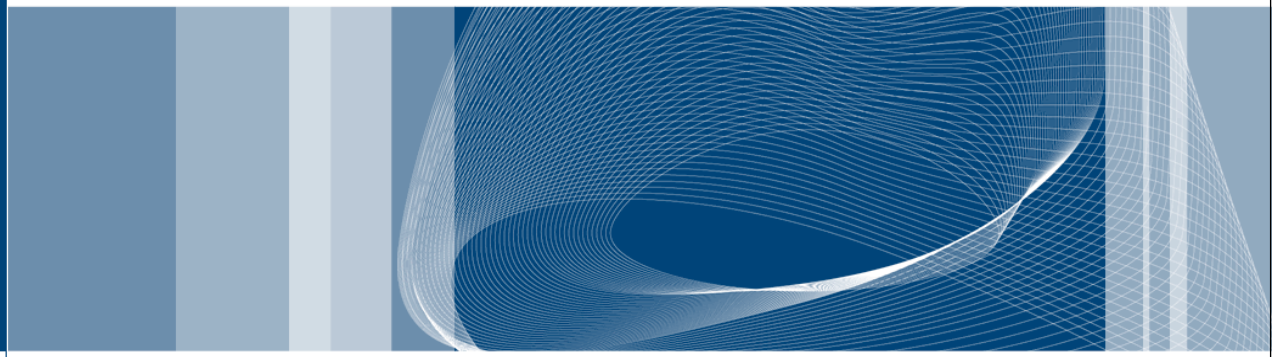
POLITECNICO DI MILANO



Fondamenti di Informatica 2013-2014

Processi

Paola Mussida
Area Servizi ICT

Stampare il pid del processo e del processo che l'ha generato.

Esercizio Pr_1 - Pid

```c
#include <stdio.h>

int main ()
{
   printf ("The process id is %d\n",
                                   (int) getpid ());

   printf ("The parent process id is %d\n",
                                  (int) getppid ());

   sleep(60);
   return 0;
}
```

Generare un processo figlio e mostrare i pid sia del padre che del figlio.

Esercizio Pr_2 - Genera


```c
#include <stdio.h>
#include <stdlib.h> //exit
#include <sys/types.h>
#include <unistd.h> //fork

int main ()
{
    pid_t child_pid;
    printf ("M the main program process id is %d\n",
                                        (int) getpid ());

    child_pid = fork ();
    if (child_pid ==-1) {
        printf("\nan error occurred executing fork\n");
        exit(0);
    }
```

Esercizio Pr_2 - Genera

```
    else {
        if (child_pid != 0) {
            printf ("P this is the parent process, with id %d\n",
                                                    (int) getpid ());

            printf ("P this is the parent process, the child's
                                process id is %d\n", (int) child_pid);
        }
        else {
            printf ("C this is the child process, with id %d\n",
                                                    (int) getpid ());

            printf ("C this is the child process, my parent has
                                        id %d\n", (int) getppid ());
        }

    }
    return 0;
}
```

Generare un processo figlio che lancia i due differenti programmi esterni

- ✓ LS
- ✓ PWD

tramite la funzione execl.

```c
#include <stdio.h>
#include <stdlib.h> //necessario per exit
#include <sys/types.h>
#include <unistd.h>



int spawnLS (char* path, char* nome,
                    char* par1, char* par2);

int spawnPWD (char* path, char* nome );
```

```
int main ()
{
    int scelta;

     /* La lista di argomenti da passare al
                                comando "ls".  */

    char * path = "/bin/ls";
    char * nome = "ls";
    char * par1 = "-l";
    char * par2 = "/";
```

Esercizio Pr_3 - Execl

```
    do{
        printf ("\nPremere: \n- 1 per ls; \n - 2 per pwd; \n");
        scanf("%d",&scelta);
    }while (scelta != 1 && scelta!=2);

    switch (scelta) {
        case 1:
                        spawnLS(path, nome, par1, par2);
                        break;

        case 2:
                        spawnPWD ("bin/pwd", "pwd");
                        break;
    }

    printf ("done with main program\n");

    return 0;
}
```

Esercizio Pr_3 - Execl


```
int spawnLS (char* path, char* nome, char* par1, char* par2)
{
    pid_t child_pid;
    child_pid = fork ();                    /* Duplica questo processo.  */

    if (child_pid ==-1) {
        printf("\nan error occurred executing fork\n");
        exit(0);
    }
    else {

        if (child_pid != 0)                 /* Questo e' il processo padre. */
                return (int) child_pid;
        else {
                /* Ora esegue LS */
                execl (path, nome, par1, par2, NULL);
                /* La funzione execl ritorna solo in caso di errore.  */
                fprintf (stderr, "an error occurred in execl\n");
                abort ();
        }

    }
}
```

Esercizio Pr_3 - Execl


```
int spawnPWD (char* path, char* nome)
{
    pid_t child_pid;
    child_pid = fork ();                    /* Duplica questo processo.  */

    if (child_pid ==-1) {
          printf("\nan error occurred executing fork\n");
          exit(0);
    }
    else {
          if (child_pid != 0)
                /* Questo e' il processo padre.  */
                return (int) child_pid;
          else {
                /* Ora esegue PWD */
                execl (path, nome, NULL);
                /* La funzione execl ritorna solo in caso di errore */
                fprintf (stderr, "an error occurred in execl\n");
                abort ();
          }
    }
}
```

Scrivere un programma in cui il padre stampi 100000 volte una stringa differente da quella stampata dal figlio.

