

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
```

```
{
    int freq[MAXPAROLA]; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE * f;
```

```
    for(i=0; i<MAXPAROLA; i++)
        freq[i]=0;
```

```
    if(argc != 2)
```

```
    {
        fprintf(stderr, "ERRORE: serve un parametro con il nome del file\n");
        exit(1);
    }
```

```
    f = fopen(argv[1], "rt");
    if(f==NULL)
```

```
    {
        fprintf(stderr, "ERRORE: impossibile aprire il file %s\n", argv[1]);
        exit(1);
    }
```

```
    while( fgets( riga, MAXRIGA, f ) != NULL )
```

Processes

Processes - Linux

Stefano Quer - Pietro Laface

Dipartimento di Automatica e Informatica

Politecnico di Torino

Processes

❖ Several processes are started at bootstrap

➤ Automatic

- Daemon Processes *eg admin*
- E-mail applications
- Various control activities, virus scan and others
- ...

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

➤ On user request

- Line printer management
- WEB server
- ...

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 o 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

- ❖ The **first process**, PID=0, is a system process
 - The **swapper**, which is responsible for **memory** management and **process** scheduling
- ❖ The second process, PID=1, is **init** a daemon executed at user level, but with super-user privileges
 - Becomes the parent of each **orphan process**s, i.e., of a child of a parent process already terminated

Process identification

```
#include <unistd.h>

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

❖ Other identifiers related to a process

```
uid_t getuid();    // Get the real user ID
gid_t geteuid();   // Get the effective ID
```

superuser at exec time

❖ **getppid** returns the identifier of the parent process

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

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>

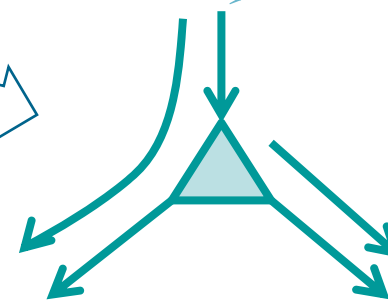
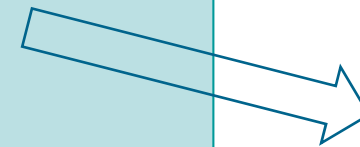
pid_t fork (void);
```

❖ 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

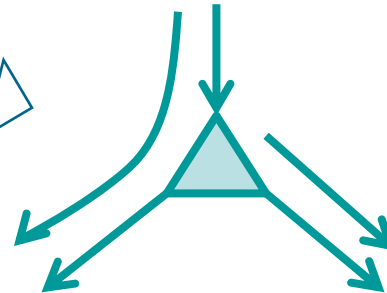
Child
continues execution
pid = 0

Process creation

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



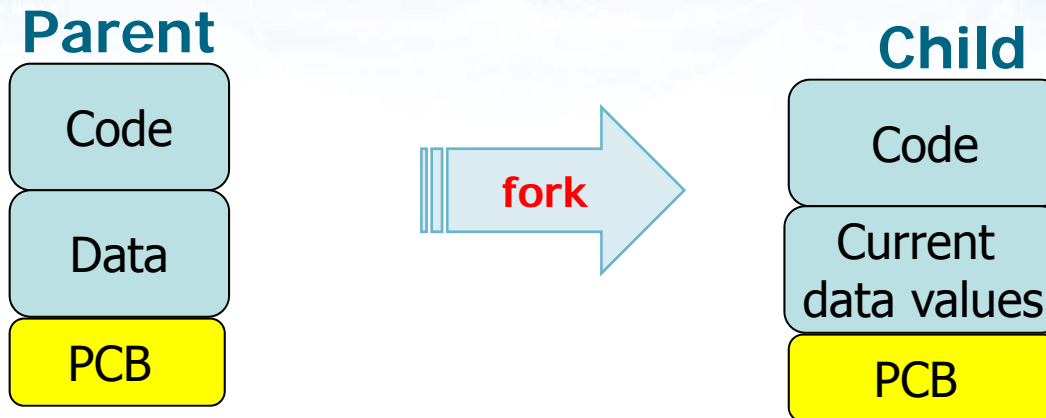
Process
(parent)



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 *cp the stack pointer*
- 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 viceversa
- ❖ Output in both cases the **Process Identifier** of the terminating process and the Process Identifier of its parent.
 - Who is the parent's parent? `bash`
 - Who is the child's parent if the parent terminates before the child?
`pid 1`

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 (tF);
```

```
    printf ("Parent: PIDreturned=%d ", pid);
```

```
    printf ("PID=%d; My parent PID=%d\n",  
            getpid(), getppid());
```

```
}
```

```
tC = atoi (argv[1]);
```

```
tF = 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
```

