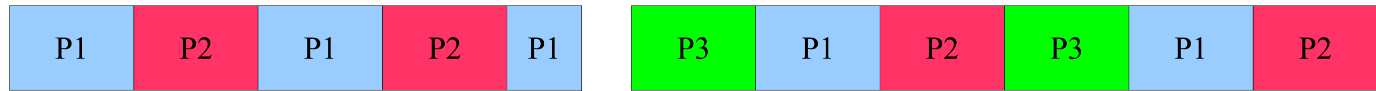
fork() e accesso alle variabili del ‘padre’

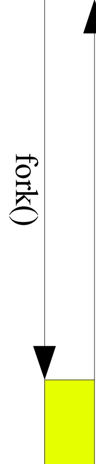
- Ogni processo può, almeno nei limiti consentiti dal kernel, clonare se’ stesso dando vita ad un nuovo processo che risulta esserne una copia identica.
- L’operazione di clonazione viene eseguita invocando la funzione di libreria *fork()*. L’unica differenza fra processo padre (che invoca la fork) e processo figlio (che parte la propria esecuzione dall’istruzione succesiva alla chiamata di fork) risiede nel valore di ritorno della fork stessa:
- il padre ottiene un numero positivo che corrisponde al *pid* del figlio; il figlio ottiene *zero.*
- Il figlio, in ogni caso, possiede una copia esatta di tutte le variabili del padre. Dopo la creazione le variabili hanno vita propria.

```
/*********************************************************
 * $Id: es32.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * Esempio elementare di fork()
 * ******************************************************/

/* interfaccia per fork() */
#include <sys/types.h>
#include <unistd.h>
/* interfaccia per printf() */
#include <stdio.h>

/* prototipo della funzione 'main' */
int main(int argc, char **argv);
static int var_s; /* static var */

/* definizione della funzione 'main' */
int main(int argc, char **argv) {
        int var_a; /* automatic var */
        var_s = 10;
        var_a = 20;
        switch(fork()) {
                case 0: /* child */
                        printf("C: accedo a var_s:%d e var_a:%d\n",
                                        var_s,
                                        var_a);
                        var_s = 11;
                        printf("C: var_s ora vale: %d\n",var_s);
                        var_a++;
                        printf("C: var_a ora vale: %d\n",var_a);
                        break;
```

```
                case -1: /* fork error */
                        break;
                default: /* father */
                        printf("F: accedo a var_s:%d e var_a:%d\n",
                                        var_s,
                                        var_a);
                        var_s = 15;
                        printf("F: var_s ora vale: %d\n",var_s);
                        var_a--;
                        printf("F: var_a ora vale: %d\n",var_a);
        }
        return(0);
}
```

Processi (user space):





S.O. (kernel space):

gestione dei processi

- fork() rappresenta lo strumento fondamentale di unix per la realizzazione di applicativi concorrenti.
- Gli effetti sul sistema di una fork() andata a buon fine sono:
  - l'inserimento di un nuovo processo nella tabella dei processi
  - la duplicazione dei dati e dello stack del processo corrente (nota: Linux implementa la fork con un copy on write, di conseguenza l'effettiva duplicazione delle informazioni avviene solo se uno dei due processi altera (write) il contenuto.
  - duplicazione della kernel area del processo (stessa tabella dei file aperti e stesso stato dei segnali)
- Il processo figlio rappresenta, a tutti gli effetti, una copia del padre. l'unica differenza consiste nel valore di ritorno della fork(). Dalla creazione in poi, ogni processo ha vita propria.

