Πανεπιστήμιο Πειραιώς
Τμήμα Πληροφορικής

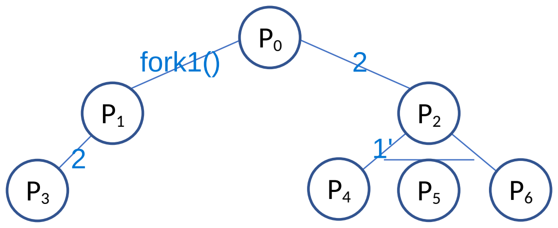
ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2023-24

ΠΡΟΑΙΡΕΤΙΚΗ ΕΡΓΑΣΙΑ

Part 1 (1 point)

Write a C program with the appropriate fork() system calls to create a total of (along with the main program) seven processes, with the following tree structure:



As main part of its work, each process Pi simply prints a message to the screen stating its name (Pi), its PID (Process ID), and its PPID (Parent Process ID). Your program must additionally include appropriate wait statements (wait() and/or waitpid() system calls) to satisfy the following constraints:

1. Process P2, before performing the main part of its work, it should wait for the completion of process P5 and additionaly one more of its two other immediate children (either P4 or P6).
2. Process P0, before executing its main task, it should wait for the completion of process P1.

Finally, process P0 after its execution is replaced (using one of the exec() family of system calls) by the cat command, which will print the source code of your program.

Part 2 (1.5 points)

Consider the following 4 functions, each of which should be executed by a different POSIX thread:

Code of thread with ID = 0

```
void *thread0(void *arg)
{
  /*
   * The identifier (ID)
   * should  be passed as
   * function parameter during
   * thread creation.
   */
  int id = …
  int i;

  printf("ID = %d\n", id);   4

  printf("A\n");    8

  for (i = 0; i < 1000000; i+
+) {
    counter++;
  }

  printf("B\n");     11
  printf("C\n");     13

  for (i = 0; i < 1000000; i+
+) {
    counter++;
  }

  return(NULL);
}
```

Code of thread with ID = 1

```
void *thread1(void *arg)
{
  /*
   * The identifier (ID)
   * should  be passed as
   * function parameter during
   * thread creation.
   */
  int id = …
  int i;

  printf("ID = %d\n", id);   3

  printf("D\n");   6

  for (i = 0; i < 1000000; i+
+) {
    counter++;
  }

  printf("E\n");    12

  return(NULL);
}
```

Code of thread with ID = 2

```
void *thread2(void *arg)
{
  /*
   * The identifier (ID)
   * should  be passed as
   * function parameter during
   * thread creation.
   */
  int id = …
  int i;

  printf("ID = %d\n", id);  2

  printf("F\n");    7

  for (i = 0; i < 1000000; i+
+) {
    counter++;
  }

  printf("G\n");   9

  for (i = 0; i < 1000000; i+
+) {
    counter++;
  }

  printf("H\n");    14

  return(NULL);
}
```

Code of thread with ID = 3

```
void *thread3(void *arg)
{
  /*
   * The identifier (ID)
   * should  be passed as
   * function parameter during
   * thread creation.
   */
  int id = …
  int i;

  printf("ID = %d\n", id);   1

  for (i = 0; i < 1000000; i+
+) {
    counter++;
  }
      sem 1
  printf("I\n");  5
  printf("J\n");  10


  return(NULL);
}
```

Write a C program in which you include and complete the above functions so that they behave as described below. You should also write the main program (main() function), and you should also define the required variables. The program will use the POSIX Threads library for thread management and POSIX semaphores for thread synchronization. The program requirements are as follows:

Define an appropriate number of POSIX semaphores (named s0, s1, s2, ...), which you will initialize appropriately (function sem_init()) in main() of your program). By using semaphores ensure that messages from all threads are always printed in the following order:

#	Message
1	ID = 3
2	ID = 2
3	ID = 1
4	ID = 0
5	I
6	D
7	F
8	A
9	G
10	J
11	B
12	E
13	C
14	H

- Define an integer variable counter, which will be initialized to 0 and will be visible to all threads. Define an appropriate number of mutex variables (named m0, m1, m2, ...), which you will initialize appropriately. By using mutexes ensure that the final value of the counter variable will always be the correct one (6000000). The program should print the counter variable in main() after all threads have finished their execution.
- Pay special attention so that threads can run concurrently for as long as possible. That is, block a thread with a semaphore only to ensure the correct order of printing messages, but allow the threads' "calculations" to run concurrently when possible.

Experiment in your program with only using semaphores or only using mutexes. Observe the results and think about why these results occur. This will help you better understand the difference between semaphores and mutexes, and when each mechanism should be used.

<u>Procedural issues</u>

Delivery of this work is optional. If the assignment is submitted, the final course grade is increased by the grade you received in the assignment (with a maximum final course grade of 10). If no work is submitted, then the final grade of the course is the grade of the written exam.

Deadline: Sunday 4/01/2024