zombi e

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
```

opha

init process PID

Exercise

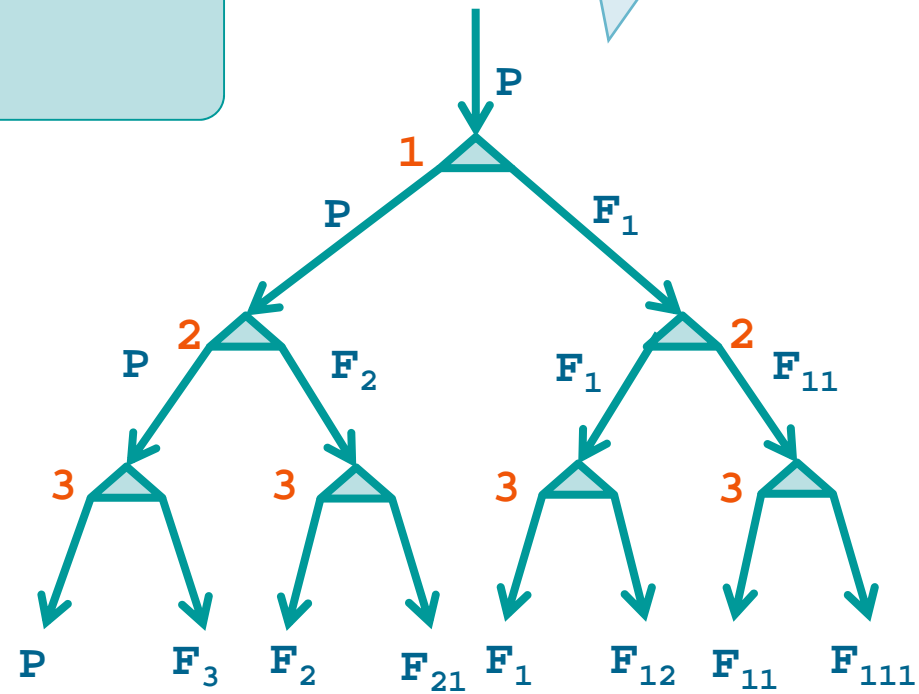
- ❖ Given the following program, draw its
 - Control Flow Graph, CFG
 - Process generation graph

```
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  
}
```

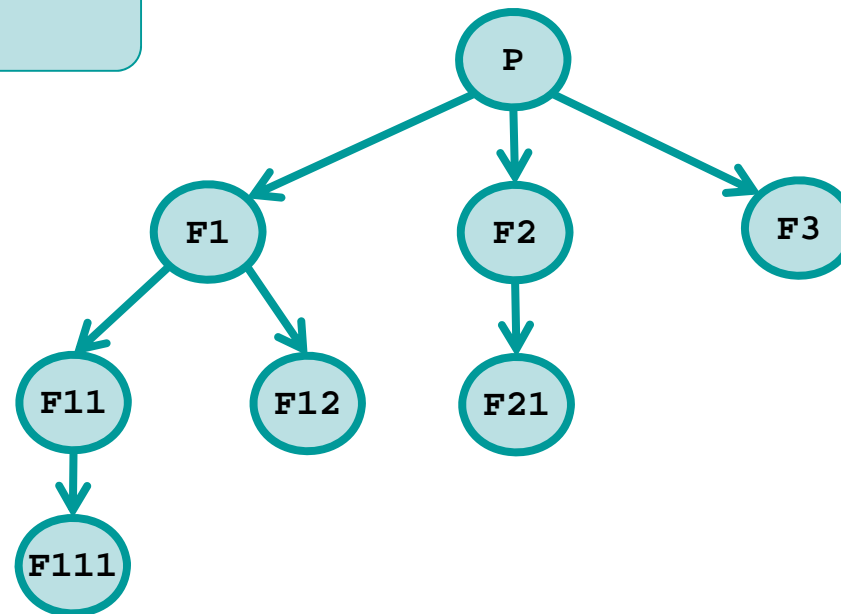
Control Flow Graph
(CFG)



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 */
```

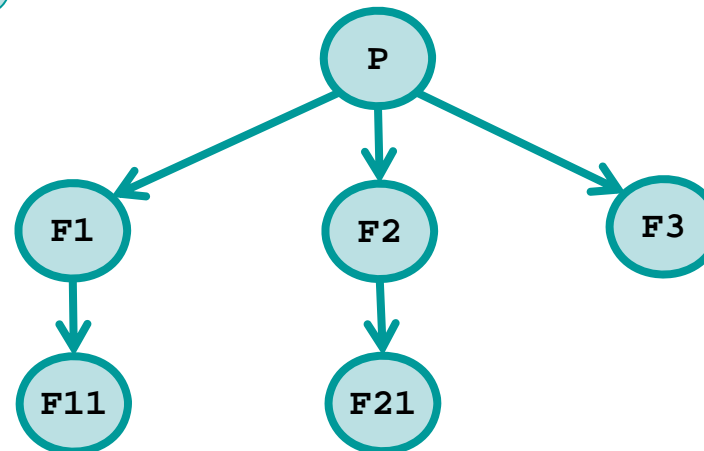
Solution

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

Control Flow Graph
(CFG)

...

Process generation tree



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 */
```

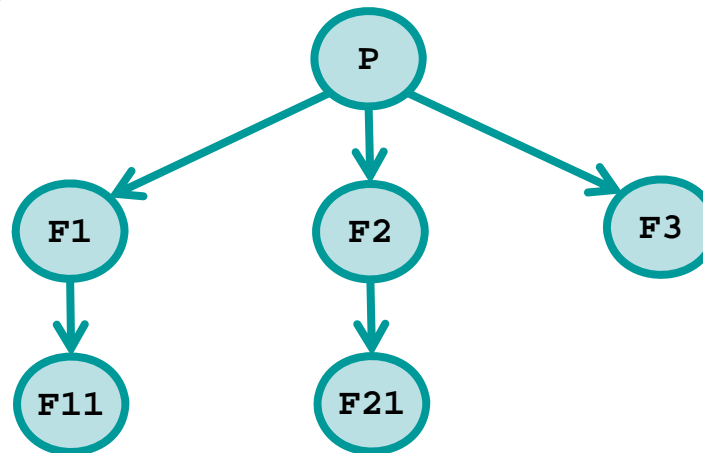
Solution

