Caselle riservate

Ex. 1	
Ex. 2	
Ex. 3	
Ex. 4	
Ex. 5	
Ex. 6	
Tot.	

Sistemi Operativi

Compito d'esame

27 Febbraio 2019

Matricola ______________ Cognome ______________________ Nome ______________________

Docente: ◯ Quer ◯ Sterpone

Non si possono consultare testi, appunti o calcolatrici a parte i formulari distribuiti dal docente. Risolvere gli esercizi negli spazi riservati. Fogli aggiuntivi sono permessi solo quando strettamente necessari. Riportare i passaggi principali.
Durata della prova: 100 **minuti.**

!strcmp( ) ⟺ strcmp( ) == Ø

1. Si supponga che dal programma successivo

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main (int argc, char *argv[]) {
  int i = 2;
  if (!strcmp (argv[0], "/home/quer/p")) {
    i++;
    fprintf (stdout, "exec 1 %d\n", i);
    execlp ("echo", "./p", "/home/quer/p", "i", NULL);
  }
  if (!strcmp (argv[0], "/home/quer/20190227")) {
    fprintf (stdout, "exec 2 %d\n", i);
    execlp ("./p", "/home/quer/p", "/home/quer/20190227", "++i", NULL);
  }
  fprintf (stdout, "i = %d\n", i);
  return (1);
}
```

Attivato da riga 3)

Attivato da riga 1)

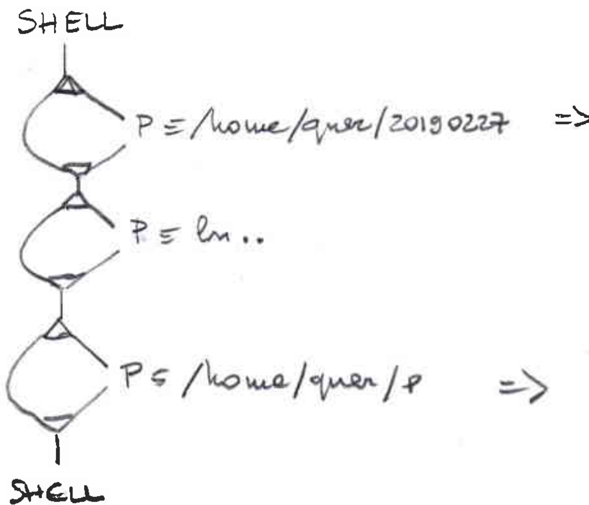
sia stato ottenuto l'eseguibile di nome `/home/quer/20190227`.

Si supponga inoltre di eseguire lo script successivo:

```
1) /home/quer/20190227
2) ln /home/quer/20190227 p
3) /home/quer/p
```

Il link è comunque inutilizzato perché la riga 3) attiva la execlp con la echo e la riga 1) che attiverebbe ./p viene eseguita prima

Si riporti il grafo di controllo del flusso e l'albero di generazione dei processi a seguito della sua esecuzione. Si indichi inoltre che cosa esso produce su video e per quale motivo.



=> PRINT "exec 2 2↵"
execlp FALLISCE (p non è ancora stato creato)
PRINT "i = 2↵"
return

=> PRINT "exec 1 3↵"
execlp → sostituisce il P con la echo
ECHO "/home/quer/p i↵"
termina

OUTPUT:
exec 2 2
i = 2
exec 1 3
/home/quer/p i

2. Si illustrino le caratteristiche delle *pipe* per la comunicazione e la sincronizzazione tra processi.

Se ne illustri inoltre l'utilizzo risolvendo il seguente problema. Due processi P_1 e P_2 desiderano scambiarsi grandi quantità di dati tramite la scrittura e la lettura dello stesso file sincronizzando le proprie operazioni sul file mediante l'utilizzo di una o più pipe. Il file memorizza un insieme indefinito di stringhe in ragione di una stringa su ciascuna riga. Ciascun processo:

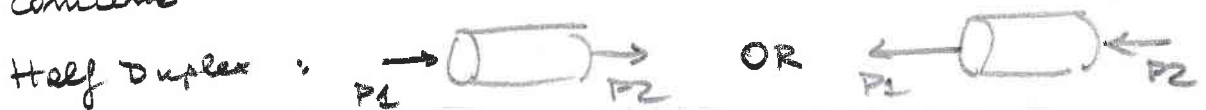
- legge il contenuto del file una stringa alla volta e visualizza le stringhe a video.
- riscrive completamente il file memorizzandoci un insieme di stringhe lette da tastiera. Le stringhe vanno lette da tastiera una alla volta. La lettura viene terminata dall'introduzione della stringa "end".

Il protocollo deve essere tale per cui quando P_1 è in esecuzione P_2 attende e viceversa. Entrambi i processi devono terminare in maniera ordinata quando da tastiera viene introdotta la stringa "END" (maiuscola).

IPC = Inter Process Communication
- Memorie condivise
- Scambio messaggi

PIPE = Half Duplex pseudo file

Permettono flusso dati tra due processi con antenato comune



Uso delle system call : pipe, read, write

read → bloccante a pipe vuota

write → bloccante a pipe piena (64KByte)

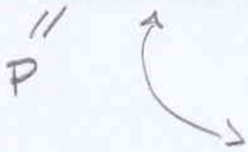
```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>