Esercizio Pr_4 - Concorrenza

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main ()
{
      pid_t pid;
      int i=0;
      printf("I'm the original process with PID %d and
                              PPID %d.\n", getpid(),getppid());

      /* Replicazione. Padre e figlio continuano da qui.*/
      pid=fork();

      if (pid == -1) {
            printf("\nan error occurred executing fork \n");
            exit(0);
      }
```

Esercizio Pr_4 - Concorrenza

```
        else {

                if (pid!=0) {
                        /* padre */
                        for (i=0; i<100000; i++)
                                printf("P\n");
                }


                else {
                        /* figlio */
                        for (i=0; i<100000; i++)
                                printf("F\n");
                }


        }
}
```

Generare un numero prefissato di processi, ognuno caratterizzato da una differente sigla composta da un carattere (A, B, C, ...) assegnata durante la creazione dal padre. Tale sigla deve essere memorizzata in ogni processo figlio generato.

Il processo padre deve visualizzare il PID di ogni processo figlio generato e la sigla assegnatagli.

Ogni processo generato deve visualizzare il proprio PID e la stringa assegnatagli dal padre.

Esercizio Pr_5 - Nomi


```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#define MAX_FIGLI 5

int main ()
{
    pid_t pid;
    char sigla;
    int i;
    printf("I'm the original process with PID %d.\n",
                                                getpid());

    /* Ciclo per generare i figli. */
    for (i = 0; i<MAX_FIGLI; i++) {
                        /* ... */
```

Esercizio Pr_5 - Nomi


```
  sigla = 'A' + i;  // Imposto la sigla i-esima
  pid = fork();     // Genero il figlio i-esimo
  if (pid == -1) {
      printf("\nan error occurred executing fork \n");
      exit(-1);
  }
  else {
      if (pid!=0) {    /* Basato sul valore di ritorno della fork */
        /* pid != 0, quindi siamo nel padre  */
        printf("I'm the parent process with PID %d.\n", getpid());
        printf("My child has signature %c and PID %d.\n",sigla,pid);
      }
      else  {        /* pid e' zero, quindi deve essere il figlio */
        printf("I'm the child process with signature %c and PID %d.
                                                    \n", sigla, getpid());
        exit(0);      /* il figlio non deve continuare il ciclo */
      }
  }
 } //Termina il ciclo for
 // In alternativa il padre avrebbe potuto fare un for con le wait di tutti i figli
 printf("PID %d terminates.\n",getpid());
 exit(0);
}
```

Implementare un programma che generi un numero prefissato di processi figli.
Il processo padre deve memorizzare, per ogni figlio generato, il PID del nuovo processo e il valore da esso restituito al termine della propria esecuzione.
Ogni processo figlio generato deve richiedere all'utente di inserire un carattere. Tale carattere costituisce il valore da restituire al padre.

Esercizio Pr_6 - Acquisizione e Ritorno


```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#define MAX_FIGLI 3

void  main ()
{
    /*array di strutture per contenere le info sui figli*/
    struct ch {
            pid_t pid;
            int res;
    } children [MAX_FIGLI];


    pid_t pidx;
    int status, i, j, uscita=0;
    char c;

    printf("I'm the original process with PID %d.\n", getpid());
```

Esercizio Pr_6 - Acquisizione e Ritorno


```
/*Generazione dei figli*/
for (i=0; i<MAX_FIGLI;i++)
{
  children[i].pid  = fork(); /* generazione i-esimo figlio */

  if (children[i].pid == -1) {      /*verifica errori fork */
        printf("\nan error occurred executing fork\n");
        exit(0);
  }
  if (children[i].pid == 0)
  {
        /* codice figlio (tutti i figli condividono lo stesso
                                                terminale) */
        printf ("\nI'm the son nr. %d ", i+1);
        printf("\nInsert a char: ");
        fflush(stdin);
        c=getchar();
        exit (c);    /* restituisce il carattere acquisito */
  }
}
```

Esercizio Pr_6 - Acquisizione e Ritorno


```
/*Attesa della fine di ogni processo figlio e memorizzazione del valore restituito*/
/* codice padre: attesa termine di tutti i figli */
for (j=0; j< MAX_FIGLI; j++)
{       /*memorizzazione valore restituito dal figlio terminato*/
        pidx= wait (&status);                   /*attesa termine figli*/

        i=0;                /*ricerca del figlio terminato nell'array*/
        uscita=0;
        while (  i<MAX_FIGLI && uscita==0  )
        {
            if (pidx == children[i].pid)
            {
                    printf ("\nSon nr. %d terminated", i+1);
             /*recupero il valore restituito dal figlio terminato*/
                    status = status /256;
                    children[i].res = status;
                    uscita=1;
            }
            i++;
        }
}
```

Esercizio Pr_6 - Acquisizione e Ritorno


```
                                        /*stampa dei risultati*/

    for(i=0; i< MAX_FIGLI; i++)
        printf("\nSon nr. %d with PID %d returned %c\n"
                                                , i+1
                                                , children[i].pid
                                                , children[i].res);

} // chiusura main
```

Implementare un programma che, lanciati due programmi, attenda la fine di entrambi per stabilire quale dei due è terminato per primo.

Il primo comando da invocare è:

"/bin/ls" con i parametri "-l" e "/".

Il secondo comando è semplicemente:

"/bin/pwd".

Esercizio Pr_7 - Primo


```c
#include <stdio.h>
#include <stdlib.h> //necessario per exit
#include <sys/types.h>
#include <unistd.h>

void main ()
{
      int pidx, status;

      /*Generazione dei due figli*/
      pid_t pid1, pid2;
```

Esercizio Pr_7 - Primo


```
/*Generazione primo figlio*/
pid1=fork();

if ( pid1 ==-1) {
        printf("\nan error occurred executing fork\n");
        exit(0);
}

else {
        if ( pid1 == 0)     /* Questo e' il primo figlio.  */
        {

            execl("/bin/ls", "ls","-l","/",NULL);
            /* La funzione execl ritorna solo in caso di errore.*/

            printf ("\nan error occurred in execl \n");
            exit(-1);

        }
```

Esercizio Pr_7 - Primo

```
else
{
        /*generazione del secondo figlio da parte del padre*/
        pid2=fork();
        if ( pid2 ==-1) {
            printf("\nan error occurred executing fork");
            exit(0);
        }
        else {
                if ( pid2 == 0) /*Questo e' il secondo figlio*/
                {
                    execl("/bin/pwd", "pwd",NULL);
                    /* La funzione execl ritorna solo in caso di
                                                    errore.  */
                    printf ("\nan error occurred in execl");
                    exit(-1);
                }
        }
  }
}
```

Esercizio Pr_7 - Primo

```
                    /*codice padre */


   /*attesa del figlio più veloce*/

    pidx = wait(&status);

    if (pidx == pid1)
            printf("\n First son is the faster");

    else
            printf("\n Second son is the faster");