```
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 */
    }
}
```

Solution

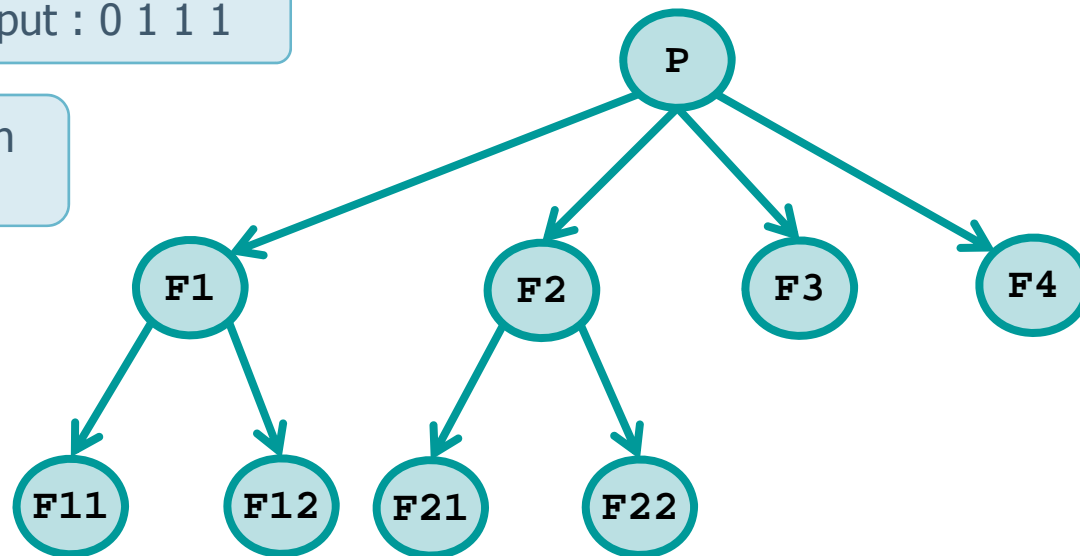
```
for (i=0; i<2; i++) {  
    printf("i: %d \n", i);  
    if (fork()) // 1  
        fork(); // 2  
}
```

Possible output : 0 1 1 1

Control Flow Graph
(CFG)

...

Process generation tree



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);
```

Possible output with
n=3

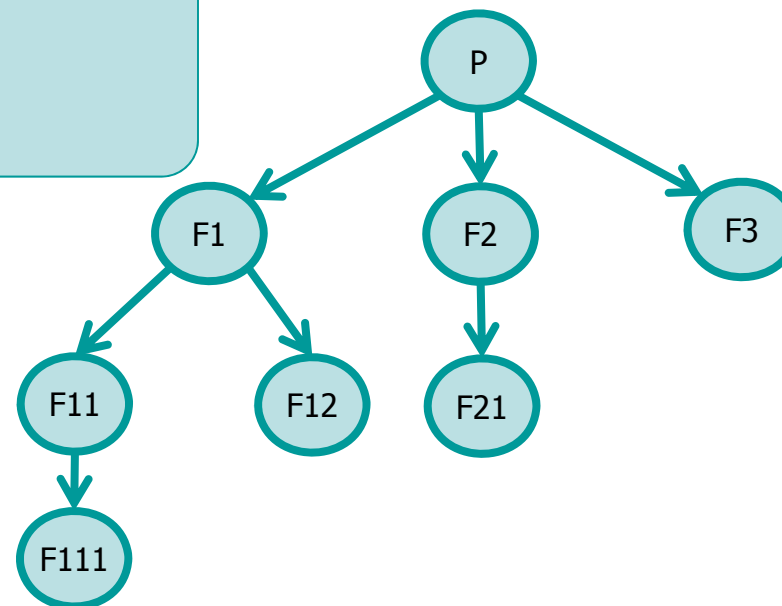
```
Proc 0 (PID=3188)
Proc 1 (PID=3188)
Proc 2 (PID=3188)
Proc 2 (PID=3191)
Proc 1 (PID=3190)
Proc 2 (PID=3190)
Proc 0 (PID=3189)
Proc 1 (PID=3189)
Proc 2 (PID=3189)
Proc 2 (PID=3192)
Proc 2 (PID=3194)
Proc 1 (PID=3193)
Proc 2 (PID=3193)
Proc 2 (PID=3195)
```

Erroneous solution 2

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

Generates 7 children processes

Process tree with
n=3



子进程进入下一个for

Solution 3

```
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 3

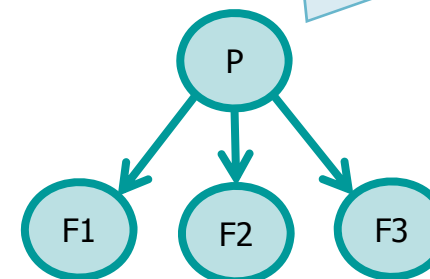
```
int i, n;
...
scanf ("%d", &n);
printf ("Start PID=%d\n",
    getpid());
for(i=0; i<K; i++) {
    if (fork() == 0) {
        printf ("Proc %d (PID=%d)\n",
            i, getpid());
        break;
    }
}
printf ("End PID=%d (PPID=%d)\n",
    getpid(), getppid());

exit(0);
```

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

> ./u04s01e06-fork
Star 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 with
n=3



把子进程break掉

公用一个父进程

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
- UID, GID, etc.
- The open file descriptors (File Description Table)
 - In particular, `stdin`, `stdout`, and `stderr`
 - Concurrent I/O operation implies producing interlaced I/O
- The root and the working directory
- System resources and their utilization limits
- Signal Table
- Etc.

❖ 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 the OS
 - 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);
```

