# Lezione 14

- *Scrivere un programma che simuli il sistema di prenotazione di voli* ACME. *Il programma acquisisce da std input il numero di posti da riservare rispettivamente dall'ufficio A e dall'ufficio B. Il* `main` *crea due processi figli,* PA *e* PB*, a cui passa il numero di posti da riservare rispettivamente dall'ufficio A e dall'ufficio B. I processi* PA *e* PB *chiamano ripetutamente la funzione* `acmebook` *per riservare i posti uno alla volta. La funzione* `acmebook` *utilizza il meccanismo del locking per effettuare le prenotazioni. Quando non ci sono più posti liberi, il programma termina, altrimenti aspetta da std input un'altra coppia di interi, rappresentanti i posti da riservare dagli uffici A e B.*

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

#define MAXLENGTH 3
char buf[MAXLENGTH];

void prenota(int n);
int acmebook();

main() {
    int n,n1,n2,fd,pid1,pid2;

    while(1) {

        if((fd=open("prenotazioni.txt",O_RDONLY))==-1) {
            perror("Errore in apertura del file delle prenotazioni");
            exit(1);
        }

        /* Lettura dei posti disponibili. */
        if(read(fd,buf,MAXLENGTH)>0) {
            n=atoi(buf);

            if(n==0) {
                printf("Non ci sono piu' posti disponibili.\n");
                exit(0);
            }

        }
        else
            break;

        close(fd);
```

```
        printf("Posti disponibili: %d\n",n);
        /* Viene chiesto il numero di posti da prenotare dall'ufficio A. */
        printf("Numero di posti da riservare dall'ufficio A: ");
        scanf("%d",&n1);
        /* Viene chiesto il numero di posti da prenotare dall'ufficio B. */
        printf("Numero di posti da riservare dall'ufficio B: ");
        scanf("%d",&n2);

        switch(pid1=fork()) {
            case -1:
                perror("Errore nella creazione del primo figlio");
                exit(2);
            case 0:
                prenota(n1);
            default:

                switch(pid2=fork()) {
                    case -1:
                        perror("Errore nella creazione del secondo figlio");
                        exit(3);
                    case 0:
                        prenota(n2);
                    default:
                        /* Il padre attende la terminazione dei figli. */
                        waitpid(pid1,NULL,0);
                        waitpid(pid2,NULL,0);
                }

        }

    }

}

void prenota(int n) {
    int i,error_code;

    for(i=0;i<n;i++) {

        /* se acmebook restituisce -1 significa che non vi sono
         * posti disponibili per soddisfare la prenotazione.
         */
        if((error_code=acmebook())==-1) {
            printf("Numero di posti insufficiente.\n");
            exit(error_code);
        }

    }

    exit(0);
```

```
}

int acmebook() {
    struct flock ldata;
    ldata.l_type=F_WRLCK;
    ldata.l_whence=SEEK_SET;
    ldata.l_start=0;
    ldata.l_len=MAXLENGTH;
    int error_code=0,fd,n,i;

    /* Apertura in lettura/scrittura del file delle prenotazioni */
    if((fd=open("prenotazioni.txt",O_RDWR))==-1) {
        perror("Errore in apertura del file delle prenotazioni");
        exit(1);
    }

    /* Write locking del file delle prenotazioni con eventuale
     * sospensione del processo nel caso in cui il file sia
     * gia' bloccato da un altro processo.
     */
    if(fcntl(fd,F_SETLKW,&ldata)==-1) {
        perror("Errore nel blocco del file delle prenotazioni");
        exit(4);
    }

    if(read(fd,buf,MAXLENGTH)>0) {
        n=atoi(buf);

        if(n>0) {
            n--;
            sprintf(buf,"%d",n);

            /* Riposizionamento all'inizio del file prima della scrittura. */
            if(lseek(fd,0,SEEK_SET)==-1) {
                perror("Errore di riposizionamento nel file delle prenotazioni");
                exit(5);
            }