}
```

Provare a creare un processo orfano.

Esercizio Pr_8 - Orfano


```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main ()
{
    pid_t pid;
     printf("I'm the original process with PID %d and PPID %d.\n",
                                                    getpid(),getppid());

    /* Replicazione. Padre e figlio continuano da qui.*/
    pid=fork();

    if (pid == -1) {
            printf("\nan error occurred executing fork \n");
            exit(0);



}
```

Esercizio Pr_8 - Orfano


```
    else {

      if (pid!=0) {  /* Basato sul valore di ritorno della fork()*/
         /* pid != 0, quindi siamo nel padre  */
         printf("I'm the parent process with PID %d and PPID %d.\n"
                                                 , getpid(),getppid());
         printf("My child's PID is %d.\n", pid);
      }
      else  {
         /* pid e' zero, quindi devo essere il figlio*/
         /*Ritardo il processo affinché abbia terminato il padre*/
         sleep(5);
         printf("I'm the child process with PID %d and PPID %d.\n"
                                                 , getpid(),getppid());
      }

      /* Entrambi i processi eseguono l'ultima istruzione prima
                                                    di terminare */
      printf("PID %d terminates.\n",getpid());

    }
}
```

Provare a creare un processo zombie.

Esercizio Pr_9 - Zombie


```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int main ()
{
    pid_t pid;

    printf("I'm the original process with PID %d and PPID %d.\n",
                                                getpid(),getppid());


    /* Replicazione. Padre e figlio continuano da qui.*/
    pid=fork();

    if (pid == -1) {
            printf("\nan error occurred executing fork \n");
            exit(0);
    }
```

Esercizio Pr_9 - Zombie


```
    else {
      if (pid!=0) {  /* Basato sul valore di ritorno della fork() */

            /* pid != 0, quindi siamo nel padre  */
            printf("I'm the parent process with PID %d and PPID %d.
                                                \n", getpid(),getppid());
            printf("My child's PID is %d.\n", pid);
            sleep(1000);
      }

      else {
            /* pid e' zero, quindi devo essere il figlio*/
            printf("I'm the child process with PID %d and PPID %d.
                                                \n", getpid(),getppid());
            /* Termina con uno stato di ritorno */
            exit(42);
      }
    }
}
```

Analisi dell'esecuzione concorrente di un programma:

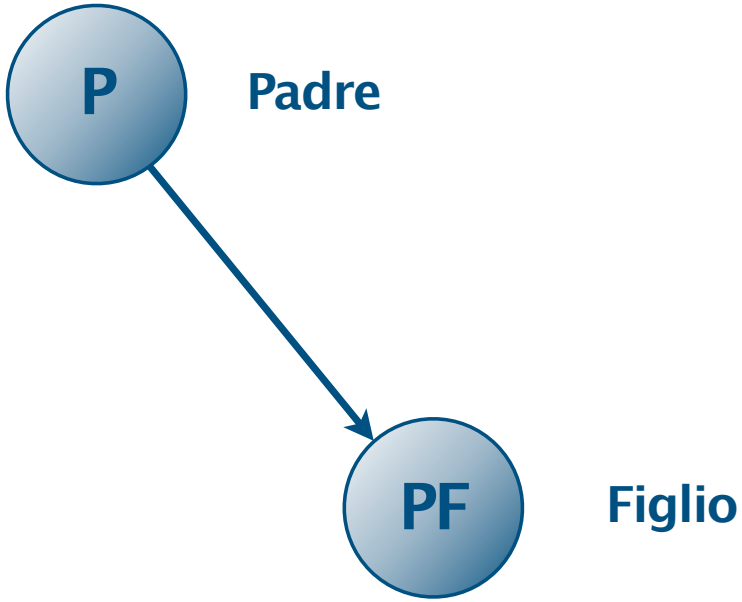
- ✓ albero dei processi;
- ✓ esposizione delle parti di codice di ogni processo;
- ✓ diagrammi di flusso temporale;

Esercizio Pr_10 - Analisi concorrenza


```
void main ()
{
    int v1,v2=14;
    pid_t pid;
    pid=fork();                                  <----   // T1 subito dopo la fork
    if (pid ==-1) {
        printf("\nan error occurred\n");
        exit(-1);
    }
    else {
        if (pid == 0)
        {
            /* Processo figlio. */
            foo(&v1);                                    // v1 = 65 in foo()
            exit(0);                             <----   // T2 subito prima della exit
         }
        else
        {
            /* Processo padre*/                          // v2 = 321 in fun()
            fun(&v2);                            <----   // T3 subito prima della wait
            wait(...);
            v1=455;                              <----   // T4 subito dopo l'assegnamento
        }
    }
}
```

Albero dei processi

Serve per evidenziare le relazioni padre-figlio



Esercizio Pr_10 - Analisi concorrenza

Esporre le parti di codice eseguite da ciascun processo: **P padre**

```
void main ()
{
    int v1,v2=14;
    pid_t pid;
    pid=fork();                                  <----   // T1 subito dopo la fork
    if (pid ==-1) {
        printf("\nan error occurred\n");
        exit(-1);
    }
    else {
        if (pid == 0)
        {
            /* Processo figlio. */
            foo(&v1);                                    // v1 = 65 in foo()
            exit(0);                             <----   // T2 subito prima della exit
         }
        else
        {
            /* Processo padre*/                          // v2 = 321 in fun()
            fun(&v2);                            <----   // T3 subito prima della wait
            wait(...);
            v1=455;                              <----   // T4 subito dopo l'assegnamento
        }
    }
}
```

Esercizio Pr_10 - Analisi concorrenza

Esporre le parti di codice eseguite da ciascun processo: **PF figlio**