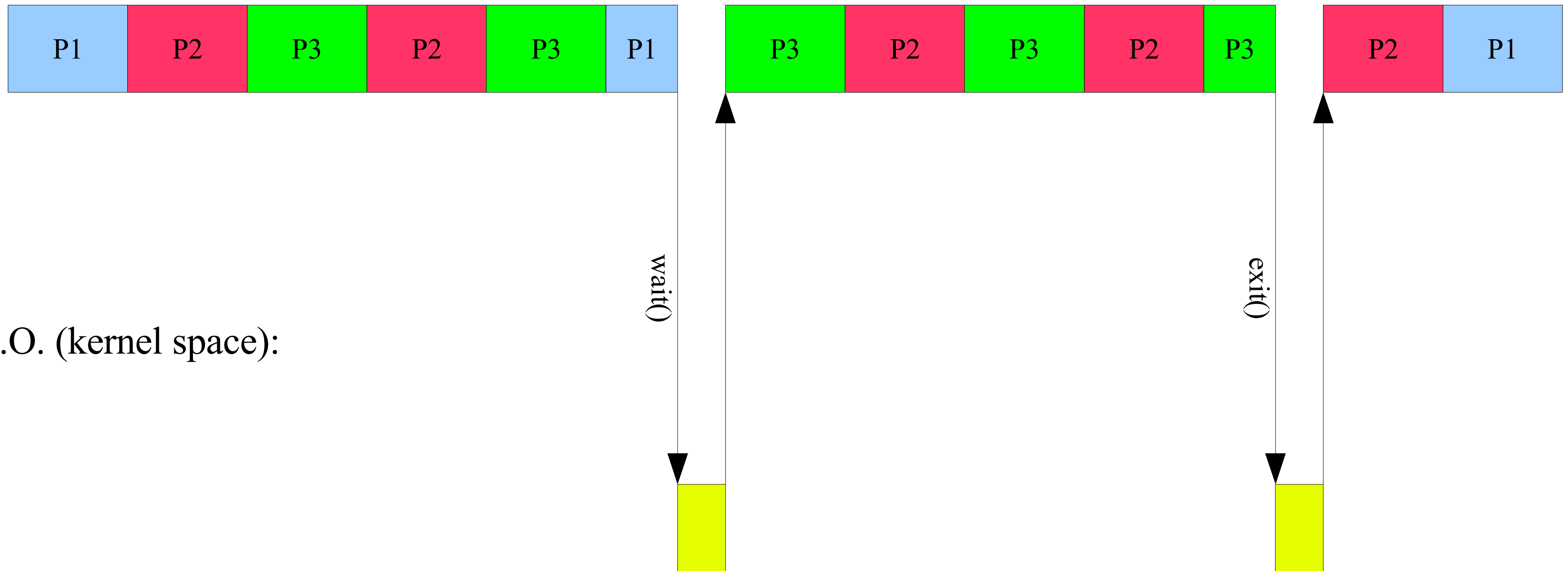
concorrenza dei processi

- il processo ‘padre’ che invoca la fork() ed il processo ‘figlio’ proseguono concorrentemente, ognuno eseguendo il proprio ‘codice’.
- il processo padre può comunque rilevare lo stato del processo figlio, in particolare attendendone il completamento con la primitiva: wait()
- mediante la wait() il padre può anche rilevare informazioni sulla terminazione del processo figlio:
  - valore di ritorno del processo
  - motivo della terminazione, ‘spontanea’ con exit() o return(), o provocata da un ‘segnale’.
- queste informazioni sono riportate dalla wait() nell’intero passato come riferimento a wait(&status).

```
/***********************************************************
 * $Id: es34.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * Passaggio di un intero da figlio a padre con exit()
 * *******************************************************/

/* interfaccia per fork() */
#include <sys/types.h>
#include <unistd.h>
/* interfaccia per printf() */
#include <stdio.h>
/* per wait() */
#include <sys/wait.h>

/* prototipo della funzione 'main' */
int main(int argc, char **argv);
static int var_s; /* static var */

/* definizione della funzione 'main' */
int main(int argc, char **argv) {
        int var_a; /* automatic var */
        int child_status;
        var_s = 10;
        var_a = 20;
        switch(fork()) {
                case 0: /* child */
                        printf("C: accedo a var_s:%d e var_a:%d\n",
                                        var_s,
                                        var_a);
                        var_s = 11;
                        printf("C: var_s ora vale: %d\n",var_s);
```

```
                var_a++;
                printf("C: var_a ora vale: %d\n",var_a);
                exit(var_a);
                break;
        case -1: /* fork error */
                break;
        default: /* father */
                wait(&child_status); /* attendo un (l'unico)
                               figlio prelevando il valore
                               di ritorno */
                printf("F: accedo a var_s:%d e var_a:%d\n",
                                var_s,
                                var_a);
                var_s = 15;
                printf("F: var_s ora vale: %d\n",var_s);
                var_a--;
                printf("F: var_a ora vale: %d\n",var_a);
                printf("F: var_a valeva %d per il figlio\n",
                        WEXITSTATUS(child_status));
    }
    return(0);
}
```

Processi (user space):



