Reserved cells

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

Operationg Systems

Examination task

27 January 2020

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.**

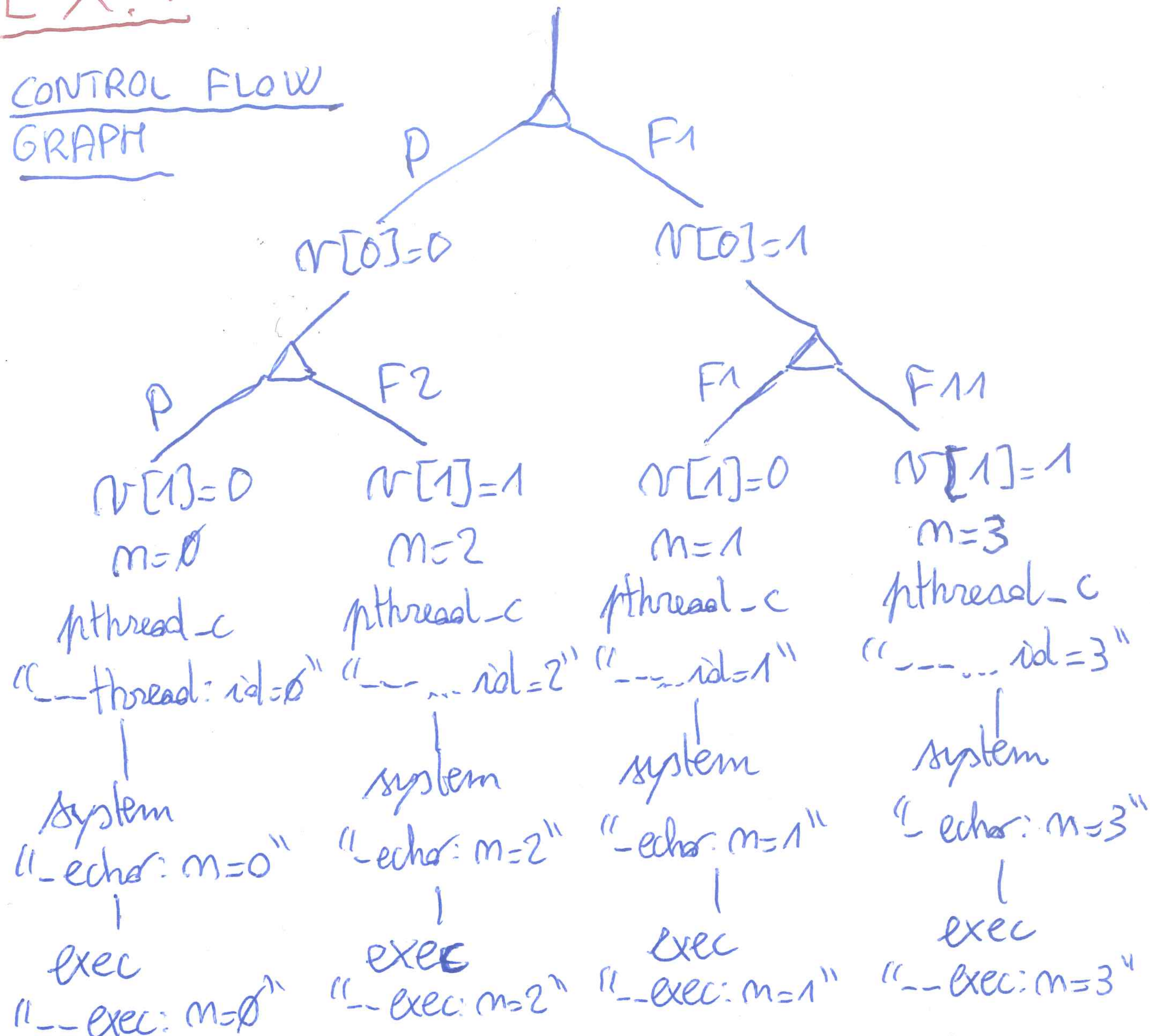
1. Suppose to execute the following program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *t1 (void *p){
  int *n = (int *) p;
  printf ("--- thread: id=%d\n", *n);
  pthread_exit (NULL);
}