```
void main ()
{
    int v1,v2=14;
    pid_t pid;
    pid=fork();                                  <----   // T1 subito dopo la fork
    if (pid ==-1) {
        printf("\nan error occurred\n");
        exit(-1);
    }
    else {
        if (pid == 0)
        {
            /* Processo figlio. */
            foo(&v1);                                    // v1 = 65 in foo()
            exit(0);                             <----   // T2 subito prima della exit
         }
        else
        {
            /* Processo padre*/                          // v2 = 321 in fun()
            fun(&v2);                            <----   // T3 subito prima della wait
            wait(...);
            v1=455;                              <----   // T4 subito dopo l'assegnamento
        }
    }
}
```

Diagramma di flusso temporale dei processi

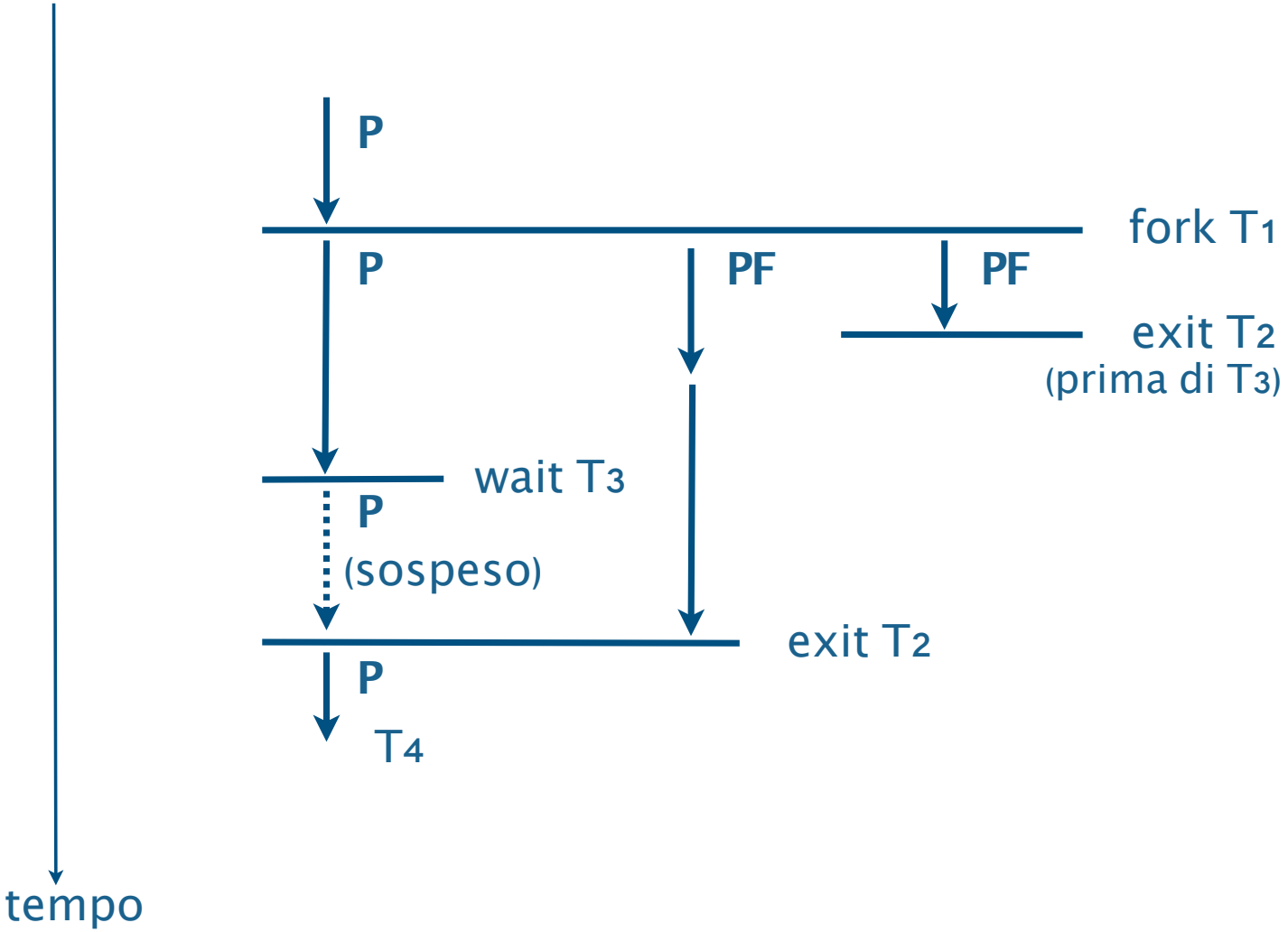


Tabella da compilare per ogni processo P e PF

	T1	T2	T3	T4
PID				
V1				
V2				

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella da compilare per il processo P: padre

	T1	T2	T3	T4
PID	PF	PF	PF	PF
V1	X	X	X	455
V2	14	?	321	321

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella da compilare per il processo PF: figlio

	T1	T2	T3	T4
PID	0	0	?	NE
V1	X	65	?	NE
V2	14	14	?	NE

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Analisi dell'esecuzione concorrente di un programma.

- Un processo padre P crea nell'ordine i tre processi figli F_1, F_2 e F_3 e, dopo averli creati, si mette in attesa della loro terminazione.

- I tre figli evolvono in modo autonomo, eseguendo tre programmi diversi, il cui comportamento è peraltro sconosciuto.
- Quando i processi figli sono tutti terminati, anche il processo padre termina, visualizzando i PID dei tre processi figli, in ordine di terminazione.
- A questo scopo, il processo padre P memorizza l'elenco dei PID dei processi figli in un array di tipo pid_t pid[3], nel quale scrive, uno dopo l'altro, i PID dei tre processi figli creati tramite la primitiva fork().

Esercizio Pr_11 - Analisi concorrenza


```
void main ()
{
    pid_t pid[3], term[3];
    int n, stato;
                <----                                   // creazione P

    /*Generazione primo figlio*/
    pid[0] = fork();<----                               // creazione F1
    //...
    if (pid[0] == 0) {
        /*corpo primo figlio*/
        exit(0);
    }
    /*Generazione secondo figlio*/
    pid[1] = fork();<----                               // creazione F2
    //...
    if (pid[1] == 0) {
        /*corpo secondo figlio*/
        exit(0);
    }

    /*Generazione terzo figlio*/
    pid[2] = fork();<----                               // creazione F3
    //...
    if (pid[2] == 0) {
        /*corpo terzo figlio*/
        exit(0);
    }
```

Esercizio Pr_11 - Analisi concorrenza


```
    /*Ciclo di attesa terminazione figli*/
    for(n=0; n<3; n++)
        term[n] = wait(&stato);


    /*stampa i PID dei figli in ordine di terminazione*/
    for(n=0; n<3; n++)
        printf("Terminato figlio con PID %d ",term[n]);

