Reserved cells

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

Operationg Systems

Examination task

Examples of questions of previous exams

ID number _____ Surname _____ Name _____

Professor: ◯ Scanzio

It is not possible to consult texts, notes or to use calculators. The only material allowed consists in the forms distributed by the professor. Solve the exercises in the reserved spaces. Additional sheets are permitted only when strictly necessary. Report the main steps for solving exercises.
Duration: 100 **minutes.**

1. **Exam 2019/02/01 Exercise 1:**
   Suppose that the hard disk of a small embedded system is composed if 24 blocks of 1 MByte each, which are numbered from 0 to 23. Suppose that the operating system keeps track of the free (occupied) blocks indicating them in a vector with the value 0 (1), and that the current situation of the disk is represented by the following vector:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1	0	0	0	0	1	0	0	1	1	1	0	1	0	0	1	0	1	0	0	1	0	0

   With reference to the file allocation methodologies *contiguous*, *linked*, *FAT* and *indexed*, indicate how the files `File1`, `File2` and `File3` (with dimension 2.4, 1.6 and 3.9 Mbyte, respectively) can be allocated.

   Schematically report the information stored in the entry directory and the relative pointers for each strategy.

EXAM 2019/02/01 (EX. 1)

1) CONTIGUOUS: Each file occupies a contiguous set of blocks

	Blocks	Dir. Start	Entry Length
File 1	2,3,4	2	3
File 2	7,8	7	2
File 3	NOT ALLOCABLE	/	/

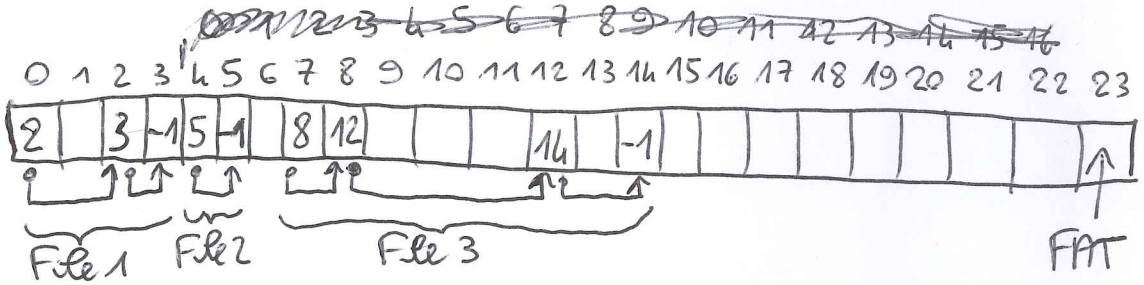
Using the algorithm: FIRST-FIT

2) LINKED: Each file is managed through a concatenated list of blocks. The references (pointers) are stored in the blocks themself.

	Blocks	Dir. Start	Entry end
File 1	0→2→3→-1	0	3
File 2	4→5→-1	4	5
File 3	7→8→12→14→-1	7	14

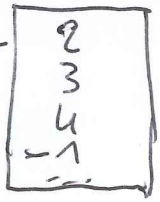
3) FAT: Like the previous, but references are stored in a FAT table. Assuming that the FAT is in block 23:

	DIR. Entry
File 1	0
File 2	4
File 3	7



4) INDEXED: For each file there is an index block containing the list of used blocks.

	Dir Entry	Index blocks
File 1	0	2, 3, 4, -1, ...
File 2	5	7, 8, -1, ...
File 3	12	14, 15, 17, 19, -1, ...



3. **Exam 2019/02/01 Exercise 4:**
   A file has the following format:

```
# prog1
f1.c
func1.c
main1.c
my_func.c
END
# prog3
main2.c
func2_2.c
my_func.c
END
...
```

   Implement a BASH script that:

- Receive the name of a file from command line. This file has the format previously indicated.
- Compile the files whose names are reported in the lines that do no begin with the character `#`. It must generate an executable whose name is reported in the line of the file that starts with the character `#` and precedes the name of the source file to be compiled. Lines containing the string `END` indicate the termination of the list of source files to be compiled.