            /* Il buffer viene 'ripulito' da eventuali caratteri spuri. */
            for(i=strlen(buf);i<MAXLENGTH;i++)
                buf[i]=' ';

            /* Scrittura su disco del nuovo numero di posti disponibili. */
            if(write(fd,buf,MAXLENGTH)==-1) {
                perror("Errore in scrittura nel file delle prenotazioni");
                exit(6);
            }

        }
        else
```

```
            error_code=-1;

    }

    /* Impostazione del componente l_type della struttura flock
     * per rimuovere il lock.
     */
    ldata.l_type=F_UNLCK;

    /* Rilascio del blocco sul file delle prenotazioni. */
    if(fcntl(fd,F_SETLKW,&ldata)==-1) {
        perror("Errore nel rilascio del file delle prenotazioni");
        exit(5);
    }

    close(fd);
    return error_code;
}
```

- *Siano* P1 *e* P2 *due processi che lavorano sullo stesso file. Supponiamo che* P1 *esegua un lock sulla sezione* SX *del file e* P2 *esegua un lock sulla sezione* SY *dello stesso file. Che cosa succede se poi* P1 *tenta di fare un lock su* SY *con* ***F_SETLKW*** *e* P2 *tenta di fare un lock su* SX *con* ***F_SETLKW****?*

  Per verificare cosa succede nella situazione descritta dal testo dell'esercizio, possiamo scrivere, compilare ed eseguire il seguente programma:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

main() {
    int fd;
    struct flock first_lock;
    struct flock second_lock;

    first_lock.l_type=F_WRLCK;
    first_lock.l_whence=SEEK_SET;
    first_lock.l_start=0;
    first_lock.l_len=10;

    second_lock.l_type=F_WRLCK;
    second_lock.l_whence=SEEK_SET;
    second_lock.l_start=10;
    second_lock.l_len=5;

    if((fd=open("file",O_RDWR))==-1) {
        perror("Errore nell'apertura del file");
        exit(1);
    }
```

```
    if(fcntl(fd,F_SETLKW,&first_lock)==-1) {
        perror("Errore nell'esecuzione della prima operazione di blocco");
        exit(2);
    }

    printf("A: blocco completato con successo (PID %d)\n",getpid());

    switch(fork()) {
        case -1:
            perror("Errore nell'esecuzione della fork");
            exit(3);
        case 0:

            if(fcntl(fd,F_SETLKW,&second_lock)==-1) {
                perror("Errore nell'esecuzione della seconda operazione di blocco");
                exit(4);
            }

            printf("B: blocco completato con successo (PID %d)\n",getpid());

            if(fcntl(fd,F_SETLKW,&first_lock)==-1) {
                perror("Errore nell'esecuzione della terza operazione di blocco");
                exit(5);
            }

            printf("C: blocco completato con successo (PID %d)\n",getpid());
            exit(0);
        default:
            /* Pausa di 10 secondi. */
            sleep(10);

            if(fcntl(fd,F_SETLKW,&second_lock)==-1) {
                perror("Errore nell'esecuzione della quarta operazione di blocco");
                exit(6);
            }

            printf("D: blocco completato con successo (PID %d)\n",getpid());
    }

}
```

Quello che succede lanciando il programma è riportato qui di seguito:

```
A: blocco completato con successo (PID 3603)
B: blocco completato con successo (PID 3604)
Errore nell'esecuzione della quarta operazione di blocco: Resource deadlock avoided
C: blocco completato con successo (PID 3604)
```

Quindi Unix è in grado di identificare il deadlock che si verifica, segnalandolo con un opportuno messaggio d'errore (ed evitando la situazione di

stallo dei due processi). In questo caso infatti la chiamata a `fcntl` ritorna immediatamente restituendo `-1` al chiamante ed impostando la variabile speciale `errno` con il valore `EDEADLK`. Nel caso in cui il deadlock coinvolga più di due processi tuttavia `fcntl` non è in grado di rilevarlo.