}
```

Tabella da compilare per ogni processo

	Creaz P	Creaz F1	Creaz F2	Creaz F3
pid[0]				
pid[1]				
pid[2]				

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Albero dei processi

Serve per evidenziare le relazioni padre-figlio

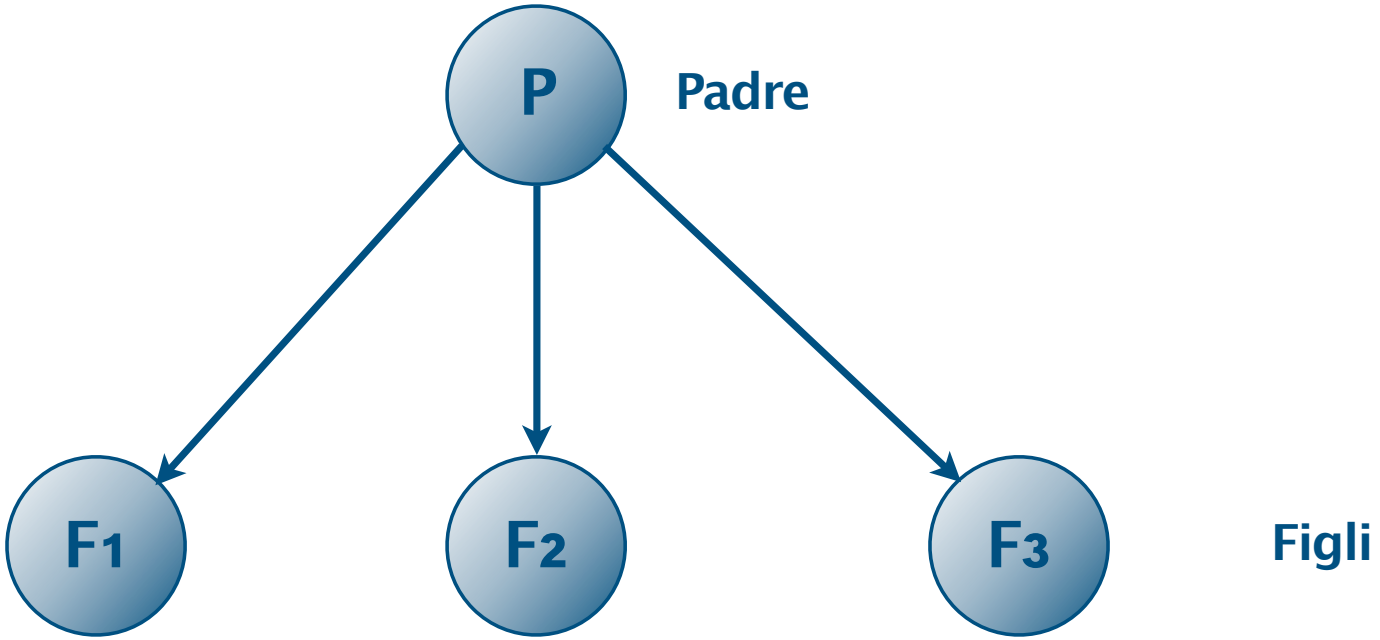


Tabella per il processo P

	Creaz P	Creaz F1	Creaz F2	Creaz F3
pid[0]	X	PID F1	PID F1	PID F1
pid[1]	X	X	PID F2	PID F2
pid[2]	X	X	X	PID F3

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella per il processo F1

	Creaz P	Creaz F1	Creaz F2	Creaz F3
pid[0]	NE	0	?	?
pid[1]	NE	X	?	?
pid[2]	NE	X	?	?

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella per il processo F2

	Creaz P	Creaz F1	Creaz F2	Creaz F3
pid[0]	NE	NE	PID F1	?
pid[1]	NE	NE	0	?
pid[2]	NE	NE	X	?

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella per il processo F3

	Creaz P	Creaz F1	Creaz F2	Creaz F3
pid[0]	NE	NE	NE	PID F1
pid[1]	NE	NE	NE	PID F2
pid[2]	NE	NE	NE	0

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Analisi dell'esecuzione concorrente di un programma:

- ✓ albero dei processi;
- ✓ esposizione delle parti di codice di ogni processo;
- ✓ diagrammi di flusso temporale;

- Il programma seguente viene eseguito inizialmente da un processo P, che crea due processi figli PF_1 e PF_2 (i cui identificatori vengono assegnati alle variabili f_1 e f_2).
- A sua volta il processo PF_1 crea due processi figli PN_1 e PN_2 (i cui identificatori vengono assegnati alle variabili n_1 e n_2).
- Tali processi PN_1 e PN_2 sono immaginabili come "nipoti" del processo P.

Esercizio Pr_12 - Analisi concorrenza


```
void main () {
    pid_t f1, f2, n1, n2;
    /*Generazione primo figlio*/
    f1 = fork();                                    // T1 dopo la fork
    //...
    if (f1 == 0) {
        /*corpo primo figlio*/
        n1 = fork();
        //...
        if (n1 == 0) {
            /* corpo primo nipote */
            fun();
            exit(0);
        } else {
            /*Codice del primo figlio*/
            n2 = fork();
            //...
            if (n2 == 0) {
                /* corpo secondo nipote */
                fun();
                exit(0);
            } else {
                /*fine primo figlio*/
                exit(0);                            // T2 prima della exit
            }
        }
    }
```

Esercizio Pr_12 - Analisi concorrenza