   The script must also generate a directory named `log` (in the case it does not already exist), and copy in this directory the output of all the compilation commands, each in a file with the same name as the executable, but with extension `.log`.

   For example, by running the script on the previous source file, the following commands should be generated:

```
gcc -Wall -o prog1 f1.c func1.c main.c my_func.c
gcc -Wall -o prog3 main2.c func2_2.c my_func.c
...
```

   and their output must be stored in the files `./log/prog1.log` and `./log/prog3.log`, respectively.

```bash
#!/bin/bash

#
# Exercise 4 of 2019/02/01
# Launch with= ./20190201_es4.sh input.txt
#
# Check arguments

if [ $# -ne 1 ] ; then
    echo "Usage: ./20190201_es4.sh <filename>"
    exit 1
fi
# Create log directory (if it does not exist yet)
if [ ! -d "logs" ] ; then
    mkdir "logs"
fi

# Read file line by line
NAME=" "
SOURCES=" "
while read line ; do
    # Handle start tines
    echo $line | grep -e "^#.*" > /dev/null
    if [ $? -eq 0 ] ; then
        NAME=$(echo $line | cut -d" " -f2)
        continue
    fi
    # Handle end lines
    echo $line | grep -e "^END$" > /dev/null
    if [ $? -eq 0 ] ; then
        gcc -Wall -o $NAME $SOURCES &> "logs/$NAME.log"
        S0URCES=" "
        NAME=" "
        continue
    fi
    # Handle intermediate tines
    SOURCES=$SOURCES" "$line
done < $1

exit 0
```

4. **Exam 2019/02/01 Exercise 5:**
   In linear algebra, matrices multiplication is an operation that produces a matrix C, by performing the product rows by columns of two given matrices A and B.

   More in details, if matrix A has dimension $[r, x]$ and matrix B has dimension $[x, c]$, then C has dimension $[r, c]$, and each element in position (i, j) can be computed as:

$$C[i][j] = \sum_{k=0}^{x-1} A[i][k] \cdot B[k][j]$$

   Write a multi-thread function

```
void mat_mul (int **A, int **B, int r, int x, int c, int **C);
```

   that can compute the product matrix C executing a thread for each of its elements.
   Each thread has to compute the value of one element, executing the row by column product previously reported.
   Define the data structure needed for the execution of each thread.

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

#define Rs 3
#define Xs 4
#define Cs 2

typedef struct thread_s {
  int **A;
  int **B;
  int r;
  int x;
  int c;
  int **C;
  int i;
  int j;
} thread_t;


static int **mat_alloc (int, int);
static void mat_init (int **, int, int);
static void mat_mul_seq (int **, int **, int, int, int, int **);
static void mat_mul (int **, int **, int, int, int, int **);
static void *prod_thread (void *);
static void mat_print (int **, int, int);


int main (void) {
  int **A, **B, **C;
  A = mat_alloc (Rs, Xs);
  mat_init (A, Rs, Xs);
  B = mat_alloc (Xs, Cs);
  mat_init (B, Xs, Cs);
  C = mat_alloc (Rs, Cs);
  fprintf (stdout, "Sequential product:\n");
  mat_mul_seq (A, B, Rs, Xs, Cs, C);
  mat_print (C, Rs, Cs);
  fprintf (stdout, "Concurrent product:\n");
  mat_mul (A, B, Rs, Xs, Cs, C);
  mat_print (C, Rs, Cs);

  return (0);
}



static int **mat_alloc (int r, int c) {
  int **m, i;
  m = (int **) malloc (r * sizeof (int *));
  for (i=0; i<r; i++) {
    m[i] = (int *) malloc (c * sizeof (int) );
  }
  return m;
}



static void mat_init (int **m, int r, int c) {
  int i, j, n;
  n=1;
  for (i=0; i<r; i++) {
    for (j=0; j<c; j++) {
      m[i][j] = n++;
    }
  }
}
```