子进程在fork时copy char str【】

Output

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

char str 都被分离

Process termination

❖ Five standard methods for process termination

- **return** from main *main return 0; because bash 0 means true*
- **exit** system call *for termination in function*
- **_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
 - If a termination signal, or a signal not caught is received
 - **abort**
 - Generates the signal `SIGABORT`
 - If the last thread of a process is cancelled

System call wait () and waitpid ()

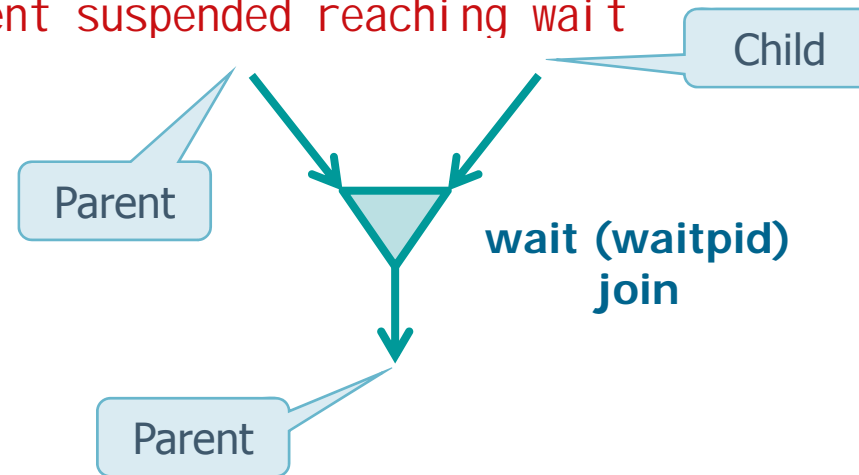
- ❖ When a process terminates (normally or not) one of its child is terminated
 - The kernel sends a signal (**SIGCHLD**) to its parent
 - For the parent this is an asynchronous event
 - The parent process may
 - Manage the child termination (and/or the signal)
 - Ignore the event (default)

System call `wait()` and `waitpid()`

❖ A parent process can manage child termination

➤ By means of system calls

- **wait** parent suspended reaching wait
- **waitpid**



➤ Using a signal handler for signal **SIGCHLD**

- This approach will be introduced in the section devoted to **signals**

System call wait ()

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

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

- ❖ 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
- ❖ Returns an error if the calling process has not children.

System call wait ()

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

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

wait 返回的另一个参数，子进程的状态信息

❖ Returns the **exit status** of a child

wait(0)

- If a child process terminates, and the parent process does not call wait, the child exit status remains pending in the **kernel Process Control Block**

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
 - Recovered using macros in **<sys/wait.h>** (**WIFEXITED**, **WIFSIGNALED**, etc.)

System call wait ()

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

pid_t wait (int *statLoc);
```

❖ Returns

- The PID of a terminated child

Example

```
...
pid_t pid, childPid;
int statVal;
...
pid = fork();
switch (pid) {
    case -1:
        printf ("Fork failed\n"); exit(1);
    case 0:
        // Child
        sleep (10);
    default:
        // Parent
        sleep (05);
}
...
```

Example

