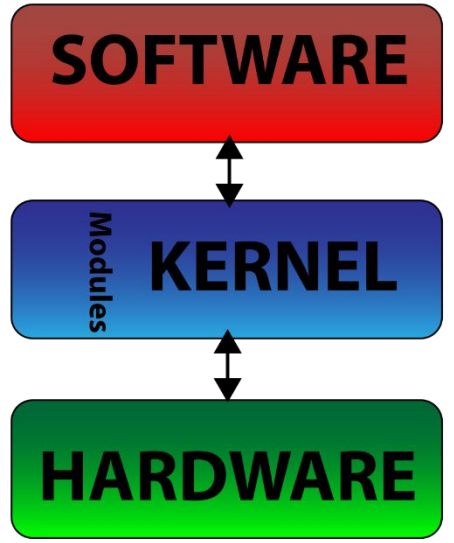
UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

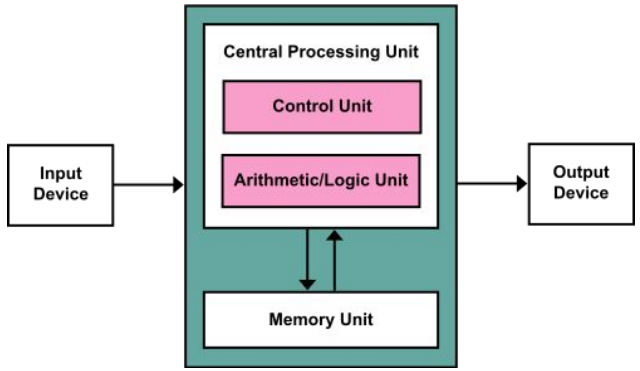
Corso di Sistemi Operativi
A.A. 2019/20



Esercitazione
Gestione Processi

Docente:
Domenico Daniele Bloisi

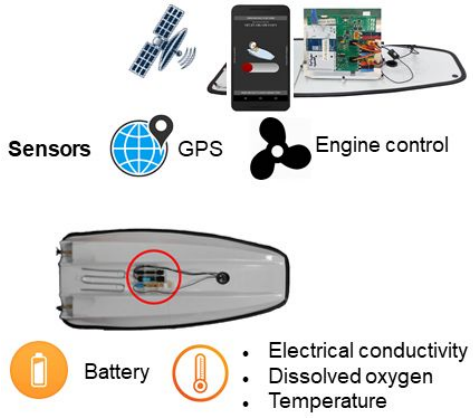
Ottobre 2019

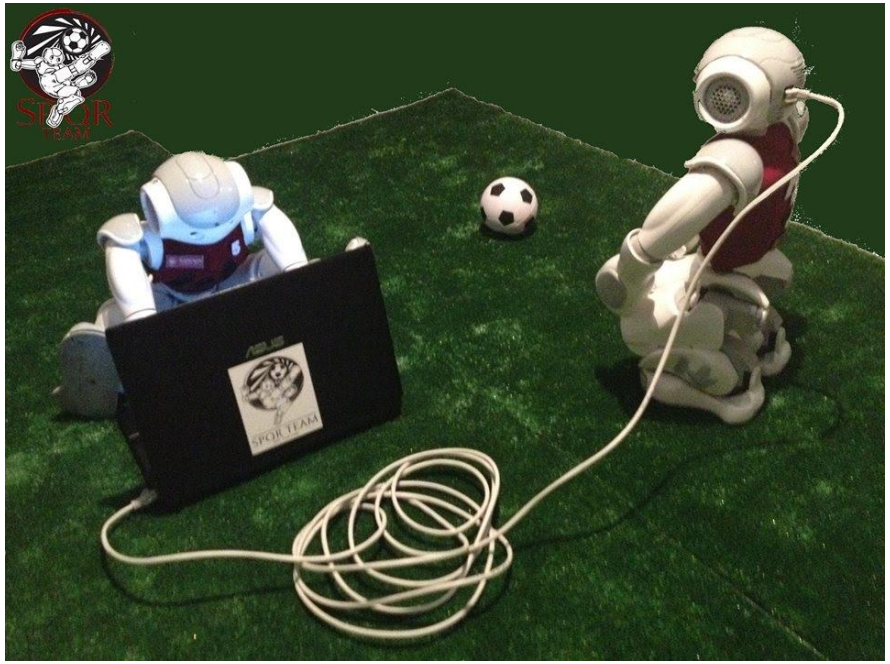


Domenico Daniele Bloisi

- Ricercatore RTD B
  Dipartimento di Matematica, Informatica ed Economia
  Università degli studi della Basilicata
  http://web.unibas.it/bloisi

- SPQR Robot Soccer Team
  Dipartimento di Informatica, Automatica e Gestionale Università degli studi di Roma “La Sapienza”
  http://spqr.diag.uniroma1.it





Ricevimento

- In aula, subito dopo le lezioni
- Martedì dalle 11:00 alle 13:00 presso:
  Campus di Macchia Romana
  Edificio 3D (Dipartimento di Matematica, Informatica ed Economia)
  II piano, stanza 15