```c
static void mat_mul_seq (int **A, int **B, int r, int x, int c, int **C) {
  int i, j, k;
  for (i=0; i<r; i++)
    for (j=0; j<c; j++) {
      C[i][j] = 0;
      for (k=0; k<x; k++)
        C[i][j] += A[i][k] * B[k][j];
    }
}


/************************************
This function is required in the exam
************************************/
static void mat_mul (int **A, int **B, int r, int x, int c, int **C) {
  pthread_t *th;
  thread_t *ts;
  int i, j;

  th = (pthread_t *) malloc(r * c * sizeof(pthread_t) );
  ts = (thread_t *) malloc(r * c * sizeof(thread_t) );

  for (i=0; i<r; i++)
    for (j=0; j<c; j++) {
      ts[i*c+j].A = A;
      ts[i*c+j].B = B;
      ts[i*c+j].C = C;
      ts[i*c+j].r = r;
      ts[i*c+j].x = x;
      ts[i*c+j].c = c;
      ts[i*c+j].i = i;
      ts[i*c+j].j = j;

      pthread_create(&th[i*c+j], NULL, prod_thread, (void *) &ts[i*c+j]);

    }

  for (i=0; i<r; i++) {
    for (j=0; j<c; j++) {
      pthread_join (th[i*c+j], NULL);
    }
  }
}



/************************************
This function is required in the exam
************************************/
static void *prod_thread (void *arg) {
  thread_t *ts = (thread_t *) arg;
  int k;

  ts->C[ts->i][ts->j] = 0;
  for (k=0; k<ts->x; k++)
    ts->C[ts->i][ts->j] += ts->A[ts->i][k] * ts->B[k][ts->j];

  return NULL;
}



static void mat_print (int **m, int r, int c) {
  int i, j;
  for (i=0; i<r; i++) {
    for (j=0; j<c; j++)
      fprintf (stdout, "%d ", m[i][j]);
    fprintf (stdout, "\n");
```