```
...
if (pid != 0) {
    // Padre
    childPid = wait (&statVal);    blocking for 5 sec
    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(code);
}
...
```

Zombie processes

- ❖ A child process terminated, whose parent is running, but has not yet 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

- ❖ If 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.

System call `waitpid()`

- ❖ 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. By default, **`waitpid()`** waits only for terminated children.

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 if >0 (pid > 0)
- Any child whose GID is equal to that of the calling process (pid = 0)
- Any child whose GID=abs(pid) (pid < -1)

typically

System call waitpid ()

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

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

❖ The **options** parameter allow additional controls

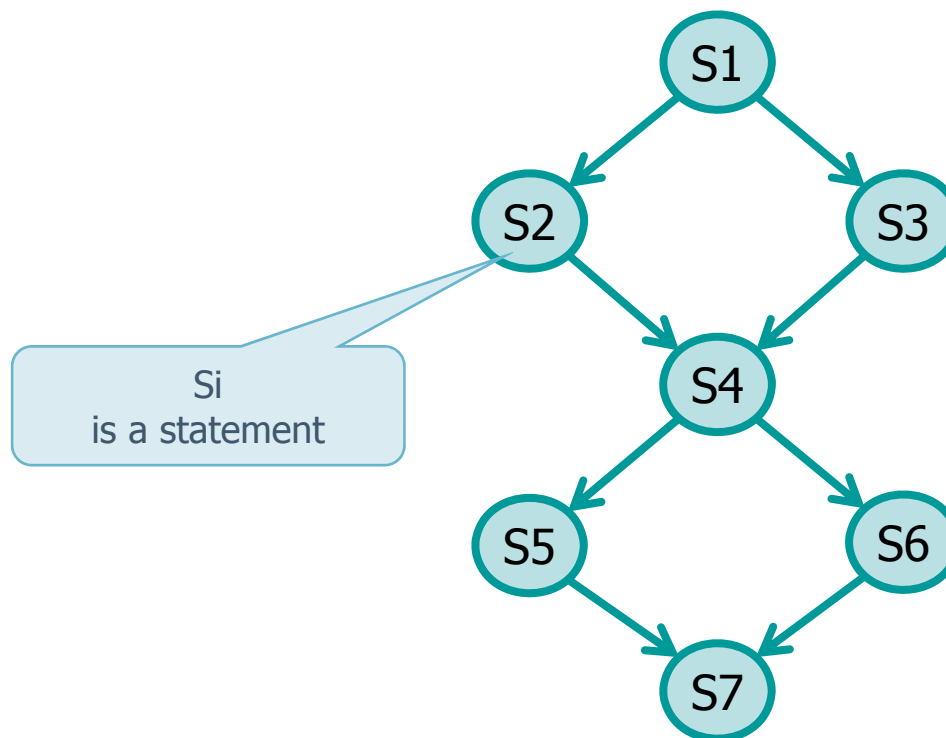
➤ **Default is 0,** or is a bitwise OR of constants

- **WNOHANG**, if the child specified by PID is running, the caller does not block (not blocking version of **wait**)
- **WCONTINUED**, etc.

Exercise

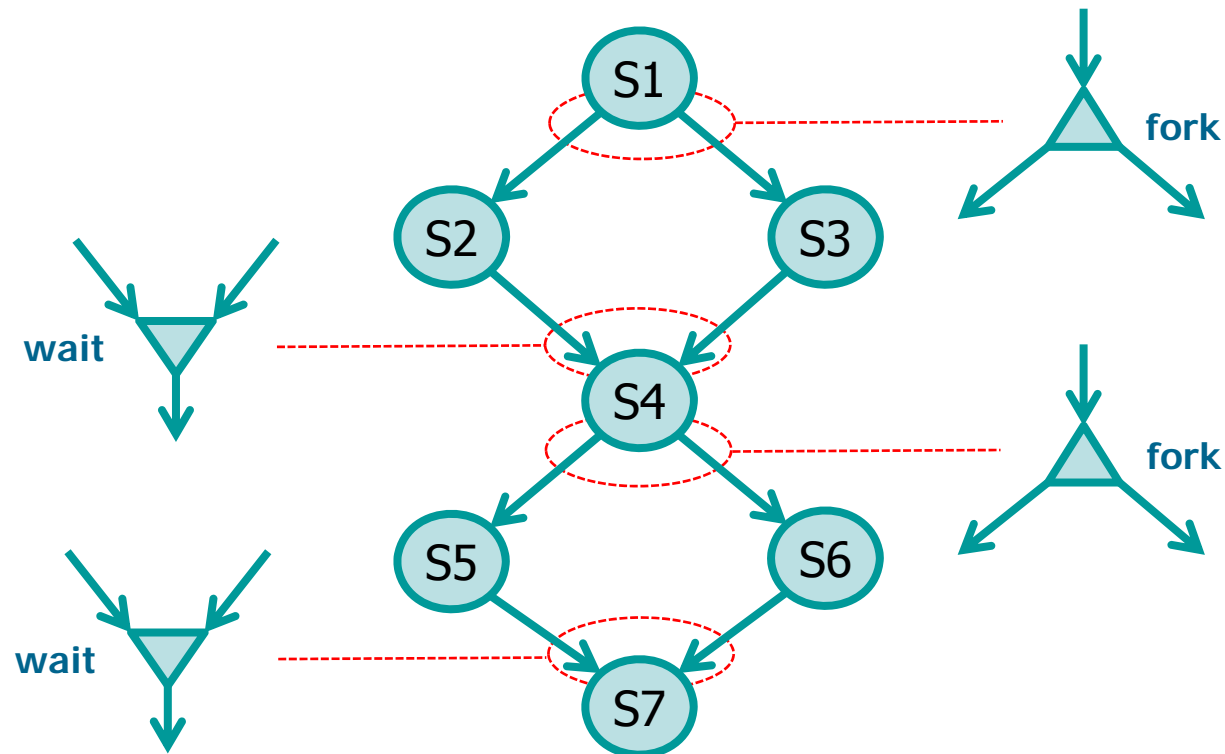
precedence graph

- ❖ 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**



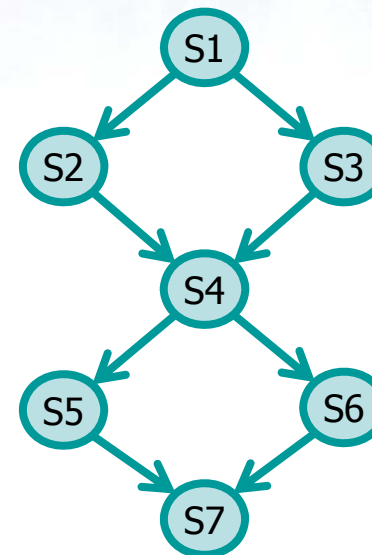
Solution

```
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 ...