#define N 132

void parent (char *, int [],int []);
void child (char *, int [],int []);
void read_file (char *);
int write_file (char *);

int main (int argc, char **argv) {
  int fd1[2], fd2[2];
  int childPid;

  pipe (fd1);
  pipe (fd2);

  childPid = fork();
  if (childPid == 0) {
    child (argv[1], fd1, fd2);
  } else {
    parent (argv[1], fd1, fd2);
  }

  wait ((void *)0);
  exit (1);
}

void  parent (char *name, int fd1[2], int fd2[2]) {
  char c;
  int stop = 0;

  close (fd1[0]); //close read end
  close (fd2[1]); //close write end

  while (stop==0) {
    printf ("PARENT WAKE UP (PID=%d) \n", getpid());
    read_file (name);
    stop = write_file (name);

    c = (stop==0) ? '0' : '1';
    write (fd1[1], &c, sizeof (char));
    if (stop==1)
      break;
    read (fd2[0], &c, sizeof (char));
    stop = (c=='0') ? 0 : 1;
  }

  close (fd1[1]);
  close (fd2[0]);

  return;
}

void child (char *name, int fd1[2], int fd2[2]) {
  char c;
  int stop = 0;

  close (fd1[1]); //close write end
  close (fd2[0]); //close read end
```

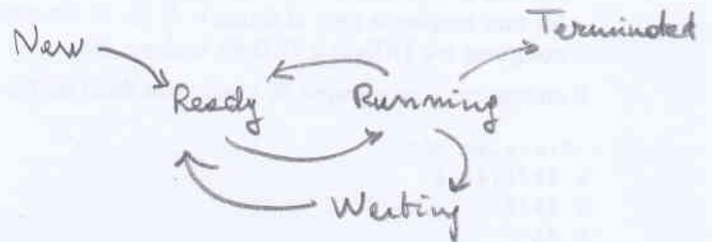
```
  while (stop==0) {
    read (fd1[0], &c, sizeof (char));
    printf ("CHILD WAKE UP (PID=%d) \n", getpid());
    if (c=='1')
      break;

    read_file (name);
    stop = write_file (name);
    c = (stop==0) ? '0' : '1';
    write (fd2[1], &c, sizeof (char));
  }

  close (fd1[0]);
  close (fd2[1]);

  return;
}

void read_file (char *name) {
  FILE *fp;
  char str[N];

  fp = fopen (name, "r");
  while (fscanf (fp, "%s", str) != EOF) {
    fprintf (stdout, "   %s\n", str);
  }
  fclose (fp);

  return;
}

int write_file (char *name) {
  FILE *fp;
  char str[N];

  fp = fopen (name, "w");
  do {
    fprintf (stdout, "   str: ");
    scanf ("%s", str);
    if (strcmp (str, "end") != 0 && strcmp (str, "END") != 0) {
      fprintf (fp, "   %s\n", str);
    }
  } while (strcmp (str, "end") != 0 && strcmp (str, "END") != 0);
  fclose (fp);