```
  }
}
```

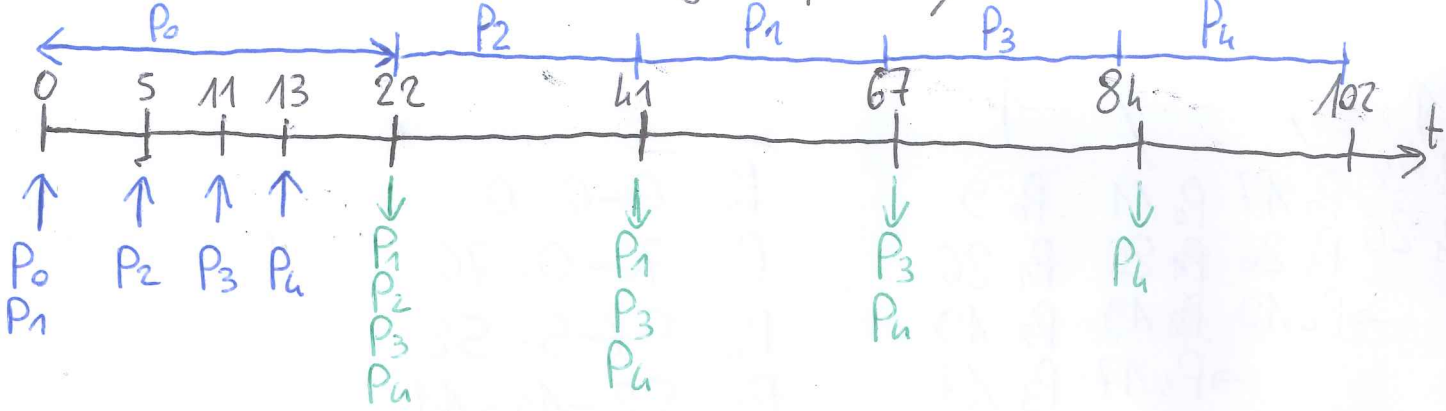
5. **Exam 2019/02/01 Exercise 6:**
Consider the following set of processes:

Process	Arrival Time	Burst Time	Priority
P_0	0	22	2
P_1	0	26	3
P_2	5	19	1
P_3	11	17	4
P_4	13	18	5

Represent using a Gantt diagram the execution of these processes using the scheduling algorithms PS (Priority Scheduling), RR (Round Robin), and SRTF (Shortest Remaining Time First).
Compute the average *waiting time* for each process and for the global set of processes. Consider a temporal *quantum* of 10 units.
Illustrate which other evaluation metrics can be used in order to compare the previously indicated scheduling algorithms.

EXAM 2019/02/01 (EX. 6)

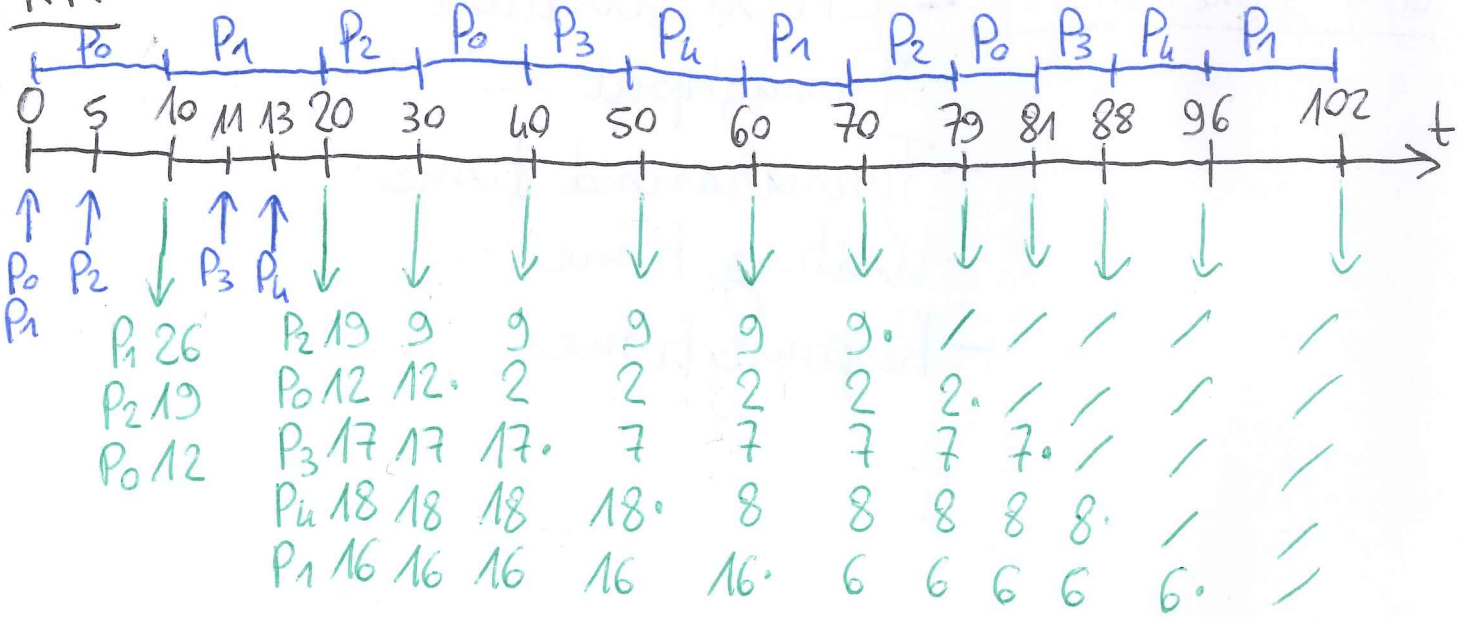
PS (Small number ⇒ higher priority)



P0 0-0=0
P1 41-0=41
P2 22-5=17
P3 67-11=56
P4 84-13=71

$$\overline{T_a} = \frac{0+41+17+56+71}{5} = 37,0$$

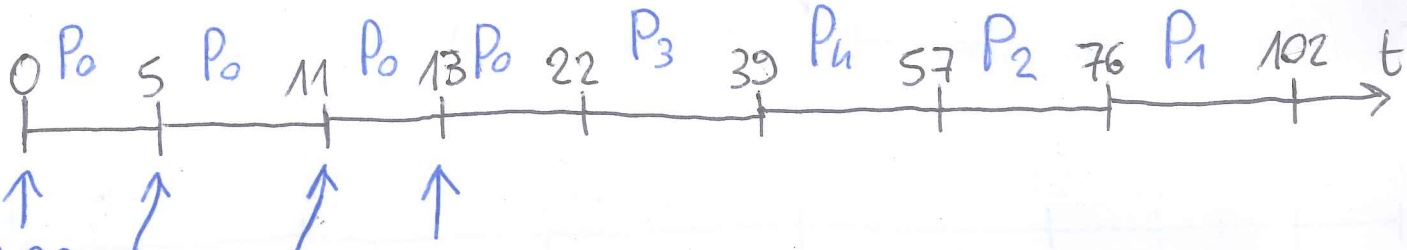
RR



P0 (0-0)+(30-10)+(79-40) = 20+39=59
P1 (10-0)+(60-20)+(96-70)=10+40+26=76
P2 (20-5)+(70-30)=15+40=55
P3 (40-11)+(81-50)=29+31=60
P4 (50-13)+(88-60)=37+28=65

$$\overline{T_a} = \frac{59+76+55+60+65}{5} = 63,0$$

SRTF



P0 22	P0 17	P0 11	P0 9
P1 26	P1 26	P1 26	P1 26
→P2 19	P2 19	P2 19	P2 19
	→P3 17	P3 17	P3 17
		→P4 18	

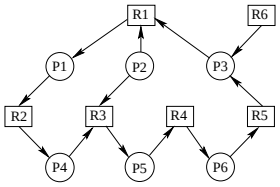
P0 $0-0=0$
P1 $76-0=76$