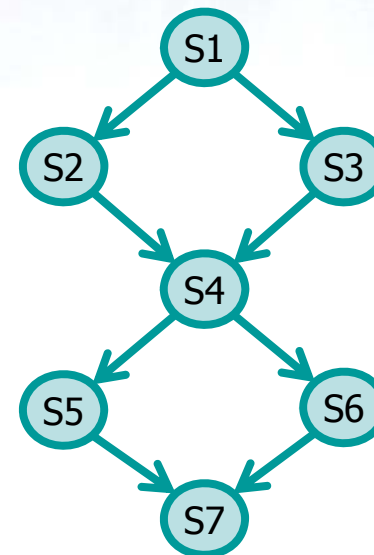
Child

Parent

Termination state
ignored

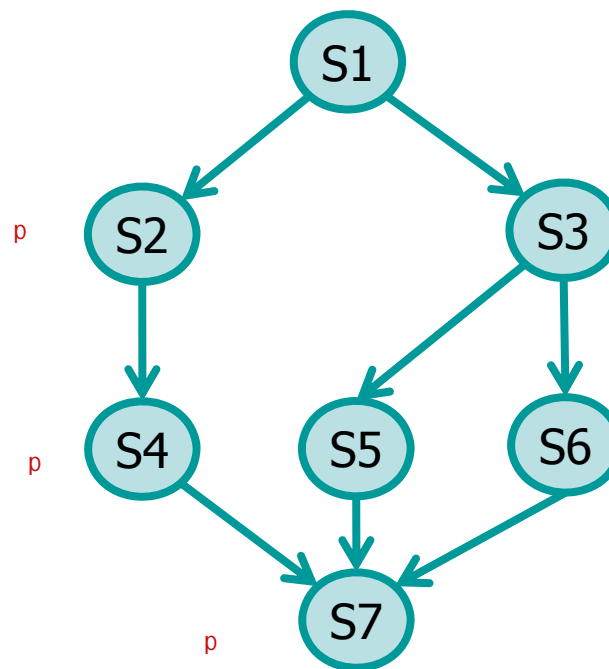
Solution

```
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);  
}
```