int main () {
  pthread_t thread;
  int i, n, v[2];
  char str[100];

  setbuf (stdout, 0);
  for (i=0; i<2; i++)
    if (fork()>0) {
      v[i] = 0;
    } else {
      v[i] = 1;
    }

  n = v[0] + v[1]*2;
  pthread_create (&thread, NULL, t1, &n);
  pthread_join (thread, NULL);

  sprintf (str, "echo '- echo: n=%d'", n);
  system (str);
  sprintf (str, "-- exec: n=%d", n);
  execlp ("echo", "bash", str, NULL);

  return 1;
}
```

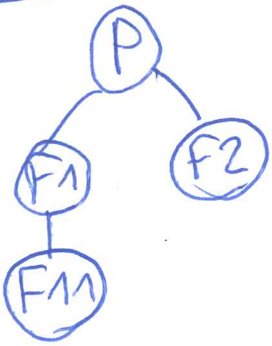
Report the *control flow graph* (CFG) and the *process generation tree* after its execution. Indicate what it produces on video and for what reason.

EX. 1

CONTROL FLOW GRAPH



PROCESS GENERATION TREE



(Each process executes a system, then each process executes an exec)

POSSIBLE OUTPUT

___ thread: id=0
___ thread: id=2
___ thread: id=1
___ thread: id=3
_ echo: m=0
_ echo: m=1
_ echo: m=2
_ echo: m=3
__ exec: m=0
__ exec: m=1
__ exec: m=2
__ exec: m=3

(Real output depends on the scheduler)

2. Describe the syntax of the system calls `wait` and `exit`, and an example on how to transfer with `exit` and `wait` an integer value from a terminated child to the parent. What happens if a parent does not call `wait` and a child terminates? Which mechanism is used by the *kernel* to identify the termination of a process?

   Two child processes, P_1 (with PID 123) and P_2 (with PID 456), which are the only children of a parent, terminates. Indicate for the following two codes the output provided by the function `printf` in the case P_1 terminates before P_2, and P_2 terminates before P_1 (4 cases in total).

   Code 1:
   ```
   while(wait()!=123);
   pid = wait();
   printf("%d\n", pid);
   ```

   Code 2:
   ```
   waitpid(123, (int *) 0, 0);
   pid = wait();
   printf("%d\n", pid);
   ```

Syntax: pid_t wait (int *wstatus); → Waits for a child process
void exit(int status); → Causes normal process termination.

Example:
```
int r_status;
if (fork()){
  wait(&r_status);
  if(WIFEXITED(r_status))
    printf("%d", WEXITSTATUS(r_status));
}else{
  exit(1);
}
```

When the parent does not call wait and one of its child terminates, the process becomes ZOMBIE.
The Kernel uses ~~the~~ signals to identify the termination of a process. In particular, it uses the signal SIGCHLD addressed to the parent.

	code 1	code 2
P_1 before P_2	456	456
P_2 before P_1	-1	456

wait returns -1 for processes without children

Process P_2 becomes ZOMBIE, and then it is ~~[illegible]~~ caught by the wait

3. Illustrate the "*Producer and Consumer*" problem (with P producers and C consumers), and report with pseudo-code a possible implementation schema. Indicate and motivate the function of all the semaphores.

   Then, adapt the previous solution to the case of exactly 3 producers and only one consumer. Each producer must generate elements in a queue dedicated to each process, the consumer must consume elements ensuring a higher priority to the queue with the major number of stored elements.

   Suggestion: use counters to track the number of elements in each queue, or alternatively use a function (e.g., `sem getvalue`) that can return the value of a semaphore.

EX. 3

PRODUCERS / CONSUMERS

init(full,0); init(empty, SIZE); init(meC,1); init(meP,1);

Prod.