P2 $57-5=52$
P3 $22-11=11$
P4 $39-13=26$

$$\overline{T_a} = \frac{0+76+52+11+26}{5} = 33,0$$

Cost functions:
- CPU utilisation
- Throughput
- Turnaround time
- Waiting time
- Response time

6. **Exam 2019/02/27 Exercise 6:**
Illustrate the terms "*safe state*", "*safe sequence*", and (with an example) "*Resource trajectory plot*".

For the management of the deadlock, two operating systems use the graph reported on the left, and the table reported on the right of the following figure.



Process	Finish	Allocation			Max			Need			Available		
		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	1	0	1	4	3	4				3	2	2
P_2	F	0	0	1	4	4	2						
P_3	F	0	1	1	2	3	3						
P_4	F	0	0	0	5	4	5						
P_5	F	0	1	0	1	4	4						

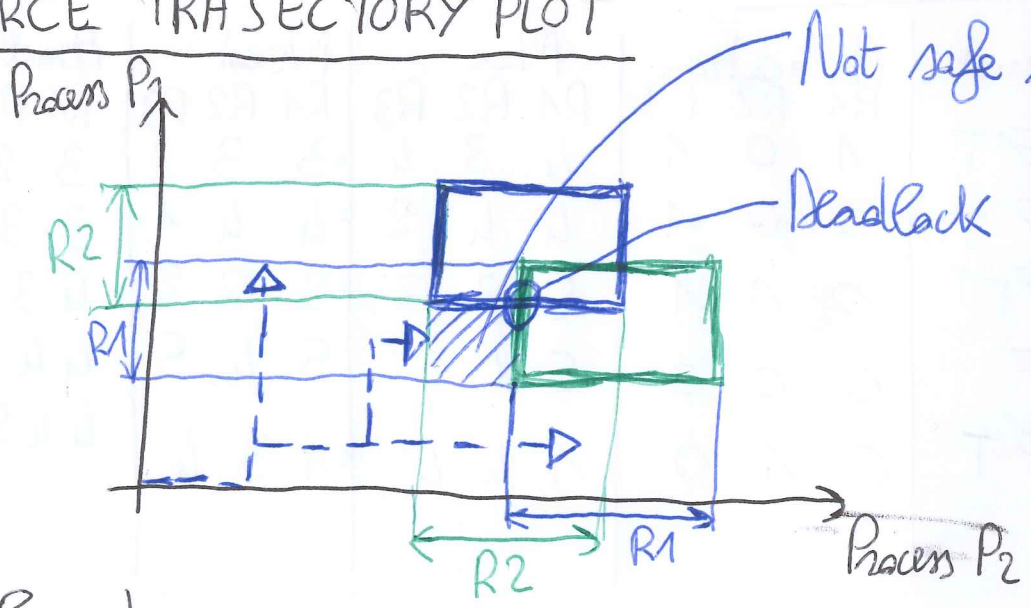
Considering the two cases separately, indicate whether the states represented are safe, and in the case reports a possible safe sequence.

EXAM 2019/02/27 (EX. 6)

SAFE STATE: The system is in a state in which it is able to allocate the resources to all the active processes, preventing the verification of a deadlock. State for which a secure sequence exists.

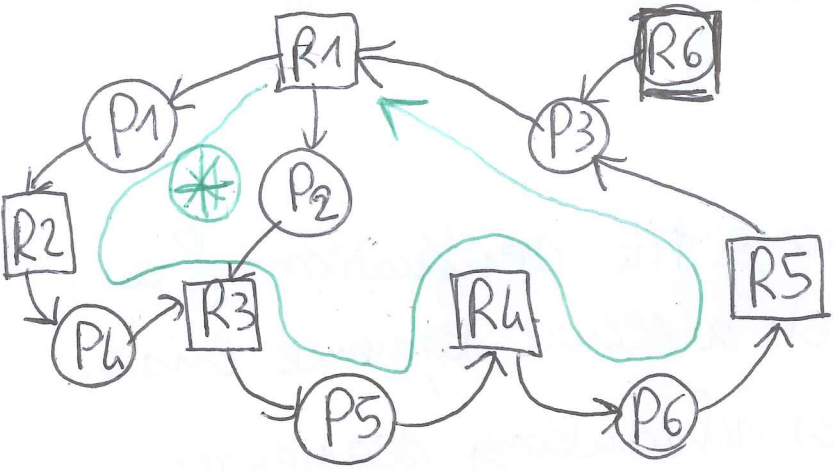
SAFE SEQUENCE: Processes scheduling sequence such that the requests of each process can be satisfied by using the resources available in the current state and those made free from the processes already terminated.