Email: domenico.bloisi@unibas.it

Credits

Queste slide derivano dai contenuti dei corsi

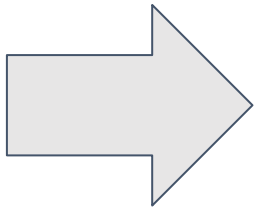
- “Sistemi Operativi”
  del Prof. Giorgio Grisetti
  https://sites.google.com/diag.uniroma1.it/sistemi-operativi-1819

- “Introduction to Computer Systems”
  Instructors Greg Ganger and Roger Dannenberg
  https://www.cs.cmu.edu/afs/cs/academic/class/15213-f09/www/lectures/11-exceptions.pdf

Esercizio 1

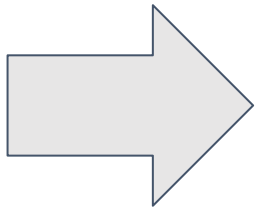
Cosa stampa il programma a lato?
Che cosa succede quando `fn0` arriva alla fine del ciclo?

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

# define A_STEPS 2
# define B_STEPS 5

const char name_0[] = "A|";
const char name_1[] = "B|";
const char *name = name_0;

void fn0() {
    for(unsigned int i = 0; i < A_STEPS; i++) {
        printf ("%s iterazione: #%d \n", name, i);
        sleep(1);
    }
}
```



Esercizio 1

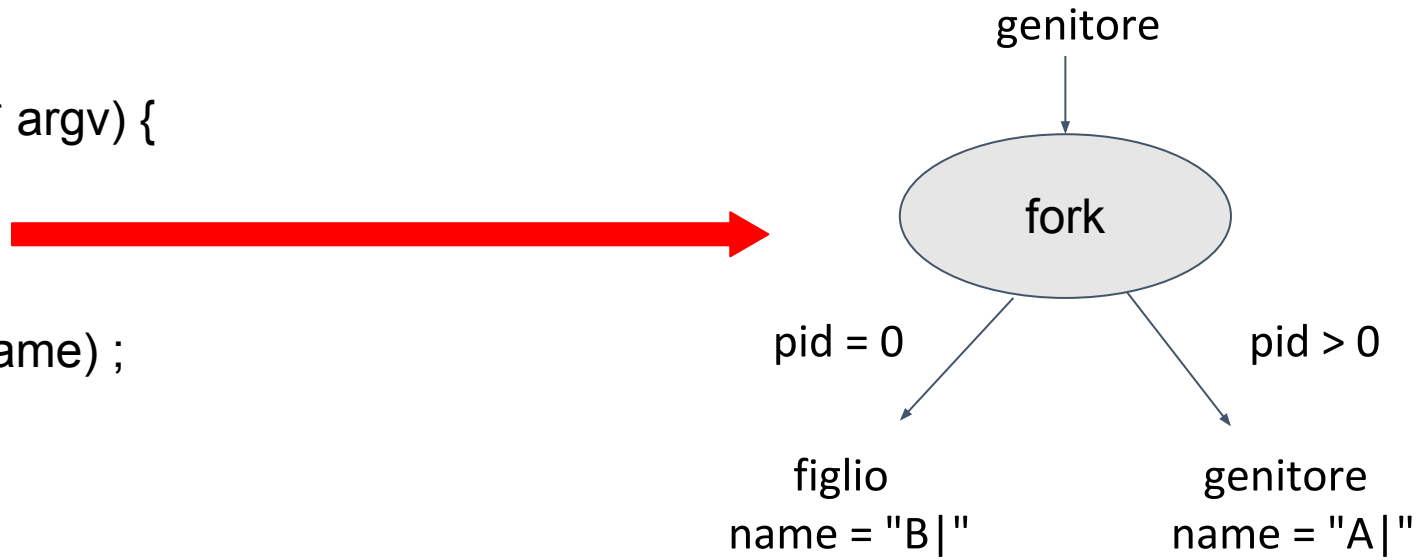
```
void fn1() {
    for(unsigned int i = 0; i < B_STEPS; i++) {
        printf ("%s iterazione: #%d \n", name, i);
        sleep(1);
    }
}

int main(int argc, char** argv) {
    printf ("ciao\n");
    pid_t pid = fork();
    if(pid < 0) {
        printf("%s exit ", name) ;
        exit(1);
    }
```