  if (strcmp (str, "END") == 0)
    return 1;
  else
    return 0;
}
```

3. Si indichino le principali differenze tra processi e thread. Si descriva la gestione di tali entità da parte del sistema operativo (struttura in memoria, etc.) e se ne riportino le caratteristiche principali con i relativi vantaggi e svantaggi. Si indichino infine le principali differenze tra thread a livello utente e thread a livello kernel. Che implicazioni hanno i due modelli sulle politiche di scheduling?

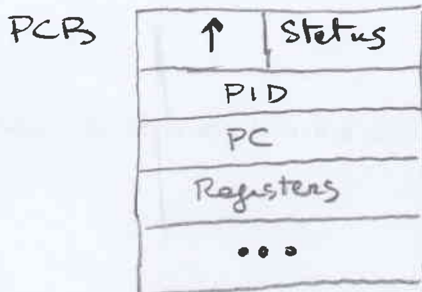
Programma ≜ entità passiva (stack, heap, etc.)

① Processo ≜ entità attiva
raggruppa risorse
stato processo =>





PCB

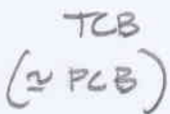


Memoria / Spazio indirizzamento
DUPLICATO

T

PID ...

② Thread ≜ processo leggero
raggruppa unità di schedulazione schedulate separatamente
condividono risorse: codice, dati
NON condividono PC, registri, stack
Più efficienti eseguirli (tempo e memoria) dei processi
Richiedono tecniche di protezione più accurate



③ Kernel level T: gestiti dal kernel tramite system call. Per ogni T il SO mantiene un TCB globale. Lo scheduling è deciso dal SO. Un T bloccato non blocca il processo. Costosi e in numero limitato.

④ User level T: gestiti nello spazio utente da programma applicativo. La tabella dei T è locale a ogni processo. Efficienti (no system call) ma il SO non ne è a conoscenza. T bloccato blocca il P. Scheduling personalizzabile per ogni P.

4. Si implementi uno script BASH di nome `isto.sh` in grado di:
   - Ricevere il nome di un direttorio sulla riga di comando.
   - Controllare il corretto passaggio dei parametri ricevuti, segnalando un errore e terminando in caso di errore.
   - Verificare che il direttorio specificato esista, segnalando un errore e terminando in caso contrario.
   - ¡ Determinare la dimensione di tutti i file regolari contenuti nel direttorio specificato sulla riga di comando e in tutti i suoi sotto-direttori.
   - Visualizzare un istogramma a barre orizzontali composte dal carattere "#" in cui la barra di posizione n rappresenta il numero di file di dimensione compresa tra $(n-1)$KByte e (n)KByte incluso. Ovvero la prima barra ha una lunghezza pari al numero di file di dimensione compresa tra 0KByte e 1KByte incluso, la seconda quelli compresi tra 1KByte e 2KByte incluso, etc.

   Il successivo è un esempio di esecuzione dello script:

```
./isto.sh dir
1 #########
2 ####
5 ##
9 #####
...
```

   Si noti che le barre dell'istogramma di dimensione nulla (e.g., 3, 4, etc.) non vanno visualizzate.

```bash
#!/bin/bash

#
# Exercise 4 of 27/02/2019
# Launch with: ./isto.sh <directory>
#

# Check arguments
if [ $# -ne 1 ]; then
    echo "Usage: isto.sh <directory>"
    exit 1
fi

# Check directory
if [ ! -d $1 ]; then
    echo "Directory $1 does not exists"
    exit 1
fi

# Get sorted list of file sizes
find $1 -type f -exec ls -l \{} \; | cut -d " " -f 5 | sort -n > "tmp.txt"

# Print istogram
bar=0
while read size; do
    if [ $(($size/1024 + 1)) -gt $bar ]; then
        if [ $bar -gt 0 ]; then
            echo ""
        fi
        bar=$(($size/1024 + 1))
        echo -n -e "$bar\t"
    fi
    echo -n "#"
done < "tmp.txt"
echo ""

# Delete temporary file
rm -f "tmp.txt"
```

```bash
#!/bin/bash

#
# Exercise 4 of 27/02/2019
# Launch with: ./isto.sh <directory>
#

# Check arguments
if [ $# -ne 1 ]; then
    echo "Usage: isto.sh <directory>"
    exit 1
fi

# Check directory
if [ ! -d $1 ]; then
    echo "Directory $1 does not exists"
    exit 1
fi

# Get the list of regular file in the given directory tree
find $1 -type f > "tmp.txt"

# Compose istogram
bar=0
while read line; do
    #size=$(wc -c < "$line")
    size=$(ls -l "$line" | cut -d " " -f 5)
    #bar=$(($size/1024 + 1))
    let "bar=(size/1024)+1"
    isto[$bar]=${isto[$bar]}"#"
done < "tmp.txt"