RESOURCE TRAJECTORY PLOT



Explanation...

Resource RESOURCE ALLOCATION GRAPH



Single instance for each resource.

(*) ≡ CYCLE ⟺ DEADLOCK (and process can end).

The state is not safe.

BANKER'S ALGORITHM

Process	Finish	Allocation			Max			Need			Avail		
		R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
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

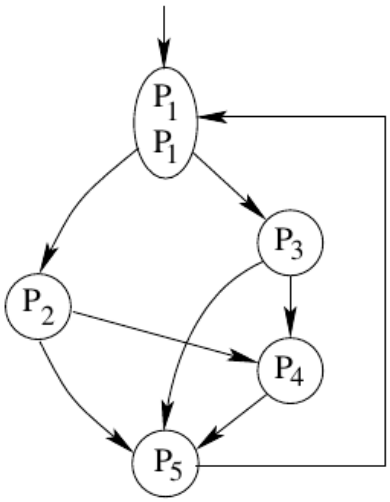
P_3 → P_1 → P_5 → P_2 → (**)

The only possible sequence (**) $P_3 \to P_1 \to P_5 \to P_2$

it is NOT complete (P_4 is not included).

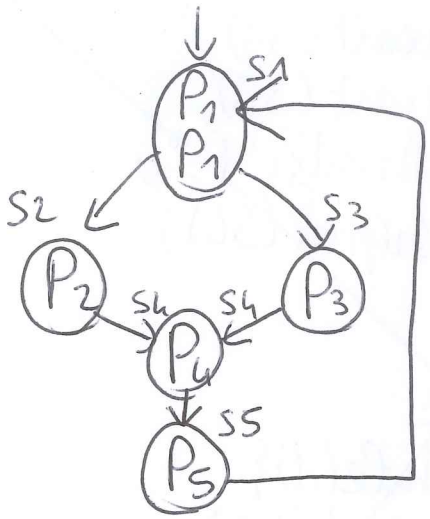
The state is NOT safe

7. **Exam 2016/02/28 Exercise 2:**
   Given the following precedence graph, implement it by using the minimum number of semaphores. Processes must be **cyclic** (i.e., function with body **while(1)**).
   Use the primitives `init`, `signal`, `wait` and `destroy`, indicate and eliminate any unnecessary arc, report the main part of processes $(P_1, ..., P_5)$, and the initialization of semaphores. Note that process P_1 must perform two iterations in sequence on its while loop for each global iteration over the entire precedence graph.



EXAM 2016/02/28 (EX. 2)

Arcs P_3-P_5 and P_2-P_5 are useless.
New precedence graph is



Solution A