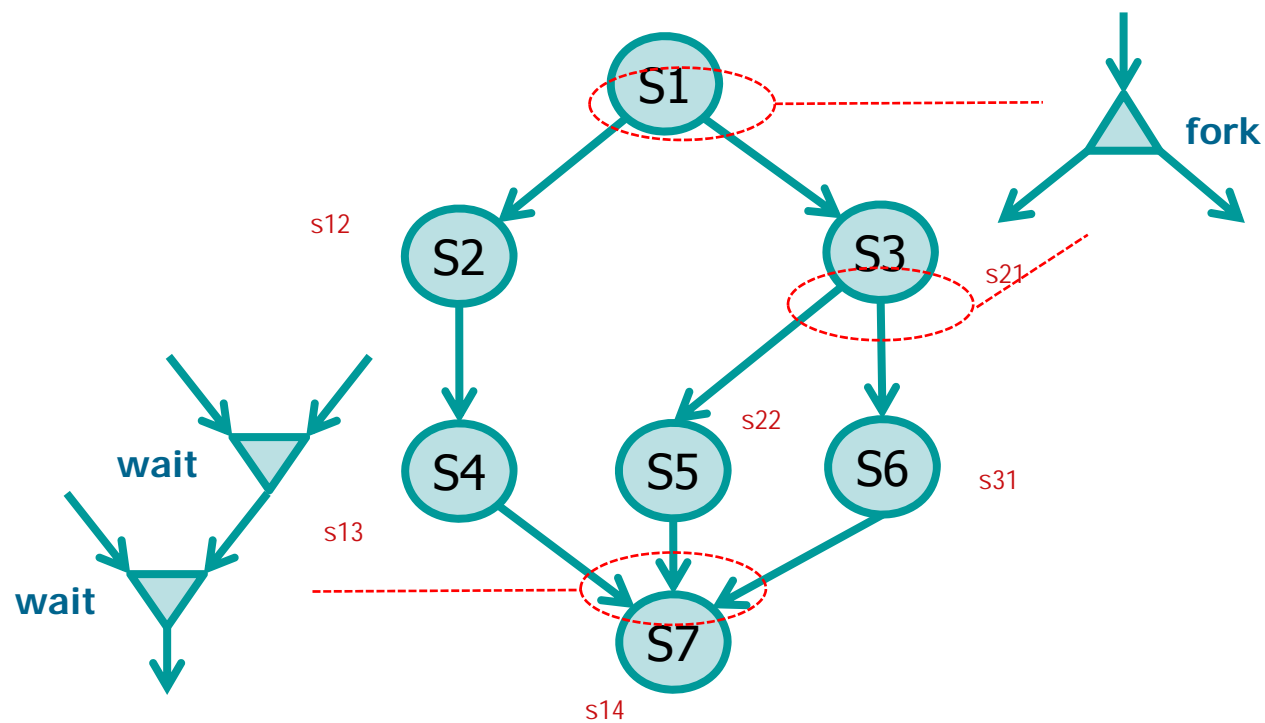
Exercise

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



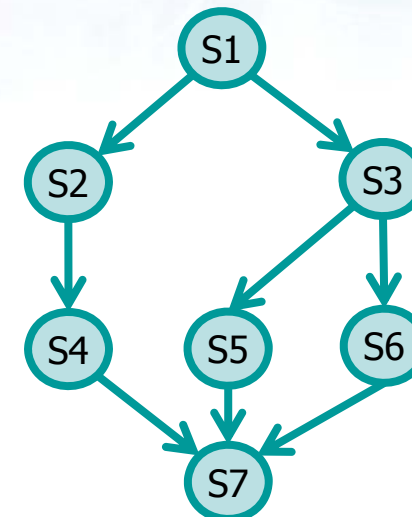
Exercise

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



Solution

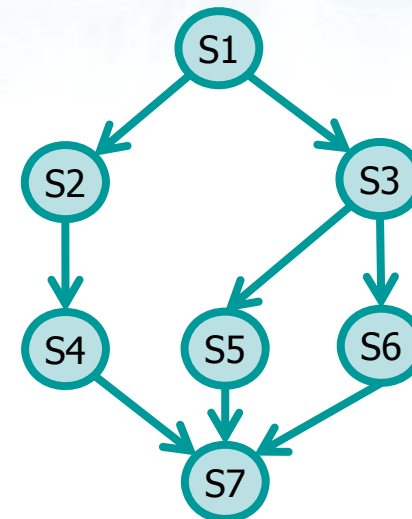
```
int main () {  
    pid_t pid;  
    printf ("S1\n");  
    if ( (pid = fork()) == -1 )  
        err_sys( "can't fork" );  
    if ( pid == 0 ){  
        P356();  
    } else {  
        printf ("S2\n");  
        printf ("S4\n");  
        while (wait((int *)0) != pid);  
        printf ("S7\n");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



wait until received

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

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

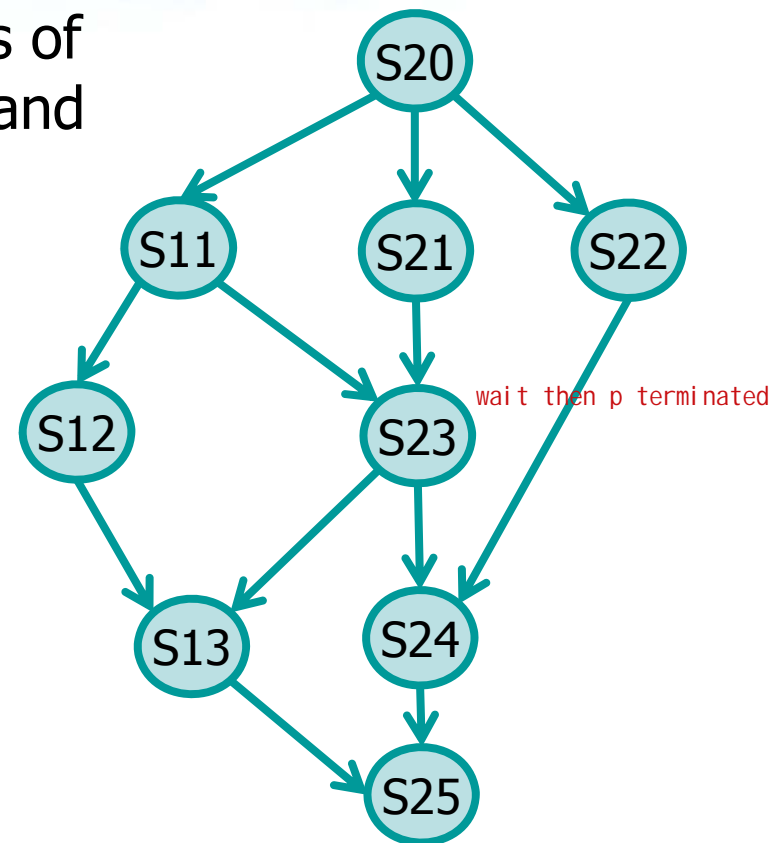
```
fprintf (stdout, "Output:\n");
for (i=0; i<n-1; i++) {
    pid = fork();
    if (pid>0) {
        pid = wait (&retValue);
        break;
    }
    fprintf (stdout, "Run PID=%d\n", getpid());
}