```
    if(pid == 0) {
        name = name_1;
        fn1();
    }
    else {
        fn0();
    }
    printf("arrivederci\n");
    exit(0);
}
```

Soluzione Esercizio 1

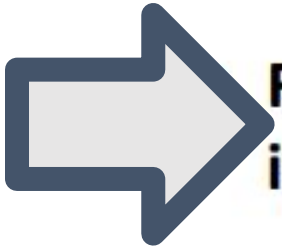
```
int main(int argc, char** argv) {
    printf("ciao\n");
    pid_t pid = fork();
    if(pid < 0) {
        printf("%s exit ", name) ;
        exit(1);
    }
    if(pid == 0) {
        name = name_1;
        fn1();
    }
    else {
        fn0();
    }
    printf("arrivederci\n");
    exit(0);
}
```



`fork`: Creating New Processes

- **`int fork(void)`**
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's **`pid`** to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```



Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Understanding fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

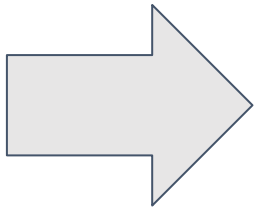
`hello from parent` *Which one is first?* `hello from child`

Soluzione Esercizio 1

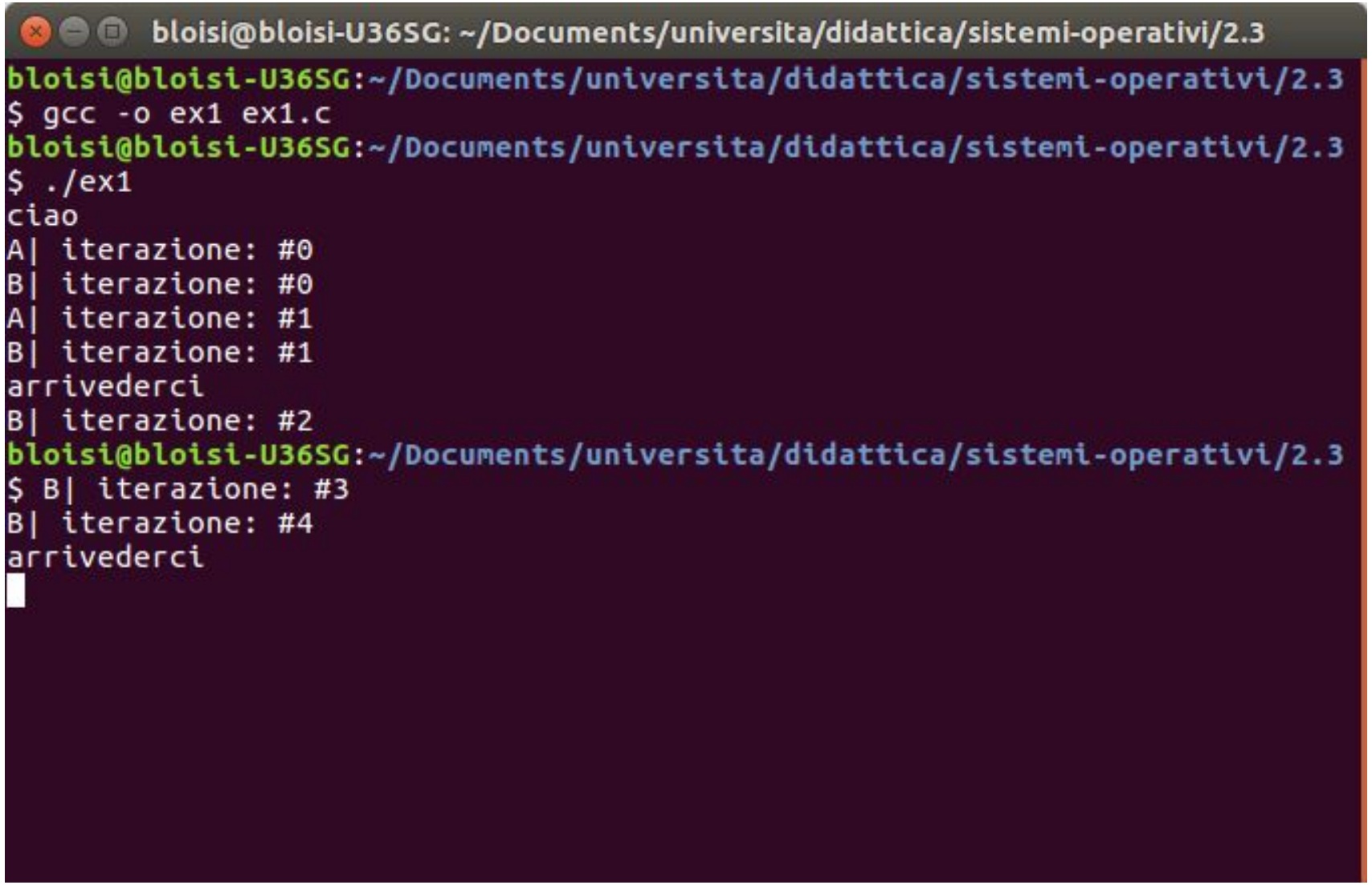
Il programma esegue una `fork()`, andando a creare un processo figlio duplicando, quindi, in memoria le variabili ed eventuali file descriptor aperti.
Poiché il processo genitore non effettua una `wait()`, esso non attenderà la terminazione del processo figlio.

Output del programma:

```
ciao
A| iterazione: #0
B| iterazione: #0
A| iterazione: #1
B| iterazione: #1
arrivederci
B| iterazione: #2
B| iterazione: #3
B| iterazione: #4
arrivederci
```



Soluzione Esercizio 1



Soluzione Esercizio 1

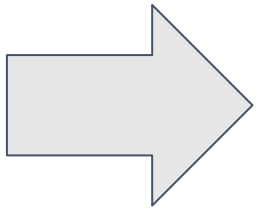
Poiché il genitore non effettua la `wait()`, una volta completato il suo ciclo, esso terminerà brutalmente, lasciando il figlio creato in precedenza orfano.

Il processo figlio verrà assegnato al processo master `init/systemd` che il primo processo generato all'avvio della macchina.

Esercizio 2

Cosa stampa il programma a lato in corrispondenza di
`/* LINEA A */` e
`/* LINEA B */` ?

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

# define NUM_STEPS 5

unsigned int value = 0;
pid_t pid;
pthread_t tid;

void* runner(void* param);

int main(int argc, char* argv []) {
    pthread_attr_t attr;
    pid = fork();

    if(pid < 0)
        return -1;
```



Esercizio 2

Cosa stampa il programma a lato in corrispondenza di `/* LINEA A */` e `/* LINEA B */` ?

```
    if(pid == 0) {
        pthread_attr_init(& attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf("linea A, valore = %d\n", value); /* LINEA A */
    }
    else {
        wait(NULL);
        printf("linea B, valore = %d\n", value); /* LINEA B */
    }
    return 0;
}

void* runner(void* param) {
    for(int s = 0; s < NUM_STEPS; ++s) {
        if(pid) { value++; }
    }
    return param;
}
```

Soluzione Esercizio 2

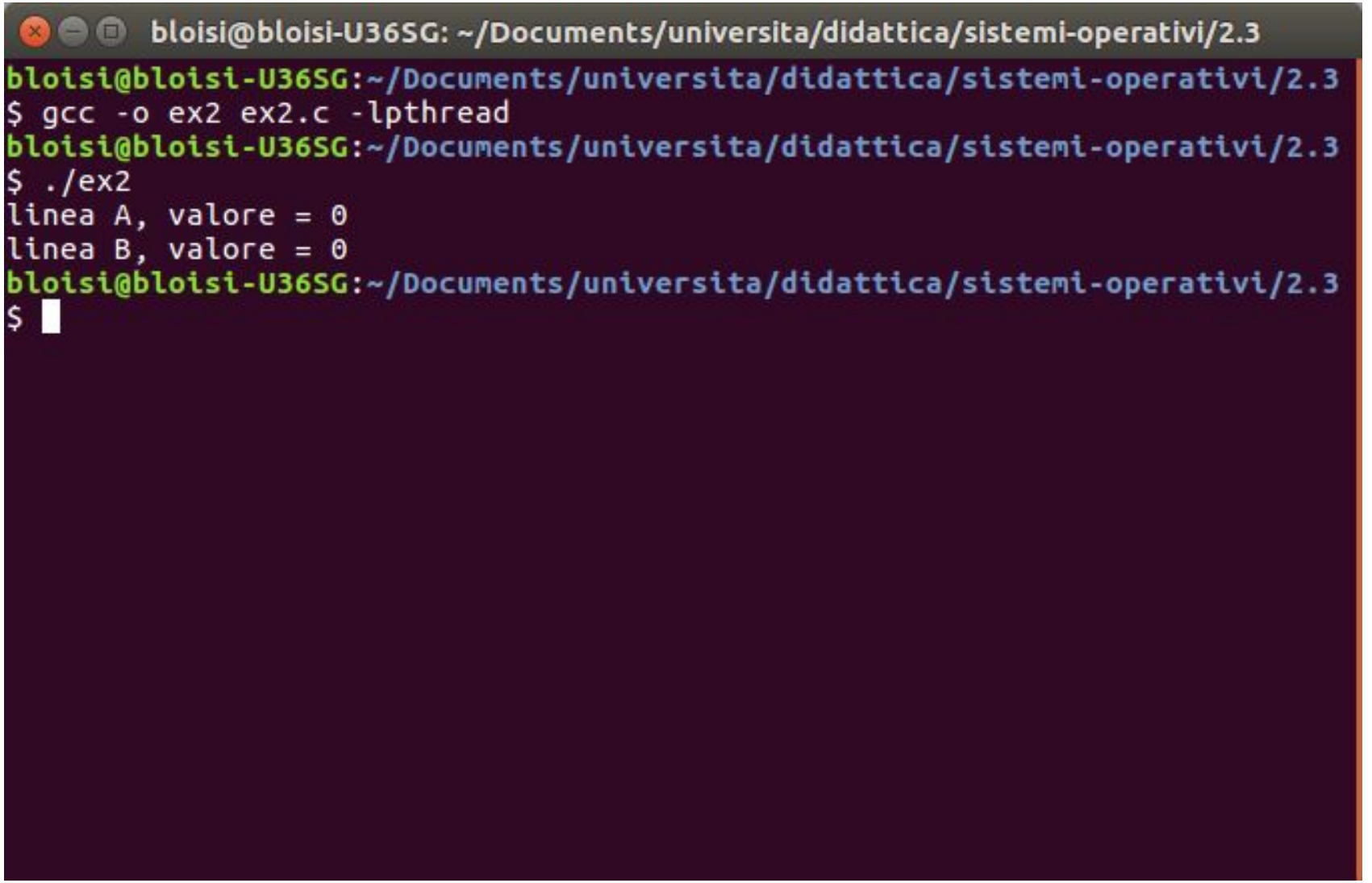
Il programma crea un processo figlio tramite fork, andando a duplicare le variabili del processo genitore. Inoltre, il figlio andrà a creare un nuovo thread, che incrementa - eventualmente - la variabile value.