```
while(1){
    produce(&val);
    wait(empty);
    wait(meP);
    enqueue(val);
    signal(meP);
    signal(full);
}
```

Cons

```
while(1){
    wait(full);
    wait(meC);
    dequeue(&val);
    signal(meC);
    signal(empty);
    consume(val);
}
```

WITH sem_getvalue()

init(full[i],0); init(empty[i], SIZE); $i = 0, 1, 2$

P_0, P_1, P_2 (Prod.)

```
while(1){
    produce(&val);
    wait(empty[i]);
    enqueue(queue[i], val);
    signal(full[i]);
}
```

Cons.

```
while(1){
    n0=sem_getvalue(full[0]);
    n1=sem_getvalue(full[1]);
    n2=sem_getvalue(full[2]);
    n=max_idx(n0,n1,n2);
    wait(full[n]);
    dequeue(queue[n],&val);
    signal(empty[n]);
    consume(val);
}
```

WITH COUNTERS

```
init(full,0); init(empty[i],SIZE); init(m[i],1); i=0,1,2
int counter[i]=0;
```

P0,P1,P2 (Prod) (i=process idx)

```
while(1){
    produce(&val);
    wait(empty[i]);
    wait(m[i]);
    enqueue(queue[i],val);
    counter[i]++;
    signal(m[i]);
    signal(full);

}
```

Cons.

```
while(1){
    wait(full);
    wait(m[0]);
    wait(m[1]);
    wait(m[2]);
    n=max_idx(counter[0],
        counter[1],counter[2]);
    for(i=0; i<3; i++)
        if(i!=n) signal(m[i]);
    dequeue(queue[n],&val);
    counter[n]--;
    signal(m[n]);
}
```

4. Implement a BASH script that receives the path of a directory from command line. The script, after checking the passage of the correct number of parameters, it must select, in the sub-tree of directors with the specified directory as root, all the regular files with dimension less than 10MB, whose name starts with the string "expense" followed by an unsigned integer number and with extension `.xyz` (e.g., `expense1.xyz`, `expense200.xyz`).

   Assume that each of these files contains a text with a format similar to the following:

   expense1.xyz
   ```
   Product Quantity Unit_price
   pasta 2 5
   pizza 1 8
   pasta 1 6
   ```

   expense200.xyz
   ```
   Product Quantity Unit_price
   pizza 2 2
   fruit 3 5
   ```

   where the first line is a header, while the following contain the name of a product, its quantity, and its unit price (separated by single spaces).

   For each selected file, the script must generate a file with the same name but with extension `.dat`, without header, that contains for each product the total, obtained by summing the quantities multiplied by the unit price of all the lines in which that product appears.

   For the example files shown above, the generated files must be the following:

   expense1.dat
   ```
   pasta 16
   pizza 8
   ```

   expense200.dat
   ```
   pizza 4
   fruit 15
   ```

```bash
#!/bin/bash

################################################################################
# Exercise 4 - Exam 2020/01/27                                                 #
#   Run with: ./es4.sh <folder>                                                #
################################################################################

# Control of arguments
if [ $# -ne 1 ]; then
    echo "Usage: es4.sh <folder>"
    exit 1
fi

# Select the files, and save the list of paths into a temporary file
find $1 -type f -size -10M -regex '.*\/spesa[0-9]+\.xyz$' > /tmp/$$

# Scan the paths of selected files
while read filename; do

    # Declaration of an associative array to store the expenses
    declare -A sums

    # Scan the content of the file, skipping the first line
    isfirst=0
    while read product quantity price; do
        if [ $isfirst -ne 0 ]; then
            let sums[$product]+=quantity*price
        fi
        isfirst=1
    done < $filename

    # Generate the output file
    name=$(basename $filename ".xyz")
    outfile="$name.dat"

    # Print the expenses in the output file
    for product in "${!sums[@]}"; do
        echo $product ${sums[$product]}
    done > $outfile

    # Delete the associative vector
    unset sums

done < /tmp/$$

# Delete temporary file
rm -f /tmp/$$

exit 0
```

```
#!/bin/bash

##############################################################################
# Exercise 4 - Exam 2020/01/27                                               #
#   Run with: ./es4.sh <folder>                                              #
##############################################################################

