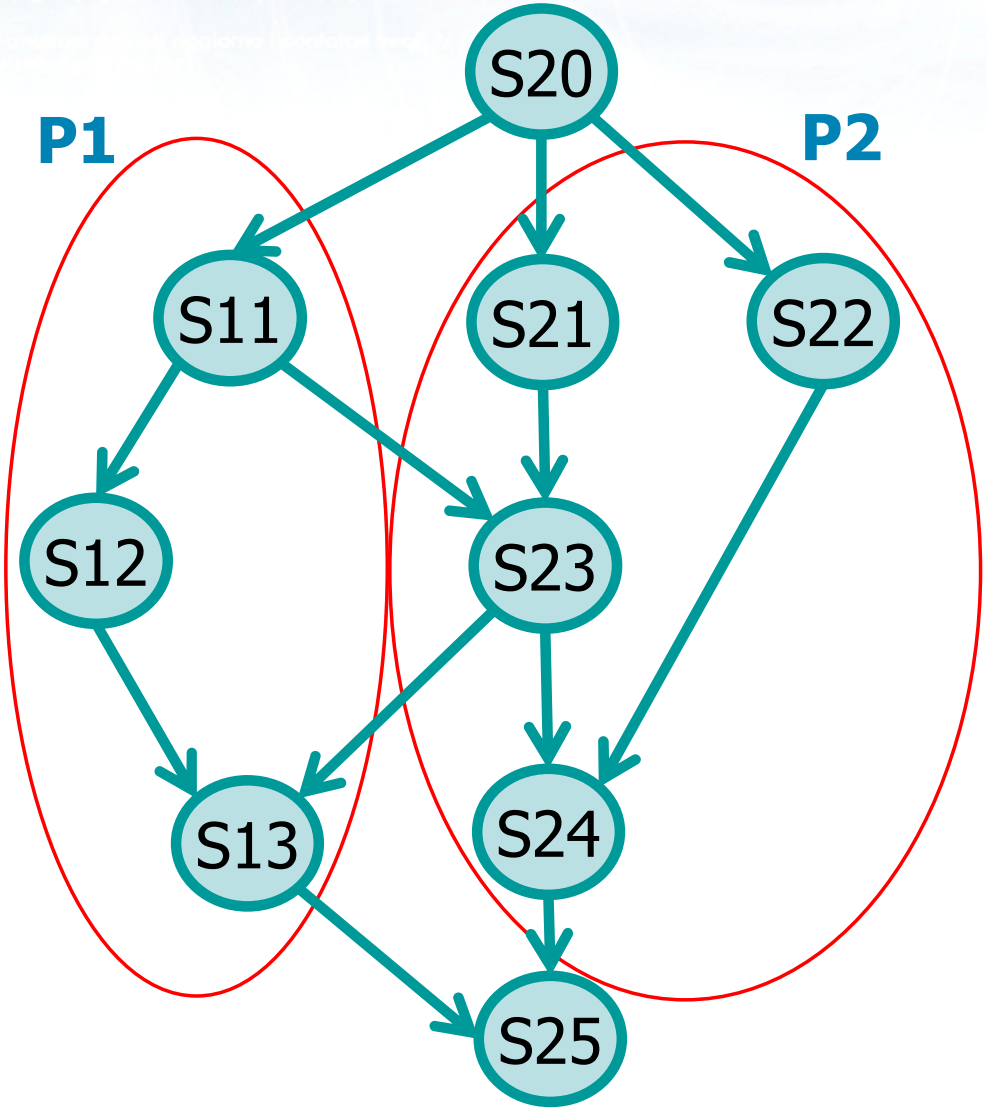


Solution



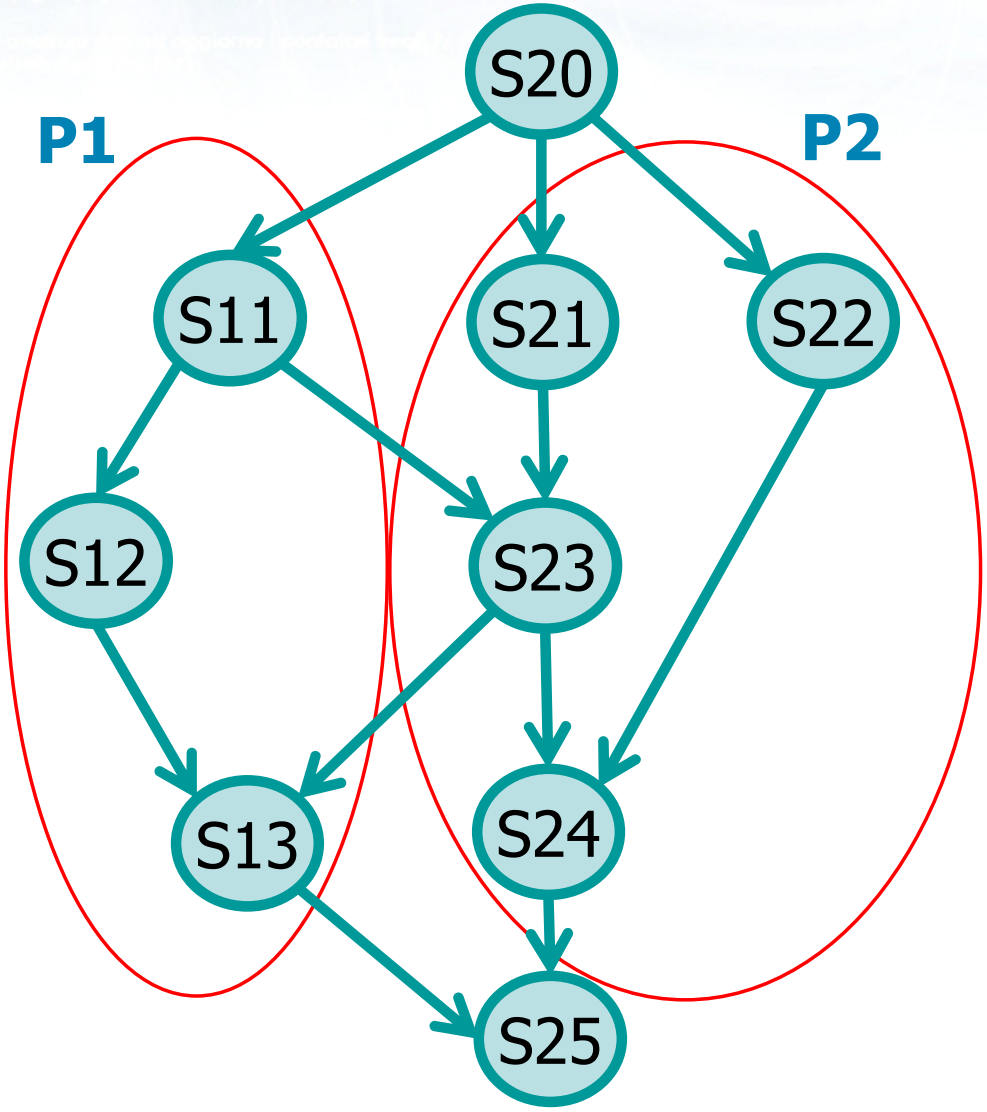

```
main () {
   S20 ();
   pid = fork ();
   if (pid>0) {
      P1 ();
      wait ((int *)0);
   } else {
      P2 ();
   }
   S25 ();
   return;
}
```

Solution




```
P1() {
   S11 ();
   pid = fork ();
   if (pid>0) {
      S12 ();
      wait((int *)0);
   } else {
      ??? To P2 ???;
      exit(0);
   }
   S13 ();
}
```

Solution

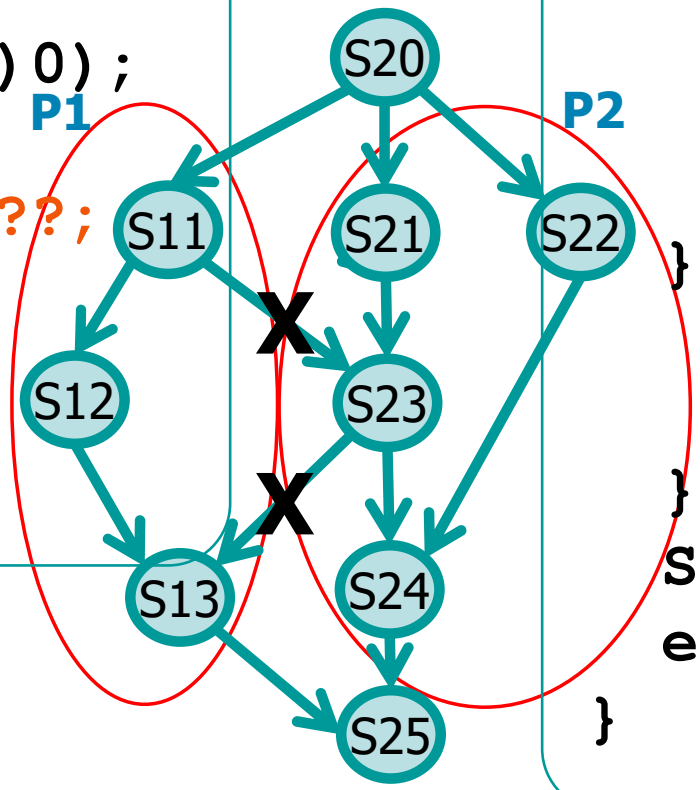



```
P2() {
   pid = fork ();
   if (pid>0) {
      S21 ();
      ??? From S1 ???;
      S23 ();
      wait((int *)0);
   } else {
      S22 ();
      exit(0);
   }
   S24 ();
   exit (0);
}
```

Solution

Unfeasible graph

```
P1() {
   S11 ();
   pid = fork ();
   if (pid>0) {
      S12 ();
      wait((int *)0);
   } else {
      ??? To P2 ???;
      exit(0);
   }
   S13 ();
}
```




```
P2() {
   pid = fork ();
   if (pid>0) {
      S21 ();
      ??? From S1 ???;
      S23 ();
      wait((int *)0);
   } else {
      S22 ();
      exit(0);
   }
   S24 ();
   exit (0);
}
```