A

Sistemi Operativi L-A Prof. Anna Ciampolini

Prima prova parziale 21 Maggio 2004

1. [2]

Si consideri il comando di shell per UNIX:

```
ls -l | cut -c1-10 | grep r-x
```
[a]

A [**F**] l'output di questo comando viene salvato su un file del direttorio corrente

B [**F**] se esiste un file nel direttorio corrente su cui ho i diritti di lettura, ed eseguo questo comando, l'output del comando conterrà almeno una riga

C [**F**] per poter eseguire questo comando, devo poter accedere al direttorio corrente con i diritti di scrittura

D [**V**] l'output di questo comando è costituito da un numero variabile di righe di 10 caratteri

E [**F**] se non esistono file nel direttorio corrente su cui ho i diritti di lettura, ed eseguo questo comando, il comando non visualizzerà alcuna riga di output

[a] **Nota:** Esempio di output del comando `ls -l`:

```
-rw-rw-r-- 1 ghedo admins 22994 May 19 14:49 f.c
```

2. [2]

Sia dato un comando `modifica-f1`, il cui proprietario è l'utente `root` (appartenente all'omonimo gruppo `root`). L'utente `root` sia anche il proprietario di un file chiamato `f1`. Infine, sia dato un utente `janet`, appartenente al gruppo `users`, e proprietario di un comando `erase`. Si supponga che le impostazioni di protezione siano:

i) per il comando `modifica-f1`:
- diritti di esecuzione a tutti gli utenti;
- SUID impostato a vero;

ii) per il file `f1`:
- diritti di scrittura solo al proprietario;

iii) per il comando `erase`:
- diritti di esecuzione a tutti gli utenti.

A [**V**] Il comando `modifica-f1`, eseguito da `root`, è abilitato a modificare il file `f1`

B [**V**] Il comando `modifica-f1`, eseguito da `janet`, è abilitato a modificare il file `f1`

C [**V**] L'utente `janet` è abilitato a eseguire `modifica-f1`

D [**V**] Il processo che esegue `modifica-f1` ha gli stessi diritti dell'utente `root`

E [**F**] Se l'utente `janet` esegue `modifica-f1`, una volta che `modifica-f1` termina, tale utente possiede i diritti di scrittura su `f1`

3. [3]

Si consideri la seguente porzione di codice di un programma C:

```
int pid1,pid2,k=0;
pid1 = fork();
k++;
pid2 = fork();
if (pid2==0) {
  printf("Should I stay?\n");
}
if (k>0) printf("Should I go?\n");
```

In condizioni di funzionamento ideale (senza errori nelle `fork`, `printf`, ...):

A [**V**] oltre al processo padre, vengono creati altri tre processi

B [**F**] il testo `Should I stay?` viene stampato una volta

C [**V**] la prima linea visualizzata in output può contenere il testo `Should I go?` oppure il testo `Should I stay?` (a seconda dell'esecuzione concorrente dei processi)

D [**V**] esistono due processi con lo stesso valore di `pid1`

E [**F**] oltre al processo padre, vengono creati altri due processi

4. [3]

Si consideri la seguente porzione di codice di un programma C:

```
int pid1,pid2,s=0,
    funzioneA(void),
    funzioneB(void);
pid1 = fork();
if (pid1==0) {
  pid2 = fork();
  if (pid2==0) {
    printf("%d",s);
    exit(0); }
  else {
    wait(&s);
    funzioneA();
    exit(1); }
}
wait(&s);
funzioneB();
```

In condizioni di funzionamento ideale (senza errori nelle `fork`, funzioni varie, ...e assumendo che `funzioneA` e `funzioneB` non modifichino il valore di `s`):

A [**V**] esiste un solo processo con `pid1` diverso da zero, ed esso termina per ultimo

B [**V**] `funzioneB` viene invocata sempre dopo che `funzioneA` è terminata

C [**V**] `funzioneA` viene invocata una e una sola volta

D [**F**] `funzioneA` viene invocata sempre prima che sia stata eseguita la `printf`

E [**F**] esiste un solo processo che termina la propria esecuzione con il valore della variabile `s` pari a zero

5. [3]

Si consideri la seguente porzione di codice di un programma C: [a]

```
int pid1,pid2;
pid1 = fork();
if (pid1==0) {
  printf("Brad!\n");
  execlp("./c1","c1","-a",NULL);
  exit(1);
}
pid2 = fork();
if (pid2==0) {
  execlp("./c1","c1","-b",NULL);
}
printf("Janet!\n");
```

Si supponga che l'unica istruzione che può "fallire" sia la `execlp`, e che per il resto tutto funzioni senza errori (`fork`, `exit`, `printf`, ...).

A [**V**] è possibile che un processo con `pid2` pari a zero (e `pid1` diverso da zero) stampi `Janet!`

B [**F**] la stringa `Janet!` viene sicuramente visualizzata più di una volta

C [**F**] la stringa `Janet!` viene visualizzata dal processo "padre" prima che gli altri processi terminino

D [**F**] è possibile che il processo che stampa `Brad!` stampi anche `Janet!`

E [**V**] in assenza di errori nella `execlp`, non verranno creati più di due processi (olre al processo padre)

[a] **Nota:** Si supponga che `c1` sia un comando eseguibile, presente nel direttorio corrente, e che lo si possa invocare con delle opzioni del tipo `-a`, `-b`, ...