# Control of arguments
if [ $# -ne 1 ]; then
    echo "Usage: es4.sh <folder>"
    exit 1
fi

# Select the files, and save the list of paths into a temporary file
find $1 -type f -size -10M -regex '.*\/spesa[0-9]+\.xyz$' > /tmp/$$

# Scan the paths of selected files
while read filename; do

    # Remove the header, and order lines by product in the second temporary file
    cat $filename | tail -n +2 | tr -s " " | sort -t " " -k 1 > "/tmp/$$_2"

    # Generate the output file
    name=$(basename $filename ".xyz")
    outfile="$name.dat"

    # Scan the content of the file, adding and printing the expenses for the
product
    current=""
    tot=0
    while read product quantity price; do
        if [ "$current" == "" ]; then
            current=$product
            tot=0
        elif [ "$product" != "$current" ]; then
            echo $current $tot >> $outfile
            current=$product
            tot=0
        fi
        let tot+=quantity*price
    done < "/tmp/$$_2"
    echo $current $tot >> $outfile

    # Delete second temporary file
    rm -f "/tmp/$$_2"

done < /tmp/$$

# Delete temporary file
rm -f /tmp/$$

exit 0
```

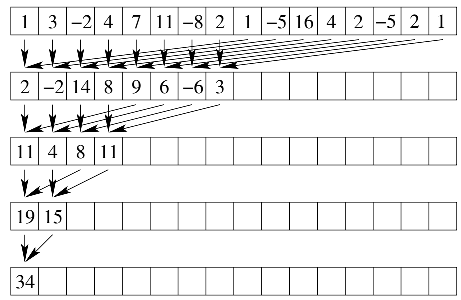
```
#!/bin/bash

################################################################################
# Exercise 4 - Exam 2020/01/27                                                 #
#   Run with: ./es4.sh <folder>                                                #
################################################################################

# Control of arguments
if [ $# -ne 1 ]; then
    echo "Usage: es4.sh <folder>"
    exit 1
fi

# Select the files, and save the list of paths into a temporary file
find $1 -type f -size -10M -regex '.*\/spesa[0-9]+\.xyz$' > /tmp/$$

# Scan the paths of selected files
while read filename; do

    # Read the product file
    for product in $(cat $filename | tail -n +2 | cut -d " " -f 1 | sort | uniq);
do

        # Extract the entries in the file for the current product in a second
temporary file
        cat $filename | tail -n +2 | grep $product | cut -d " " -f 2,3 > "/tmp/$
$_2"
        expense=0

        # Sum the expenses for the current product
        while read quantity price; do
            let expense+=quantity*price
        done < "tmp.txt"

        # Generate the output file
        name=$(basename $filename ".xyz")
        outfile="$name.dat"

        # Print on the output file
        echo "$product $expense" >> $outfile

        # Delete second temporary file
        rm "/tmp/$$_2"
    done

done < /tmp/$$

# Delete the temporary file
rm -f /tmp/$$

exit 0
```

5. A function receives as parameters a vector of integers (`vet`) and its dimension (`n`), which is supposed to be equal to a power of 2:

```
int array_sum (int *vet, int n);
```

The function must return the sum of the elements of the vector. The sum has to be computed using a concurrent version of the following algorithm, which is illustrated in the figure for a vector with dimension `n = 16`:

```
int i, k;
k = n/2;
while (k != 0) {
  for (i=0; i<k; i++) {
    vet[i] += vet[i+k];
  }
  k=k/2;
}
```



In particular, the function must apply the steps of the previous algorithm, ensuring that all sum operations are executed (in parallel) by $n/2$ separate threads. Each thread is associated with one of the first $n/2$ cells of the vector. Each thread takes care of executing all the sums whose result must be stored in the cell of the vector associated with it. Note that the number of sums each thread will have to execute depends on the position of the cells of the vector associated with it. Manage synchronization between threads with semaphores, so that all sums are made respecting precedences.

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

typedef struct {
    int *vet;
    sem_t *sem;
    int n;
    int id;
} args_t;

void * adder(void * arg) {

    // Get argurments
    sem_t *sem = ((args_t *) arg)->sem;
    int *vet = ((args_t *) arg)->vet;
    int id = ((args_t *) arg)->id;
    int n = ((args_t *) arg)->n;

    // Perform addition synchronizing with the other threads
    int k=n/2;
    while(k != 0) {
        if(k < n/2)
            sem_wait(&sem[id + k]);
        vet[id] += vet[id + k];
        k=k/2;
        if(id >= k) {
            sem_post(&sem[id]);
            break;
        }
    }

    // Terminate thread
    pthread_exit(0);
}

int array_sum(int *vet, int n) {
    int k=n/2;
    pthread_t *tids;
    args_t *args;
    sem_t *sem;

    // Allocate thread id array
    tids = (pthread_t *) malloc(k*sizeof(pthread_t));

    // Initialize semaphores
    sem = (sem_t *) malloc(k*sizeof(sem_t));
    for(int i=0; i<k; ++i) {
        sem_init(&sem[i], 0, 0);
    }

    // Allocate array of args
    args = (args_t *) malloc(k*sizeof(args_t));
    for(int i=0; i<k; ++i) {
        args[i].id = i;
        args[i].vet = vet;
        args[i].n = n;
        args[i].sem = sem;
    }

    // Start threads
    for(int i=0; i<k; ++i) {
        pthread_create(&tids[i], NULL, adder, &args[i]);
    }

    // Wait for sum to be complete
    pthread_join(tids[0], NULL);
```

```c
    // Destroy semaphores
    for(int i=0; i<k; ++i) {
        sem_destroy(&sem[i]);
    }

    // Free memory
    free(tids);
    free(sem);
    free(args);

    // Return sum
    return vet[0];
}

int main(int argc, char **argv) {
    int res = 0;
    for(int i=0; i<10000; i++) {
        if(i%1000==0) printf("%d\n", i);
        int vet[16] = {1, 3, -2, 4, 7, 11, -8, 2, 1, -5, 16, 4, 2, -5, 2, 1};
        int newres = array_sum(vet, 16);
        if(i == 0) res = newres;
        else if(res != newres) printf("Discrepancy %d %d\n", res, newres);
    }
    printf("Result: %d\n", res);
}
```

6. Clarify the main differences between an ASCII (or text) and a binary file. What advantages and disadvantages do the latter offer?

   Illustrate the main differences between the functions `fopen` and `open`, between `fprintf` and `write`, and between `fscanf` and `read`.

   Explain the differences between the *linked* and *indexed* allocations when saving files, illustrating their advantages and disadvantages. For the indexed allocation in the UNIX/LINUX environment, you have also to indicate what is the meaning of the terms "directory block", "directory entry", "data block" and "i-node".

Text file is "line-oriented" and "byte-oriented", coded in ASCII (and other codings).
Binary file is "bit-oriented".
Advantages binary: typically more compact, allows serialization.
Disadvantages binary: cannot be read by a general-purpose editor, incompatibility with different architectures, coding must be known.

Differences between fopen, open, ...
The former (fopen, fprintf, fscanf) are POSIX and ANSI C library functions, the latter (open, write, read) are system calls. The former are implemented using the latter. The former provide buffered and formatted input/output, the latter are byte oriented, ...
eventually reports the prototype (if known).

Linked allocation: Each block contains a pointer to the next
Advantages: Allows a dynamic allocation of the file, eliminates external fragmentation
Drawbacks: Efficient only for sequential access, wasted space to store pointers, the lose of a block prevent the access to the next blocks

Indexed allocation: A block (i-node) contains pointers to all the data blocks of the file.

Advantages: Direct access to any block of the file.
Drawbacks: Overhead due to i-node access, if the i-node is lost all the file is lost.

Directory block: list of directory entries
Directory entry: Couple filename + i-node number or dirname + i-node number.
Data block: Block in which the content of the file is stored.
i-node: portion of non-volatile memory in which information about the file is stored (permissions, timestamps, ...) and it also contains pointers to data blocks.