Tuttavia, la funzione eseguita nel nuovo thread - `runner` - scriverà solo se la variabile pid sarà != 0, condizione mai verificata poiché il thread fa parte del processo figlio.

Date queste considerazioni, l'output del programma sarà il seguente:

```
linea A, valore = 0
linea B, valore = 0
```

Soluzione Esercizio 2



Esercizio 3

Indicare quale dei seguenti può essere un possibile output sulla shell per il programma a lato:

- A
- CDA
- CABEDE
- BA
- C
- BAC

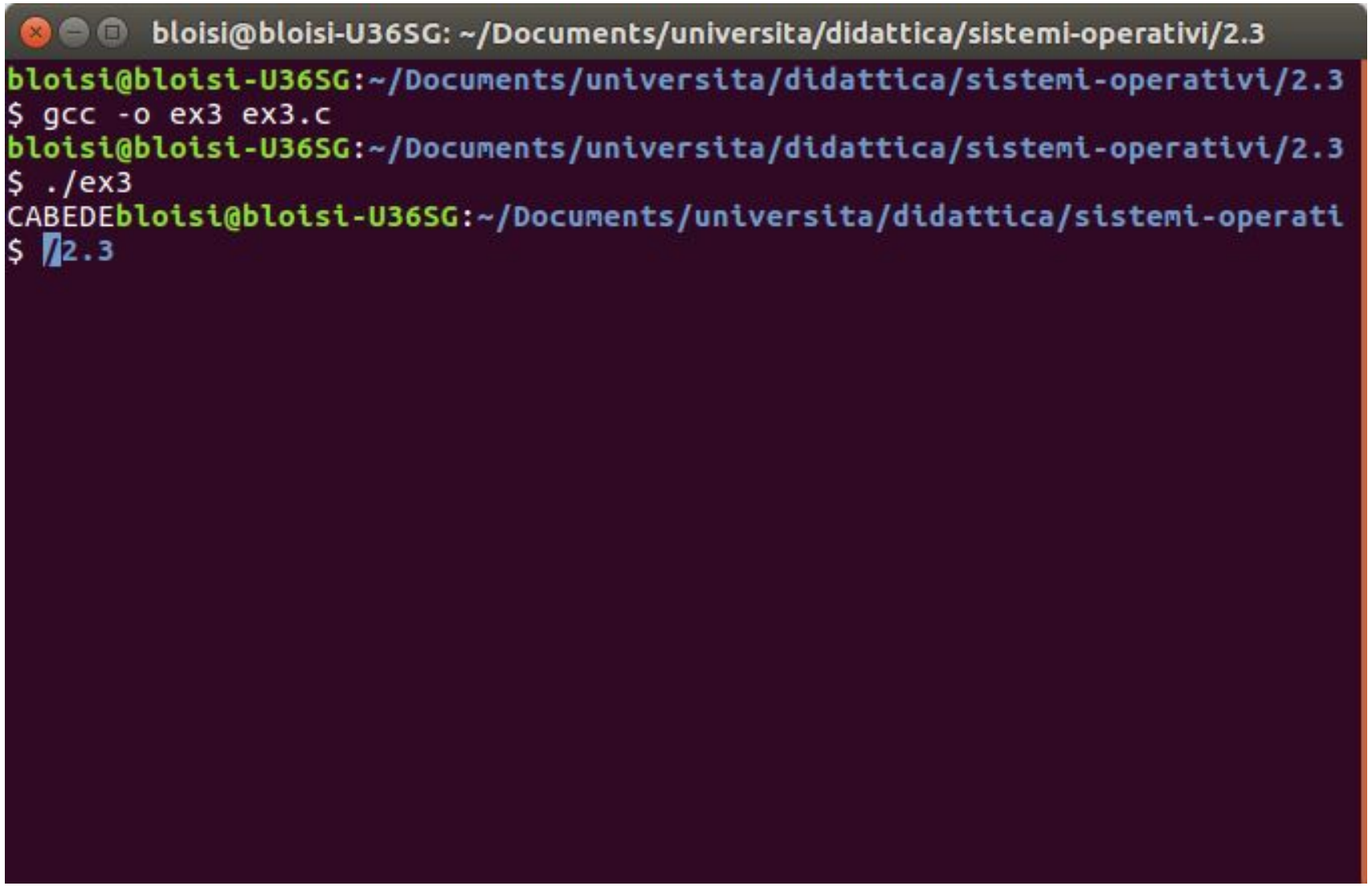
```
int main (int argc, char** argv) {
    if(fork() == 0) {
        if(fork() == 0) {
            printf ("A");
            return 0;
        }
        else {
            wait(NULL);
            printf("B");
        }
    }
    else {
        if(fork() == 0) {
            printf("C");
            exit(0);
        }
        else {
            wait(NULL);
        }
        wait(NULL);
        printf("D");
    }
    printf ("E");
    return 0;
}
```

Soluzione Esercizio 3

Tra quelli elencati, l'unico output valido sarà `CABEDE`, a causa delle wait posizionate in ogni processo padre.