# Print istogram
for bar in "${!isto[@]}"; do
    echo -e "$bar\t${isto[$bar]}"
done

# Delete temporary file
rm -f "tmp.txt"
```

5. **Per i candidati iscritti al corso nell'anno accademico 2018–2019.**
La sequenza di Fibonacci $F(n)$ è tale per cui:

$$
\begin{aligned}
F(1) &= 1 \\
F(2) &= 2 \\
F(i) &= F(i-1) + F(i-2) \qquad \text{con i} > 2
\end{aligned}
$$

Ad esempio, la sequenza di Fibonacci per i primi $n = 7$ numeri è la seguente:

1 1 2 3 5 8 13

Si scriva un programma concorrente multi-thread in grado di:

- ricevere il numero n sulla riga di comando
- generare esattamente n thread, in modo che il thread di posizione i generi il numero di Fibonacci di posizione i (utilizzando il valore calcolato dai thread di posizione $(i-1)$ e $(i-2)$) e lo visualizzi
- sincronizzare i thread in modo da ottenere il comportamento desiderato.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

sem_t *sem;
int *fib;

//static void *fib_thread_up (void *);
static void *fib_thread_down (void *);
static void *fib_thread_up (void *);

int main (int argc, char **argv) {
  int n;

  setbuf (stdout, 0);
  n = atoi(argv[1]);
  fib = (int *) malloc (n * sizeof (int));
  if (fib==NULL) {
    fprintf (stderr, "Allocation error.\n");
    exit (0);
  }

  // Solution 1
  {
  int i, *index;
  pthread_t *th;

  sem = (sem_t *) malloc ((n+1) * sizeof (sem_t));
  th = (pthread_t *) malloc (n * sizeof (pthread_t));
  index = (int *) malloc (n * sizeof (int));
  if (sem==NULL || th==NULL || fib==NULL || index==NULL) {
    fprintf (stderr, "Allocation error.\n");
    exit (0);
  }
  sem_init (&sem[0], 0, 1);    // First thread to start
  for (i=1; i<n; i++) {
    sem_init (&sem[i], 0, 0);
  }
  sem_init (&sem[n], 0, 0);    // This thread
  fprintf (stdout, "Solution 1 (from 1 to n ... up)\n");
  for (i=0; i<n; i++) {
    index[i] = i;
    pthread_create (&th[i], NULL, fib_thread_up, (void *) &index[i]);
  }
  sem_wait (&sem[n]);
  fprintf (stdout, "\n");

  for (i=0; i<n; i++) {
    sem_destroy (&sem[i]);
  }
  free (sem);
  free (th);
  free (index);
  }

  // Solution 2
  {
  pthread_t th;

  fprintf (stdout, "Solution 2 (from n to 1 ... down)\n");
  pthread_create (&th, NULL, fib_thread_down, (void *) &n);
  pthread_join (th, NULL);
  fprintf (stdout, "\n");
  }
```

IL PROGRAMMA INCLUDE 2 SOLUZIONI, UNA CHE UTILIZZA I SEMAFORI E UNA NO

```
  free (fib);

  return 1;
}

static void *fib_thread_up (void *arg) {
  int *p, n;

  p = (int *) arg;
  n = *p;

  sem_wait (&sem[n]);
  if (n < 0) {
    fprintf (stdout, "Error (n=%d)\n", n);
  } else {
    if (n==0 || n==1) {
      fib[n] = 1;
    } else {
      fib[n] = fib[n-2] + fib[n-1];
    }
  }
  fprintf (stdout, "%d ", fib[n]);
  sem_post (&sem[n+1]);

  pthread_exit (NULL);
}

static void *fib_thread_down (void *arg) {
  int *p, n, tmp;
  pthread_t th;

  p = (int *) arg;
  n = *p;
  if (n <= 0) {
    fprintf (stdout, "Error (n=%d)\n", n);
  } else{
    if (n==1) {
      fib[n] = 1;
    } else {
      if (n==2) {
        fib[n] = 1;
        tmp = n - 1;
        pthread_create (&th, NULL, fib_thread_down, (void *) &tmp);
        pthread_join (th, NULL);
      } else {
        tmp = n - 1;
        pthread_create (&th, NULL, fib_thread_down, (void *) &tmp);
        pthread_join (th, NULL);
        fib[n] = fib[n-2] + fib[n-1];
      }
    }
  }
  fprintf (stdout, "%d ", fib[n]);
  pthread_exit (NULL);
}
```