```
    else
    {
        /*Codice del padre*/
        wait(...);
        /*Generazione secondo figlio*/
        f2= fork();
        //...
        if (f2 == 0) {
            /*corpo secondo figlio*/
            fun();
            exit(0);          <----   // T3 prima della exit
        }
        else
        {
            /*Codice del padre*/
            wait(...);
            exit();
        }
    }
}
```

Tabella da compilare per ogni processo

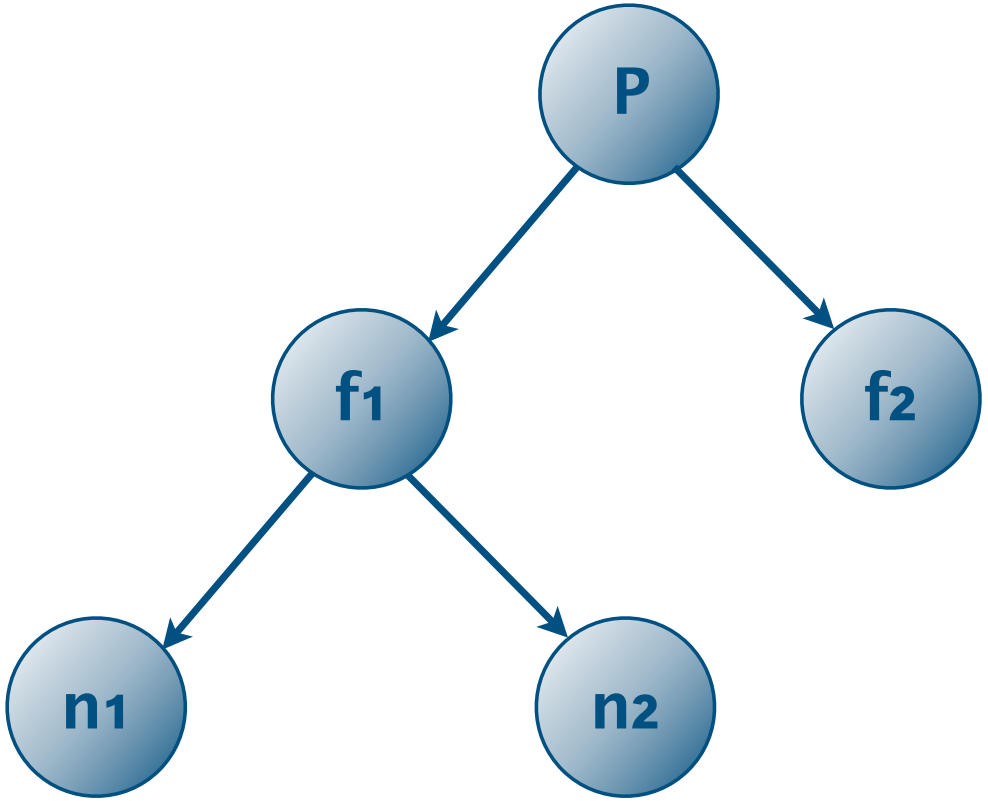
	T_1	T_2	T_3
f_1			
f_2			
n_1			
n_2			

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Albero dei processi

Serve per evidenziare le relazioni padre-figlio



Padre

Figli

Nipoti

Esercizio Pr_12 - Analisi concorrenza


```
void main () {
    pid_t f1, f2, n1, n2;
    /*Generazione primo figlio*/
    f1 = fork();
    //...
    if (f1 == 0) {
        /* ... */
    }
    else
    {
        /*Codice del padre*/
        wait(...);
        /*Generazione secondo figlio*/
        f2= fork();
        //...
        if (f2 == 0) {
            /* ... */
        }
        else
        {
            /*Codice del padre*/
            wait(...);
            exit();
        }
    }
}
```

Esporre le parti di codice eseguite da ciascun processo: **P padre**

Esercizio Pr_12 - Analisi concorrenza

Esporre le parti di codice eseguite da ciascun processo: **f1**

```
f1 = fork();
//...
if (f1 == 0) {
    /*corpo primo figlio*/
    n1 = fork();
    //...
    if (n1 == 0) {
        /* ... */
    } else {
        /*Codice del primo figlio*/
        n2 = fork();
        //...
        if (n2 == 0) {
            /* ... */
        } else {
            /*fine primo figlio*/
            exit(0);
        }
    }
}
```

Esercizio Pr_12 - Analisi concorrenza

Esporre le parti di codice eseguite da ciascun processo: **f2**

```
f2= fork();
//...
if (f2 == 0) {
    /*corpo secondo figlio*/
    fun();
    exit(0);
}
```

Esercizio Pr_12 - Analisi concorrenza

Esporre le parti di codice eseguite da ciascun processo: **n1**