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

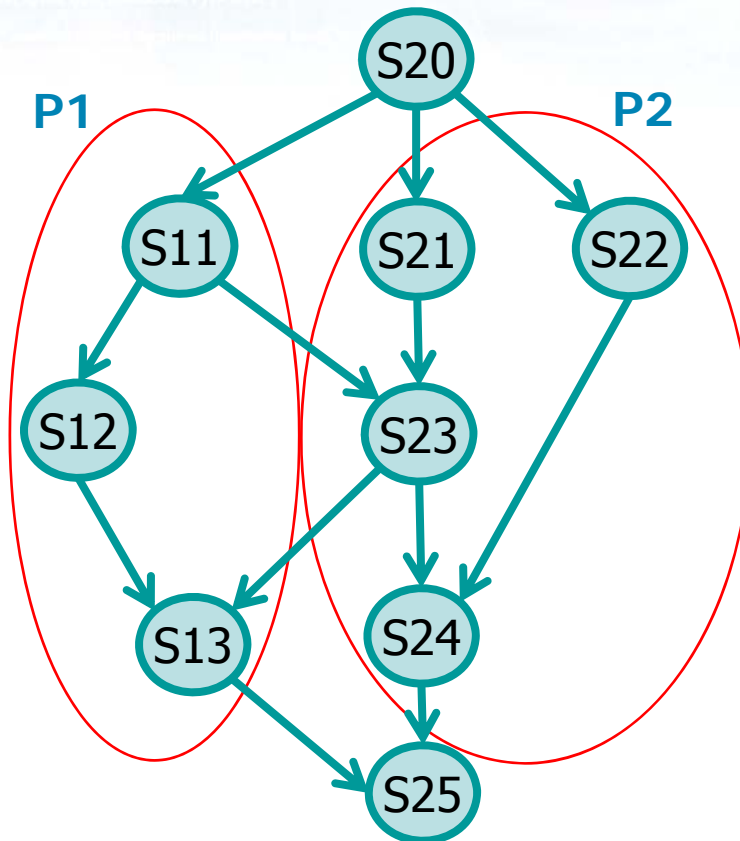
exit (0);
}
```

Exercise

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

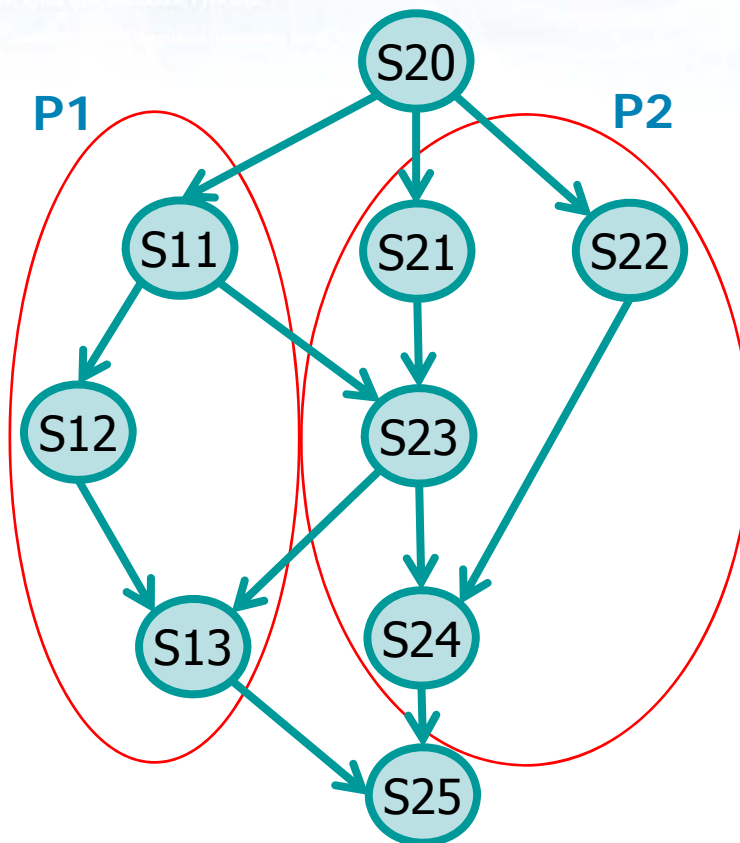


Solution



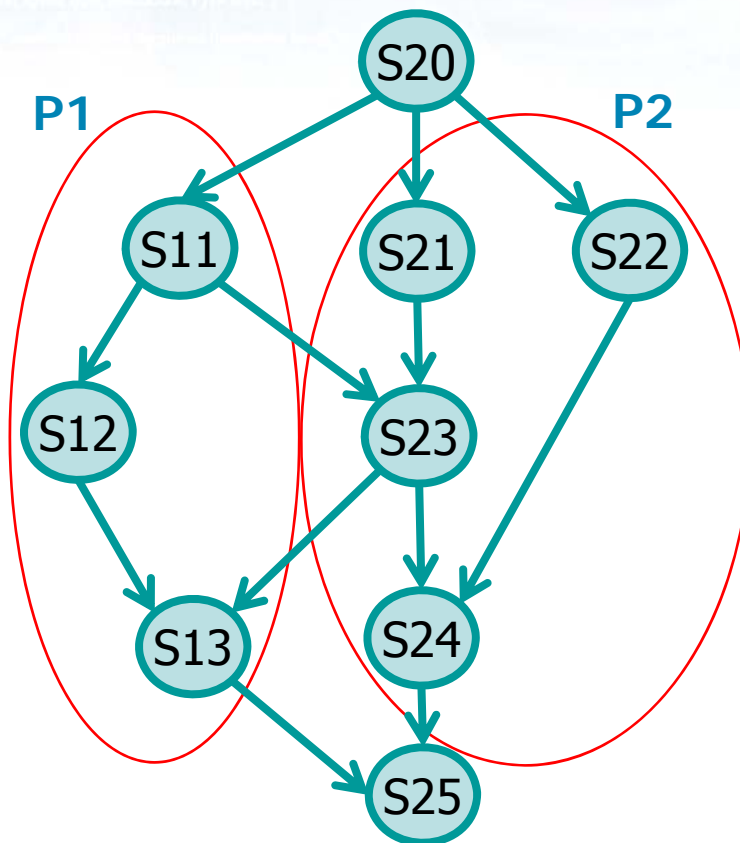
```
main () {  
    S20 ();  
    pid = fork ();  
    if (pid>0) {  
        P1 ();  
        wait ((int *)0);  
    } else {  
        P2 ();  
    }  
    S25 ();  
    return;  
}
```

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);  
}
```