Per i candidati iscritti al corso prima dell'anno accademico 2018–2019.
Un server registra le operazioni di autenticazione da parte dei propri utenti mediante un file di log costituito da messaggi (uno per riga) aventi il seguente formato:

```
Data Ora [Oggetto] Result Username IPaddress
```

dove `Data` e `Ora` sono stringhe contenenti rispettivamente data e ora del messaggio, `Oggetto` è un codice univoco che rappresenta l'operazione oggetto del messaggio (tale codice inizia per "register" nel caso delle registrazioni ed inizia per "login" nel caso dei tentativi di accesso), `Result` è l'esito dell'operazione (`SUCCESS` nel caso di successo, `FAIL` nel caso di errore), `Username` è il nome dell'utente che ha richiesto l'operazione e `IPaddress` è l'indirizzo IP da cui è avvenuta la richiesta.

Un esempio di tale file di log è riportato di seguito. Supporre che il formato del file di log sia sempre corretto.

Scrivere uno script AWK che, ricevuto in input il percorso del file di log, permetta di rilevare tutti quei i casi in cui, per un utente correttamente registrato, siano avvenuti due o più tentativi di login errati nello stesso giorno da parte di indirizzi IP diversi da quello di registrazione. Per ognuno di questi casi lo script deve stampare su output un messaggio di warning contente il numero di tali tentativi di login, il nome dell'utente ed il giorno in cui questi sono stati effettuati.

Ad esempio, dato il seguente input:

```
27/02/2019 14:30:00 [register.plain] FAIL failguy 55.44.22.3
27/02/2019 14:30:00 [register.oauth] SUCCESS okguy 94.103.22.8
27/02/2019 14:40:00 [login.plain] FAIL okguy 94.103.22.8
27/02/2019 14:40:10 [login.oauth] SUCCESS okguy 94.103.22.8
27/02/2019 16:00:00 [login.fb] FAIL okguy 66.9.233.233
27/02/2019 16:50:00 [login.oauth] SUCCESS okguy 66.9.233.233
28/02/2019 09:00:00 [login.oauth] FAIL okguy 55.169.16.233
28/02/2019 09:00:00 [login.plain] FAIL okguy 55.169.16.233
28/02/2019 21:00:00 [register.oauth] SUCCESS goodguy 179.144.12.12
01/03/2019 16:00:20 [login.oauth] FAIL failguy 94.103.22.8
01/03/2019 14:40:00 [login.fb] FAIL failguy 94.103.22.8
01/03/2019 08:00:00 [login.oauth] SUCCESS goodguy 202.1.1.76
01/03/2019 16:40:00 [login.oauth] FAIL goodguy 202.1.1.76
01/03/2019 16:41:00 [login.plain] FAIL goodguy 202.1.1.76
```

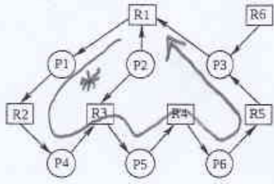
Lo script deve stampare in output:

```
Warning! 2 suspicious logins for user okguy in date 28/02/2019
Warning! 2 suspicious logins for user goodguy in date 01/31/2019
```

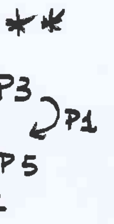
```
#!/usr/bin/awk

# To call like: awk -f es5.awk <file.c>

BEGIN {
  while(getline < ARGV[1]) {
    if ($3 ~ /^\[register/ && $4 == "SUCCESS") {
      ips[$5] = $6
    }
  }
}

$3 ~ /^\[login/ && $4 == "FAIL" {
  if ($5 in ips && $6 != ips[$5]) {
    logins[$5"-"$1]++
  }
}

END {
  for (userDay in logins) {
    split(userDay, res, "-")
    user = res[1]
    day = res[2]
    if (logins[userDay] >= 2) {
      printf ("Warning! %d suspicious logins for user %s in date %s\n",
              logins[userDay], user, day)
    }
  }
}
```

6. Si illustri che cosa si intende per "stato sicuro", "sequenza sicura" e (con un esempio) per "grafico delle traiettorie delle risorse".