Si noti che la `return` a riga 5 e la `exit` a riga 15 non interrompono l'intero programma, ma soltanto l'esecuzione dei processi figli in cui sono locate: il carattere 'E' quindi viene stampato.

Soluzione Esercizio 3



Domanda

Cosa contiene il Process Control Block (PCB) di un processo?

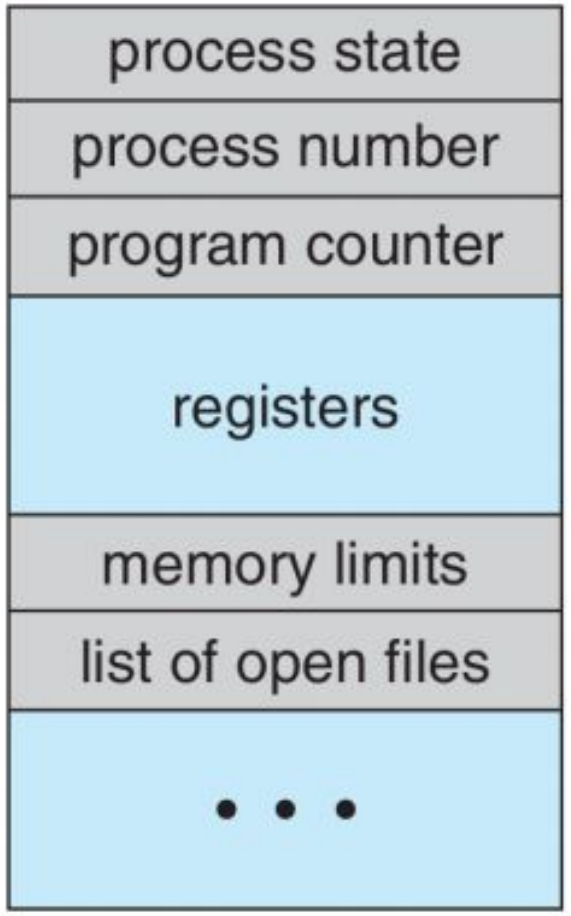
Risposta

Il PCB di un processo è una struttura dati che contiene tutte le informazioni relative al processo a cui è associato.

Esempi di informazioni contenute nel PCB sono:

- Stato del processo (running, waiting, zombie ...)
- Program Counter (PC), ovvero il registro contenente la prossima istruzione da eseguire
- Registri della CPU
- Informazioni sulla memoria allocata al processo
- Informazioni sull'I/O relativo al processo.

Una illustrazione di tale struttura dati è riportata qui a lato.