```
init(S1,1); init(S2,Ø); init(S3,Ø); init(S4,Ø); init(S5,Ø);
init n=0;
```

P1

```
while(1){
  wait(S1);
  printf("P1");
  if(n==0){
     signal(S1); }
     n=1;
  }else{
     n=Ø;
     signal(S2);
     signal(S3);
  }
}
```

P2

```
while(1){
  wait(S2);
  printf("P2");
  signal(S4);
}
```

P3

```
while(1){
  wait(S3);
  printf("P3");
  signal(S4);
}
```

P4

```
while(1){
  wait(S4);
  wait(S4);
  printf("P4");
  signal(S5);
}
```

P5

```
while(1){
  wait(S5);
  printf("P5");
  signal(S1);
}
```

```
destroy(S1....S5);
```

Solution B

```
init(S1,2); init(S2,0); init(S3,0); init(S4,0); init(S5,0);
```

P1
```
while(1){
  wait(S1);
  printf("P1");
  signal(S2);
  signal(S3);
}
```

P2
```
while(1){
  wait(S2);
  wait(S2);
  printf("P2");
  signal(S4);
}
```

P3
```
while(1){
  wait(S3);
  wait(S3);
  printf("P3");
  signal(S4);
}
```

P4
```
while(1){
  wait(S4);
  wait(S4);
  printf("P4");
  signal(S5);
}
```

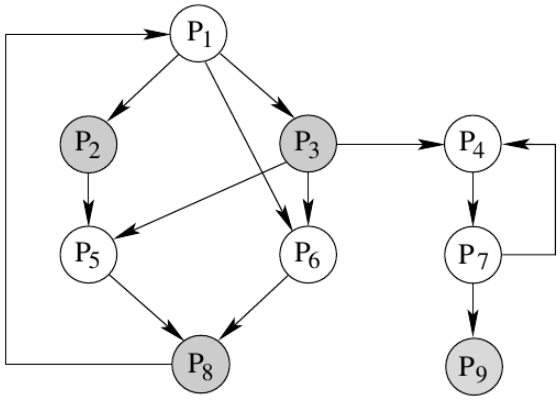
P5
```
while(1){
  wait(S5);
  printf("P5");
  signal(S1);
  signal(S1);
}
```