6. [3]

Si consideri l'immagine di un processo P nel sistema operativo Unix.

A [**V**] La *User Structure* di P è necessaria quando P entra nello stato *running*.

B [**F**] La *Process Structure* di P non è necessaria quando P è sospeso (cioè è nello stato *sleeping*).

C [**F**] La *Process Structure* di P contiene il puntatore al codice eseguito da P.

D [**F**] La *Process structure* di P contiene i pid di tutti i processi figli (generati da P).

E [**V**] La *Process structure* di P contiene il pid del processo padre.

7. [3]

Si consideri un sistema operativo multiprogrammato, non *time sharing*.

A [**V**] È possibile che un processo utilizzi la CPU per un intervallo di tempo arbitrariamente lungo.

B [**F**] Solo un processo alla volta può trovarsi nello stato di *ready*.

C [**V**] Un processo può attendere l'assegnazione della CPU per un intervallo di tempo arbitrariamente lungo.

D [**F**] Un processo può passare dallo stato *running* allo stato ready.

E [**F**] Un processo che esce dallo stato *running* entra sempre nello stato *terminated*.

8. [3]

Si considerino i thread $t1$ e $t2$ appartenenti al task Ta, e $t3$ e $t4$ appartenenti al task Tb.

A [**V**] Se il sistema operativo è multiprogrammato a divisione di tempo (*time sharing*), è possibile che, se $t1$ si sospende indefinitamente, $t2$ attenda indefinitamente l'assegnazione della CPU.

B [**F**] Se $t1$ si sospende indefinitamente, allora $t3$ attende indefinitamente l'assegnazione della CPU.

C [**F**] Il cambio di contesto tra $t1$ e $t2$ è più costoso nel caso in cui i thread siano realizzati a livello *user*, rispetto al caso di realizzazione a livello *kernel*.

D [**F**] Il cambio di contesto tra $t1$ e $t4$ è più costoso nel caso in cui i thread sono realizzati a livello *user*, rispetto al caso di realizzazione a livello *kernel*.

E [**F**] Se il sistema operativo è multiprogrammato a divisione di tempo (*time sharing*), la terminazione di $t2$ implica la terminazione del task Ta.

9. [3]

Si consideri un sistema operativo S e la sua organizzazione interna.

A [**V**] Se l'organizzazione interna di S è basata su *microkernel*, S può essere esteso piu' facilmente rispetto ad un sistema monolitico.

B [**F**] Se l'organizzazione interna di S è basata su *microkernel*, il costo medio di interazione tra le diverse componenti del sistema S è più basso rispetto al caso di sistema monolitico.

C [**F**] Se l'organizzazione interna di S è basata su *microkernel*, tutte le funzionalità offerte dal sistema operativo sono realizzate completamente dal kernel.

D [**V**] Se l'organizzazione interna di S è basata su *kernel monolitico*, la richiesta di un servizio a S avviene sempre mediante system call.

E [**V**] Se l'organizzazione di S è modulare a livelli, l'efficienza del sistema può decrescere al crescere del numero dei livelli.

10. [3]

Si considerino due processi interagenti $P1$ e $P2$:

A [**F**] Nel modello ad ambiente locale l'interazione tra $P1$ e $P2$ può non richiedere l'intervento del sistema operativo.

B [**V**] Nel modello ad ambiente locale, se $P1$ e $P2$ interagiscono mediante comunicazione indiretta, è possibile che il mittente non conosca l'identificatore del destinatario.

C [**V**] Nel modello ad ambiente locale, se $P1$ e $P2$ interagiscono mediante comunicazione diretta, è possibile che il destinatario non conosca l'identificatore del mittente.

D [**V**] Nel modello ad ambiente locale, se $P1$ e $P2$ interagiscono mediante comunicazione indiretta, è necessario utilizzare una mailbox.

E [**V**] Nel modello ad ambiente locale, se il canale di comunicazione ha capacità nulla, la comunicazione tra $P1$ e $P2$ avviene sempre in modo sincrono.

11. [3]

Si consideri la seguente applicazione concorrente (scritta in pseudo-C), composta da 2 processi, nell'ipotesi di modello ad ambiente globale:

```
#define N1 0
#define N2 0
shared semaphore s1,s2;
shared int V=0;
s1.value=N1;
s2.value=N2;
```

```
/* Processo P1 */
int K=1; /* var.locale a P1*/

main() {
    signal(&s1);
    while(K)
    {   wait (&s1);
        if (V< 3)
        {   printf("%d\n", V);
            signal(&s2);
        }
        else
        { K=0;
          printf("%d\n", V);
       }
    }
}
```

```
/* Processo P2 */
int K=2;/* var.locale a P2*/

main() {
    while(K)
    {   wait (&s2);
        if (K>1)
        {   V+=2;
            printf("%d\t%d\n",V,K--);
        }
        else
        {   V++;
            printf("%d\t%d\n",V,K--);
        }
   signal(&s1);
    }
}
```

A [**F**] Il processo P2 viene sospeso indefinitamente.

B [**V**] L'output contiene la stringa `2 2`.

C [**V**] Il processo P1 esegue 3 `printf`.

D [**F**] L'output contiene la stringa `3 2`.

E [**F**] Il processo P1 può non terminare.