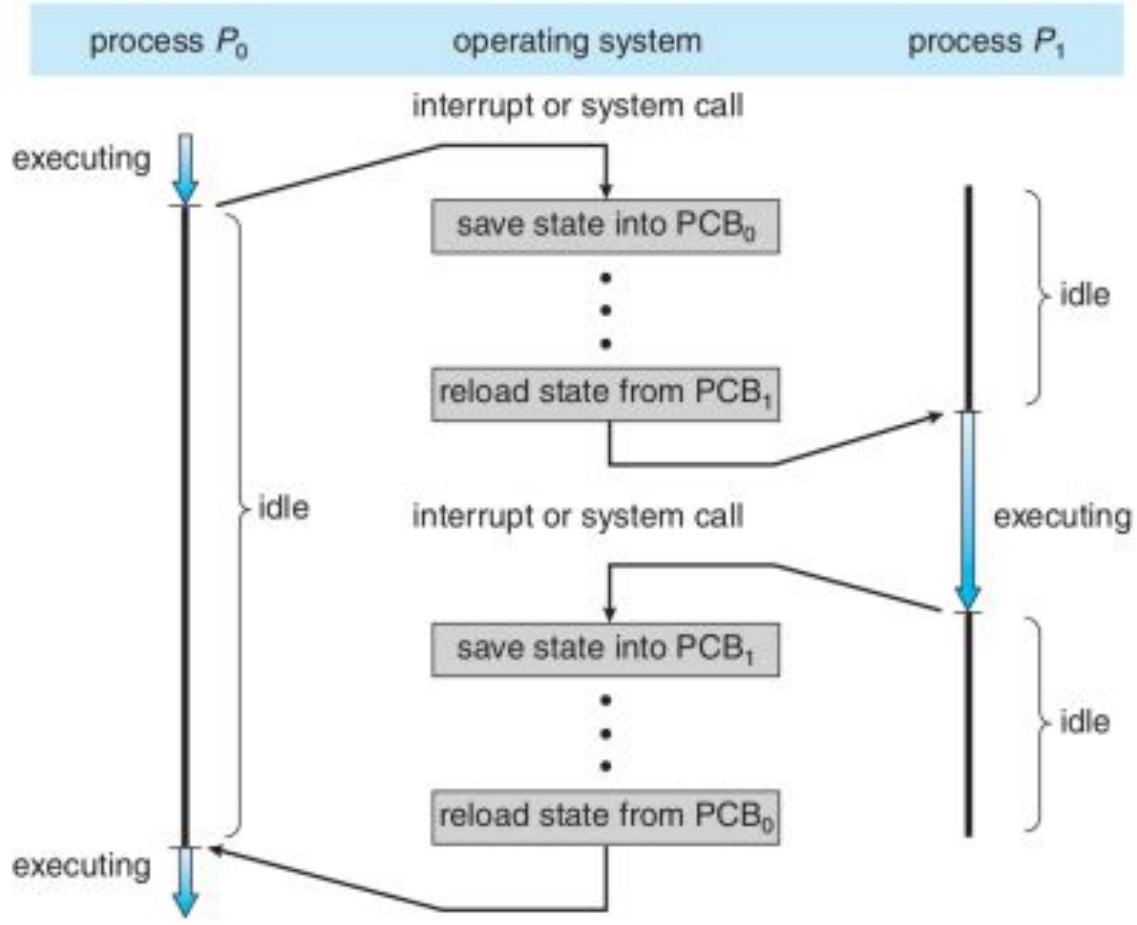


Domanda

Illustrare il meccanismo di Context Switch, avvalendosi di opportune illustrazioni grafiche.

Risposta

Supponiamo di avere due processi P_0 e P_1 e che il primo sia in stato “running”. Un context switch si verifica quando l’Operating System (OS), per mettere in esecuzione P_1, salva lo stato corrente di P_0 in modo da poterne ripristinare l’esecuzione in un secondo momento. L’esempio discusso può essere riassunto visivamente nella figura a lato.



Il context switch è fonte di overhead a causa delle varie operazioni necessarie a compiere lo scambio, le quali includono il salvataggio dello stato, il blocco e la riattivazione della pipeline di calcolo, lo svuotamento e il ripopolamento della cache.

Domanda

Che relazione c’è tra una system call, un generico interrupt e una trap? Sono la stessa cosa?

Risposta

Si tratta di tre concetti distinti:

- Una system call (o syscall) è una chiamata diretta al sistema operativo da parte di un processo di livello utente (ad esempio, una richiesta di I/O).
- Un interrupt è un segnale asincrono proveniente da hardware o software per richiedere la gestione immediata di un evento.
- Gli interrupt software sono definiti trap.

A differenza delle syscall, gli interrupt esistono anche in elaboratori privi di Sistema Operativo (ad esempio, in un microcontrollore). In seguito alla chiamata di una syscall, verrà generata una trap (interrupt software), in modo da poter richiamare l'opportuna funzione associata a tale syscall utilizzando la Syscall Table (ST).

Domanda

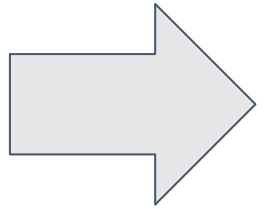
Spiegare brevemente cos’è il Direct Memory Access (DMA) e quando viene usato.

Risposta

Alcuni dispositivi necessitano di trasferire grandi quantità di dati a frequenze elevate. Effettuare tali trasferimenti tramite il protocollo genericamente usato per altri tipi di periferiche richiederebbe l'intervento della CPU per trasferire un byte alla volta i dati - tramite un processo chiamato Programmed I/O (PIO).
Ciò provocherebbe un overhead ingestibile per l'intera macchina, consumando inutilmente la CPU.

Per consentire il corretto funzionamento di dispositivi che necessitano di trasferire grandi quantità di dati a frequenze elevate, evitando gli svantaggi del PIO, si utilizzano controllori dedicati che effettuano DMA.

Nel DMA, i controllori scrivono direttamente sul bus di memoria. La CPU sarà incaricata soltanto di "validare" tale trasferimento e poi sarà di nuovo libera di eseguire altri task.



Risposta

Le periferiche che necessitano di trasferire grandi quantità di dati a frequenze elevate sono molto comuni ai giorni nostri e sono usate nella maggior parte dei dispositivi elettronici - pc, smartphones, server …