condivisione dei file

- processo padre e processo figlio condividono la stessa entry nella tabella di sistema dei file aperti, quindi ogni operazione di uno dei due processi sul file influenza anche l’altro.
- se un processo scrive su di un file condiviso, l’indice di scrittura viene aggiornato per entrambe i processi -> se l’altro processo scrive a sua volta, non sovrascrive ma appende.
- se un processo legge da un file condiviso, l’indice di lettura viene aggiornato per entrambe i processi -> se l’altro processo legge a sua volta, non rilegge gli stessi caratteri ma quelli seguenti.
- la chiusura del file da uno dei due processi, comunque, non implica la chiusura anche per l’altro ma solo la rimozione della condivisione.

```
/*******************************************************
 * $Id: es35.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * Condivione di file handler
 * *****************************************************/

/* interfaccia per fork() */
#include <sys/types.h>
#include <unistd.h>
/* open */
#include <sys/stat.h>
#include <fcntl.h>

/* prototipo della funzione 'main' */
int main(int argc, char **argv);

/* definizione della funzione 'main' */
int main(int argc, char **argv) {
        int i, fh;
        /* preambolo comune */
        if((fh=open("tmpfile",O_CREAT|O_RDWR|O_TRUNC,00777))<0) {
                /* error on open/creat */
                exit(-1);
        }
        switch(fork()) {
                case 0: /* child */
                        for(i=0;i<50;i++) write(fh,"child\n",6);
                        close(fh);
                        break;
                case -1: /* fork error */
                        break;
```

```
		default: /* father */
			for(i=0;i<50;i++) write(fh,"father\n",7);
			close(fh);
	}
	return(0);
}
```

Accesso alle variabili di ambiente

- Le variabili di ambiente possono essere lette accedendo manualmente all’array di stringhe passato come terzo parametro al main:
  - int main(int argc, char *argv[], char *envp[])
- le stringhe di ambiente, comunque lette, si presentano nella forma: “name=value”
- La libc fornisce una serie di funzioni per accedere o modificare le variabili di ambiente (environment):
  - char *getenv(const char *name), restituisce il valore (stringa) della variabile name se è definita o NULL diversamente,
  - int putenv(char *string), inserisce string (nella forma “nome=valore”)
  - int setenv(char *name, char* value, int overwrite), inserisce la coppia “name=value” se non esiste o la aggiorna se overwrite è nonzero.

```
/*******************************************************
 * $Id: es36.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * Accesso alle variabili ambiente (env)
 * *****************************************************/
#include <stdio.h>
/* prototipo della funzione 'main', con accesso specifico
 * ad envp */
int main(int argc, char **argv, char **envp);

/* definizione della funzione 'main' */
int main(int argc, char **argv, char **envp) {
        for(;*envp!=NULL;envp++) {
                printf("%s\n",*envp);
        }
        return(0);
}
```

```
/*********************************************************
 * $Id: es37.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * Accesso alle variabili ambiente con getenv
 * *****************************************************/
#include <stdio.h>
#include <stdlib.h>
/* prototipo della funzione 'main', senza accesso specifico
 * ad envp */
int main(int argc, char **argv);

/* definizione della funzione 'main' */
int main(int argc, char **argv) {
        if(argc>1) {
                /* the varname arg is present */
                printf("Var: %s vale %s\n",
                                argv[1],
                                getenv(argv[1]));
        }
}
```

```
/*******************************************************
 * $Id: es38.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * Modifica delle variabili ambiente con putenv
 * *****************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* prototipo della funzione 'main', senza accesso specifico
 * ad envp */
int main(int argc, char **argv);
static char mybuf[20];
static int mybuf_room;

/* definizione della funzione 'main' */
int main(int argc, char **argv) {
        if(argc>2) {
                /* varname and varvalue args are present */
                mybuf_room = sizeof(mybuf)-1; /* space left */
                if((mybuf_room-=strlen(argv[1]))>=0) {
                        strcat(mybuf,argv[1]);
                }
                if((mybuf_room-=1)>=0) {
                        strcat(mybuf,"=");
                }
                if((mybuf_room-=strlen(argv[2]))>=0) {
                        strcat(mybuf,argv[2]);
                }
```

```
            if(mybuf_room>=0) {
                    if(putenv(mybuf)) {
                            /* error on putenv */
                            printf("Error on putenv\n");
                    } else {
                    printf("Var: %s vale %s\n",
                            argv[1],
                            getenv(argv[1]));
                    }
            } else {
                    printf("No room left on mybuf\n");
            }
    }
    return(0);
}
```

esecuzione di programmi

- all’interno di un programma ‘c’ è possibile richiamare altri programmi eseguibili utilizzando le funzioni della libc appartenenti alla famiglia ‘exec’.
- I dati ed il codice del programma che richiede exec vengono sovrascritti da quelli del programma invocato. Se exec ha esito positivo, non torna.
- i descrittori dei file aperti si propagano al programma eseguito
- i segnali (vedi oltre...) collegati a funzioni vengono riportati allo stato iniziale (il codice di tali funzioni viene sovrascritto!)
- se il nuovo eseguibile ha i bit SUID o SGID settati, l’id effettivo dell’utente/gruppo del processo viene aggiornato mentre rimane invariato l’id reale di utente/gruppo.

la famiglia exec...