```
n1 = fork();
//...
if (n1 == 0) {
    /* corpo primo nipote */
    fun();
    exit(0);
}
```

Diagramma di flusso temporale dei processi

caso 1: PN1 termina dopo T1 ma prima di T2

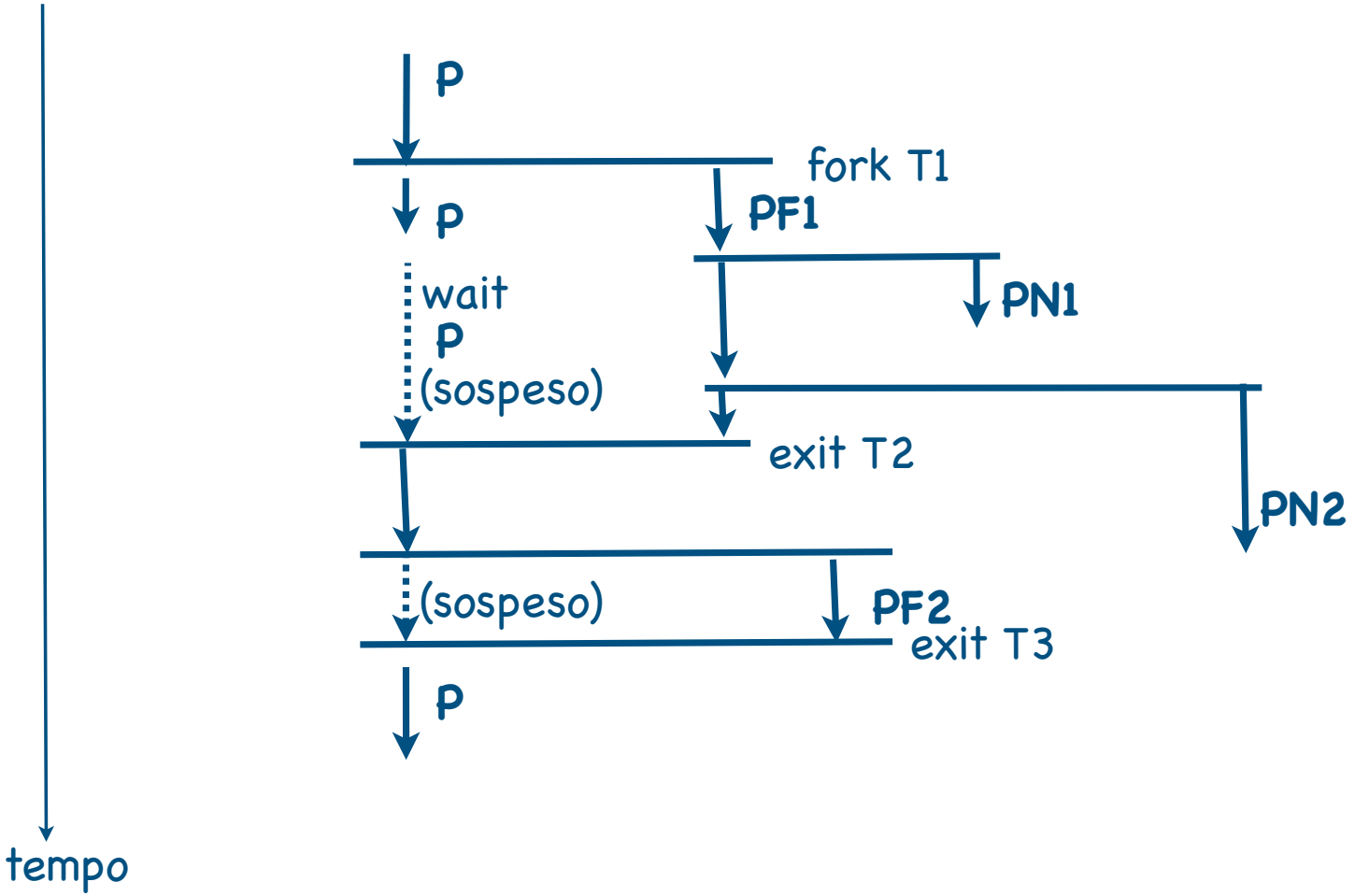


Diagramma di flusso temporale dei processi

caso 2: PN1 termina dopo T2 ma prima di T3

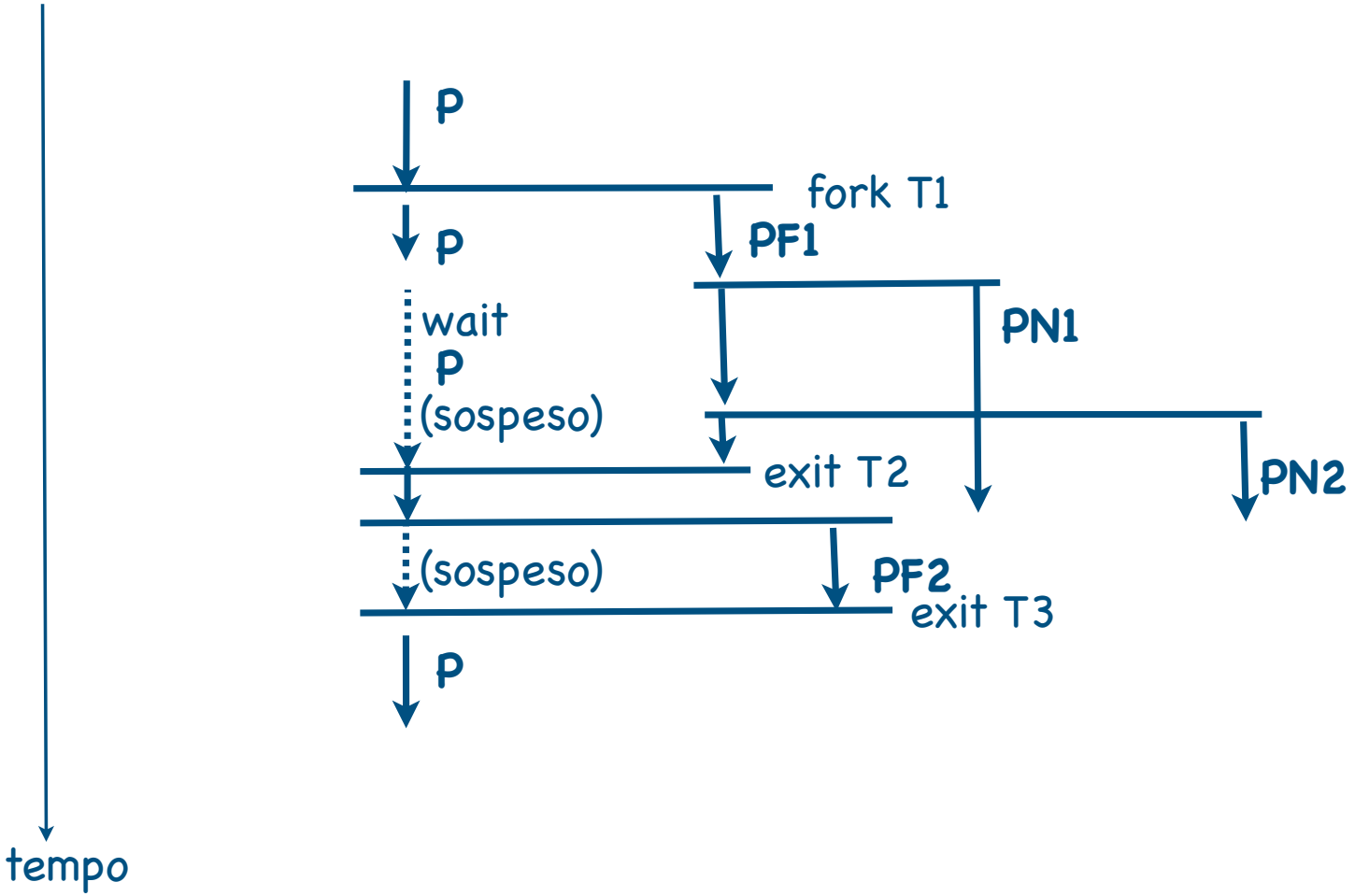


Diagramma di flusso temporale dei processi

caso 3: PN1 termina dopo T3

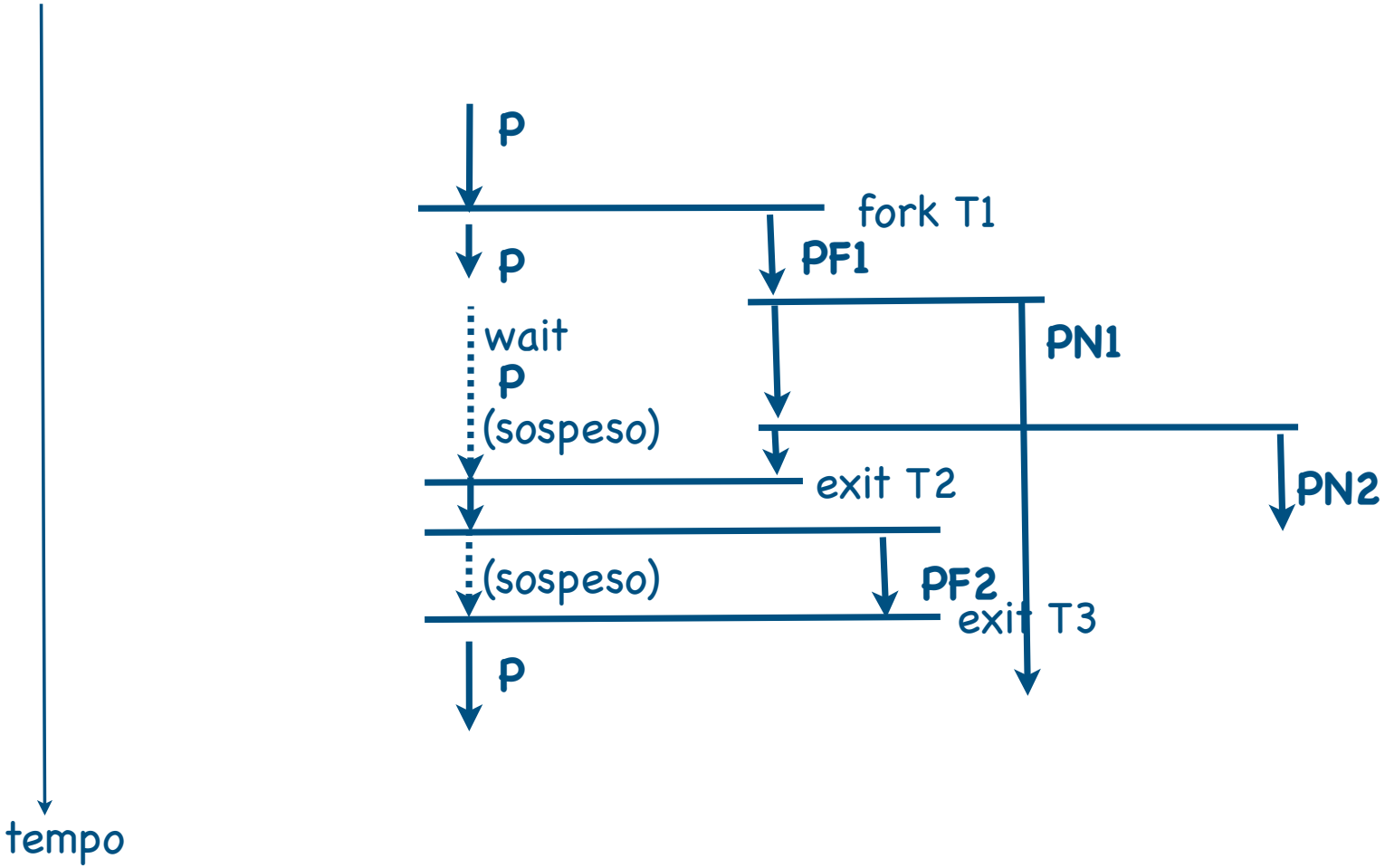


Tabella da compilare per il processo P

	T_1	T_2	T_3
f_1	PF_1	PF_1	PF_1
f_2	X	X	PF_2
n_1	X	X	X
n_2	X	X	X

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella da compilare per il processo f_1

	T_1	T_2	T_3
f_1	0	0	NE
f_2	X	X	NE
n_1	X	PN_1	NE
n_2	X	PN_2	NE

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella da compilare per il processo f2

	T1	T2	T3
f1	NE	NE	PF1
f2	NE	NE	0
n1	NE	NE	X
n2	NE	NE	X

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella da compilare per il processo n1

	T1	T2	T3
f1	NE	?	?
f2	NE	?	?
n1	NE	?	?
n2	NE	?	?

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Analisi dell'esecuzione concorrente di un programma.

- Si supponga che tutte le chiamate ai servizi di sistema abbiano sempre successo e che il S.O. assegni ai processi creati dei pid consecutivi a partire da 111.

Esercizio Pr_13 - TdE 30-04-2002