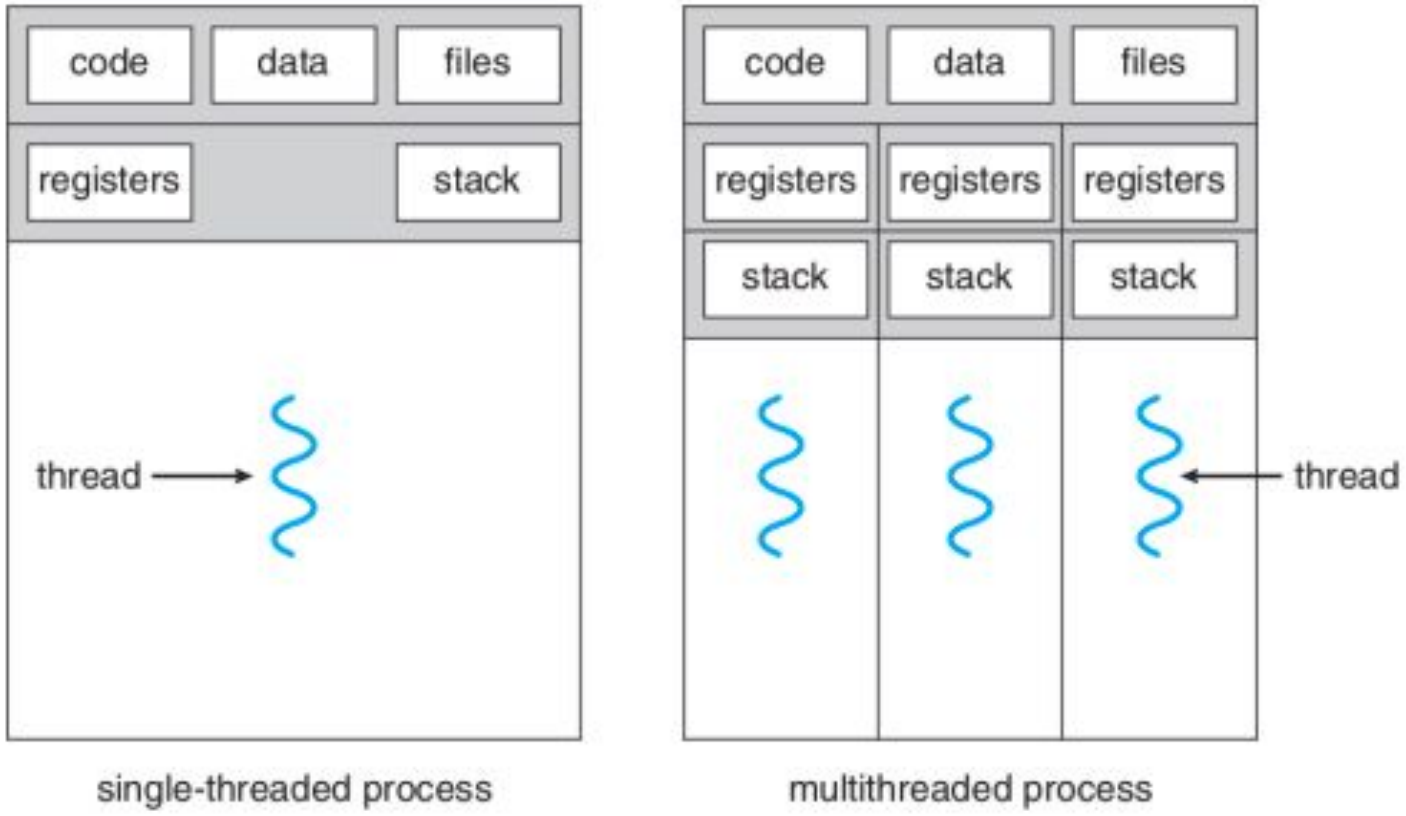
Esempi di periferica che si avvalgono di controller DMA sono videocamere, dischi, schede video, schede audio...

Domanda

Richiede meno risorse la creazione di un nuovo thread o di un nuovo processo? Si motivi la risposta, evidenziando quali risorse vengono utilizzate in entrambi i casi.

Risposta

Creare un thread - sia esso a livello kernel o utente - richiede l'allocazione di una data structure contenente il register set, lo stack e altre informazioni quali la priorità, come riportato nella figura in basso.



Risposta

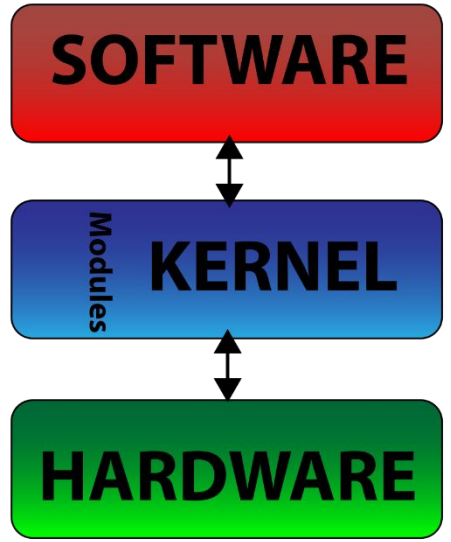
Creare un nuovo processo, invece, è una operazione relativamente costosa poiché richiede l'allocazione di un nuovo Process Control Block (PCB).

Il PCB contiene tutte le informazioni del processo, quali pid, stato del processo, informazioni sull'I/O, il Program Counter (PC) e la lista delle risorse aperte dal processo. Inoltre, il PCB include anche informazioni sulla memoria allocata dal processo.

In definitiva, quindi, la creazione di un thread richiede meno risorse rispetto a quelle necessarie per la creazione di un nuovo processo.

UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

Corso di Sistemi Operativi
A.A. 2019/20



Esercitazione
Gestione Processi

Docente:
Domenico Daniele Bloisi

Ottobre 2019