Per gestire le condizioni di stallo, due sistemi operativi utilizzano rispettivamente il grafo riportato a sinistra e la tabella riportata a destra della seguente figura.



Processo	Fine	Assegnate			Massimo			Necessità			Disponibilità		
		R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	~~F~~ T	1	0	1	4	3	4	3	3	3 •	3	2	2
P_2	~~F~~ T	0	0	1	4	4	2	4	4	1 •	3	3	3
P_3	~~F~~ T	0	1	1	2	3	3	2	2	2 •	4	3	4
P_4	F	0	0	0	5	4	5	5	4	5	4	4	4
P_5	~~F~~ T	0	1	0	1	4	4	1	3	4 •	4	4	5



Considerando i due casi separatamente, si indichi se gli stati rappresentati sono sicuri e in caso lo siano si riporti una possibile sequenza sicura.

① STATO SICURO

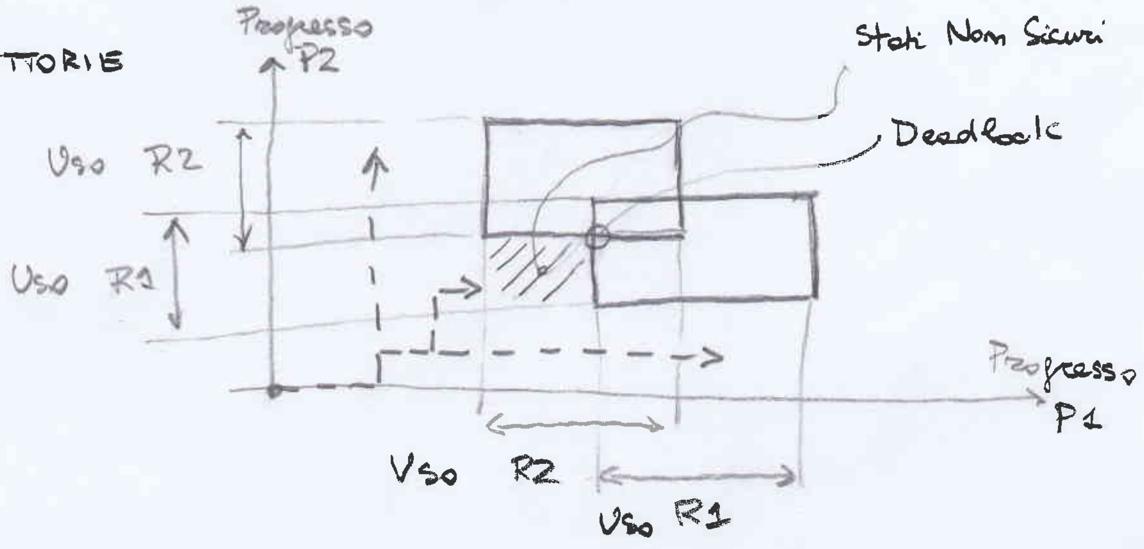
Il sistema è in uno stato dal quale è in grado di allocare le risorse a tutti i processi attivi e impedire il verificarsi di uno stallo.
Stato per cui esiste una sequenza sicura.

② SEQUENZA SICURA

Sequenza di schedulazione dei processi tale che le richieste di ciascun processo possano essere soddisfatte impiegando le risorse disponibili nello stato corrente sommate a quelle liberate dai processi già terminati

③ GRAFICO TRAIETTORIE (2D)

o progresso congiunto di 2P



Spiegazione...

④ GRAFO ALLOCAZIONE

le risorse sono unitarie
* = ciclo ⇒ vi è stallo e nessun processo può terminare. Lo stato NON è sicuro.

⑤ BANCHIERE

L'unica sequenza ** P3 → P1 → P5 → P2 possibile NON è completa (manca P4). Lo stato NON è sicuro.