```
destroy(S1); .... destroy(S5);
```

The following solution cannot be accepted

P1
```
while(1){
  wait(S1);
  printf("P1");
  printf("P1");
  signal(S2);
  signal(S3);
}
```

8. **Exam 2014/09/08 Exercise 3:**
Given the following precedence graph, implement it by using the minimum number of semaphores. White processes are **acyclic**, while gray processes are **cyclic** (i.e., function with body **while(1)**).
Use the primitives `init`, `signal`, `wait` and `destroy`. Report semaphores initialization and destruction, and the main part of processes $(P_1, ..., P_9)$.



EXAM 2014/09/08 (EX. 3)

Arc $P_1 - P_6$ is useless

```
init(S2,0); init(S3,0); init(S4,1); init(S5,0);
init(S8,0); init(S9,0);
```

P1
```
printf("P1");
signal(S2);
signal(S3);
exit(0);
```

P2
```
while(1){
  wait(S2);
  printf("P2");
  if(fork()==0){
    P5();
  }
}
```

P3
```
while(1){
  wait(S3);
  printf("P3");
  signal(S5);
  if(fork()==0){
    P4();
  }
  if(fork()==0){
    P6();
  }
}
```

P4
```
wait(S4);
printf("P4");
if(fork()==0){
  P7();
}
exit(0);
```

P5
```
wait(S5);
printf(P5);
signal(S8);
exit(0);
```

P6
```
printf("P6");
signal(S8);
exit(0);
```

P7
```
printf("P7");
signal(S4);
signal(S9);
exit(0);
```

P8
```
while(1){
  wait(S8);
  wait(S8);
  printf("P8");
  if(fork()==0){
    P1();
  }
}
```

P9
```
while(1){
  wait(S9);
  printf("P9");
}
```

```
destroy(S2);
...
...
destroy(S9);
```

9. **Exam 2019/02/27 Exercise 1:**
Suppose that compiling the following program

```
#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);
}
```

an executable with name `/home/quer/20190227` has been obtained.

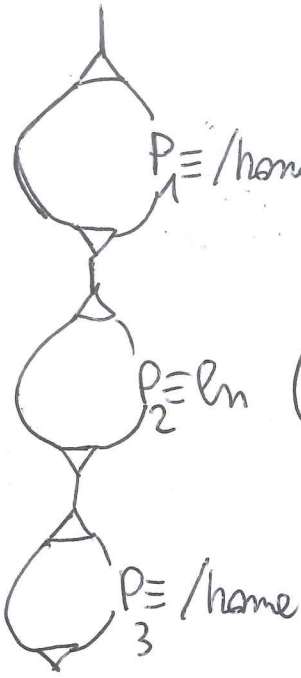
Then, suppose to run the following script:

```
/home/quer/20190227
ln /home/quer/20190227 p
/home/quer/p
```

Report the *control flow graph (CFG)* and the *process generation tree* produced by its execution. In addition, indicate what it produces on the screen and for what reason.

EXAM 2019/02/27 (EX. 1)

CFG: LINUX SHELL

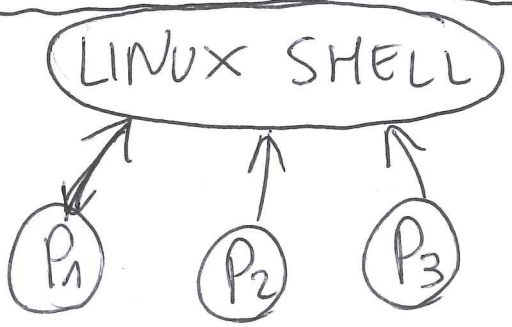


$P_1 \equiv$ /home/quer/20190227 ⇒ PRINT "exec 2 2↵"
execlp FAILS (link p does not exist)
PRINT "i = 2↵"
return

$P_2 \equiv$ ln (Do not execute code)

$P_3 \equiv$ /home/quer/p ⟹ PRINT "exec 1 3↵"
execlp } Substitute the image of p with echo
ECHO "/home/quer/p i↵"
ends

PROCESS GENERATION TREE: ~~Not present because~~



OUTPUT:

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