```
1  void main () {
2      pid_t pid;
3      int i, j, dati[2], status;
4      i=0;
5      dati[0]=dati[1]=-1;
6      for (j=0; j<2; j++) {
7          pid=fork();
8          if (pid==0){
9              dati[i]=j;
10             if (j == 0) {
11                 execl("/bin/pwd","pwd",NULL);
12                 exit(1);
13             }
14             exit(1);  //istr. eseguita solo dal 2o figlio
15         }
16         if (j == 1) pid = waitpid(pid, &status, 0);
17         i++;
18     }
19     exit(0);
20 }
```

Tabella per il processo P

	i	pid	dati[0]	dati[1]
pre istr. 6	0	X	-1	-1
pre istr. 14	1	112	-1	-1
pre istr. 19	2	112	-1	-1

Legenda:

- NE: il contesto non esiste;
- n: la variabile esiste e ha valore n;
- P o PF: indicano i PID dei rispettivi processi;
- X: la variabile esiste ma non è stata ancora inizializzata;
- ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella per il processo 111

	i	pid	dati[0]	dati[1]
pre istr. 6	NE	NE	NE	NE
pre istr. 14	?	?	?	?
pre istr. 19	?	?	?	?

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.

Tabella per il processo 112

	i	pid	dati[0]	dati[1]
pre istr. 6	NE	NE	NE	NE
pre istr. 14	1	0	-1	1
pre istr. 19	NE	NE	NE	NE

Legenda:

- ✓ NE: il contesto non esiste;
- ✓ n: la variabile esiste e ha valore n;
- ✓ P o PF: indicano i PID dei rispettivi processi;
- ✓ X: la variabile esiste ma non è stata ancora inizializzata;
- ✓ ?: la variabile può non esistere o essere caratterizzata da diversi valori.