- int execve(char *filename, char *argv[], char *envp[])
  - capostipite: argv ed envp passati come tabella, filename non espanso
- int execl(char *filename, char *arg, ...)
  - envp uguale al corrente, argv come lista variabile di stringhe terminate da NULL, filename non espanso
- int execlp(char *filename, char *arg, ...)
  - come execl, ma filename viene ricercato secondo la var. PATH
- int execle(char *filename, char *arg, ..., char *envp[])
  - come execl ma envp viene passato come tabella
- int execv(char *filename, char *argv[])
  - come execl ma argv viene passato come tabella
- int execvp(char *filename, char *argv[])
  - come execv ma filename viene ricercato secondo PATH

```
/*****************************************************
 * $Id: es39.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * Ereditarieta' dell'environment
 * ***************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
/* prototipo della funzione 'main', senza accesso specifico
 * ad envp */
int main(int argc, char **argv);
static char mybuf[20];
static int mybuf_room;

/* definizione della funzione 'main' */
int main(int argc, char **argv) {
        if(argc>2) {
                /* varname and varvalue args are present */
                mybuf_room = sizeof(mybuf)-1; /* space left */
                if((mybuf_room-=strlen(argv[1]))>=0) {
                        strcat(mybuf,argv[1]);
                }
                if((mybuf_room-=1)>=0) {
                        strcat(mybuf,"=");
                }
                if((mybuf_room-=strlen(argv[2]))>=0) {
                        strcat(mybuf,argv[2]);
                }
```

```
		if(mybuf_room>=0) {
			if(putenv(mybuf)) {
				/* error on putenv */
				printf("Error on putenv\n");
			} else {
			printf("Var: %s vale %s\n",
				argv[1],
				getenv(argv[1]));
			}
			printf("Exec:%d\nerrno: %d\n",
			execl("es36","es36",NULL),
			errno);
		} else {
			printf("No room left on mybuf\n");
		}
	}
	return(0);
}
```

fork() & exec

- spesso l’esecuzione di un programma esterno è demandata ad un processo figlio, in modo che il programma corrente possa continuare la propria esecuzione:
- le shell utilizzano proprio questo modello per eseguire i comandi esterni
- prima dell’exec, il padre può ‘preparare’ l’environment ed i file descriptor con i quali eseguire il comando esterno: si pensi a come la shell possa instaurare la ridirezione:
  - la shell esegue il parsing della command line identificando il nome del programma, i simboli di ridirezione ed i nomi dei file ‘bersaglio’
  - la shell ‘forka’ un nuovo processo, apre i file bersaglio come stdin, stdout, stderr, ...
  - il processo forkato richiede la exec per eseguire il comando esterno

```
/*********************************************************
 * $Id: es40.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * Ereditarieta' dell'environment
 * ******************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
/* prototipo della funzione 'main', senza accesso specifico
 * ad envp */
int main(int argc, char **argv);
static char mybuf[20];
static int mybuf_room;

/* definizione della funzione 'main' */
int main(int argc, char **argv) {
        if(argc>2) {
                /* varname and varvalue args are present */
                mybuf_room = sizeof(mybuf)-1; /* space left */
                if((mybuf_room-=strlen(argv[1]))>=0) {
                        strcat(mybuf,argv[1]);
                }
                if((mybuf_room-=1)>=0) {
                        strcat(mybuf,"=");
                }
                if((mybuf_room-=strlen(argv[2]))>=0) {
                        strcat(mybuf,argv[2]);
                }
```

```
			if(mybuf_room>=0) {
				if(putenv(mybuf)) {
					/* error on putenv */
					printf("Error on putenv\n");
				} else {
				printf("Var: %s vale %s\n",
					argv[1],
					getenv(argv[1]));
				}
				if(fork()==0) { /* the child */
					printf("Exec:%d\nerrno: %d\n",
							execl("es36","es36",NULL),
							errno);
				}
			} else {
				printf("No room left on mybuf\n");
			}
		}
		return(0);
	}
```

```
/***********************************************************
 * $Id: es41.c,v 1.1 2002/02/22 15:05:48 valealex Exp $
 * exec con ridirezione
 * ********************************************************/
/* interfaccia per fork() */
#include <sys/types.h>
#include <unistd.h>
/* interfaccia per printf() */
#include <stdio.h>
/* open */
#include <fcntl.h>
#include <sys/stat.h>
/* prototipo della funzione 'main' */
int main(int argc, char **argv);
/* definizione della funzione 'main' */
int main(int argc, char **argv) {
        switch(fork()) {
                case 0: /* child */
                        close(1); /* chiudo stdout */
                        open("tmpfile",O_CREAT|O_RDWR|O_TRUNC,00777);
                        /* open apre il file con il primo descrittore
                         * libero: 0, occupato; 1, libero */
                        execl("es36","es36",NULL);
                        break;
                case -1: /* fork error */
                        break;
                default: /* father */
                        printf("sono es41, ancora attivo!\n");
        }
        return(0);
}
```