CS69001 Computing Laboratory – I

Assignment No: C1

Date: 03–October–2018

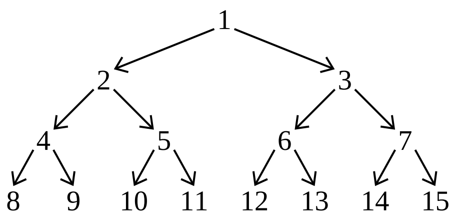
In this assignment, you create process trees simulating two different topologies. Each process in the tree is given a unique user-defined tag. The root process that you run by executing your code (**`a.out`**) is called the *initial process*, and is given the *first* tag. Other processes created by your program obtain their appropriate tags from their respective parents. Each process must know the tag of itself, and the tags of the child processes it creates. Define a suitable structure to store the following information against every process.

- The tag of the process.
- The PID (system-generated) of the process.
- The number of child processes that the process forks.
- An array of the tags of the child processes forked.
- An array of the PIDs of the child processes forked.

Write a function **`dowork()`** that simulates some work by calling the **`sleep()`** or the **`usleep()`** function. This function should sleep for a random duration in the range 1–5 seconds. Write two different programs for the following parts. If a process with tag T creates a process with tag T', we denote this by $T \longrightarrow T'$.

Part 1: Full binary tree

The user supplies the number $d \leqslant 10$ of levels in the tree of processes to be created. The initial process gets tag 1. Other processes get the tags $2, 3, 4, \ldots, 2^d - 1$ in the level-by-level fashion. The following figure illustrates the process tree for $d = 4$. Here, each non-leaf process with tag T forks two child processes with tags $2T$ and $2T + 1$. Leaf processes (those at depth $d - 1$) do not fork any process at all.



Write a function **`createtree(d)`** to create a full binary tree of processes with d levels. For each process, the function returns the information structure (see above) pertinent to that process.

Write a function **`printinfo()`** in order to print the data for a process stored in the structure returned by **`createtree()`**. Use a format shown below. For a process, t $[p]$ stands for its tag t followed by its PID p.

Write a function **`waittree()`** to be called by every process. The function waits until all the child processes terminate. After each child termination, the process prints which child (the child tag) has terminated. When all child processes terminate, the process itself terminates after printing a diagnostic message.

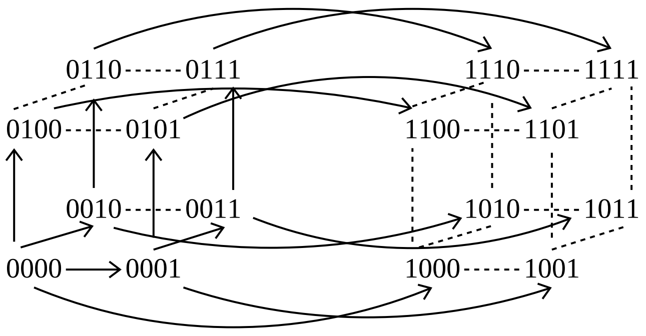
The **`main()`** function should call **`createtree()`**, **`printinfo()`**, **`dowork()`**, and **`waittree()`** in that order, and should be the same for all the processes.

Sample output

```
d = 3
+++ Process  1 [7203]: 2 children:  2 [7204] and  3 [7205]
+++ Process  6 [7206]: 0 children
+++ Process  3 [7205]: 2 children:  6 [7206] and  7 [7207]
+++ Process  7 [7207]: 0 children
+++ Process  4 [7208]: 0 children
+++ Process  5 [7209]: 0 children
+++ Process  2 [7204]: 2 children:  4 [7208] and  5 [7209]
--- Process  6 [7206]: Going to terminate
--- Process  4 [7208]: Going to terminate
--- Process  7 [7207]: Going to terminate
--- Process  5 [7209]: Going to terminate
    Process  3 [7205]: Child  6 [7206] terminated
    Process  3 [7205]: Child  7 [7207] terminated
--- Process  3 [7205]: Going to terminate
    Process  2 [7204]: Child  5 [7209] terminated
    Process  1 [7203]: Child  3 [7205] terminated
    Process  2 [7204]: Child  4 [7208] terminated
--- Process  2 [7204]: Going to terminate
    Process  1 [7203]: Child  2 [7204] terminated
--- Process  1 [7203]: Going to terminate
```

Part 2: Hypercube

Create a d-dimensional hypercube of processes from the user input of $d \leqslant 10$. The 4-dimensional hypercube is shown below. The processes are given d-bit tags. The hypercube is not a tree, and contains additional edges. The tree edges (the parent-child relationship in the process tree) are shown as arrows, whereas the additional edges are shown as dotted lines. In this assignment, do not worry about the dotted lines.



The tags can be identified as d-bit integers (and so are in the range 0 to $2^d - 1$). The initial process has the tag 0 (treated as an integer). It creates a child with tag 1. Then, the processes with tags $0, 1$ fork two child process (one each) with tags $2, 3$. Then, the processes $0, 1, 2, 3$ create four more child processes (again one each) with tags $4, 5, 6, 7$, and so on.

Write functions **`createcube(d)`** and **`waitcube()`** similar to those explained in Part 1. Copy the functions **`printinfo()`** and **`dowork()`** from Part 1. The **`main()`** function (same for each process) should be similar to that in Part 1. Now, different processes spawn different numbers of children (in the range 0 to d). So each process must know exactly how many child processes to wait for.

Sample output

```
d = 3
+++ Process 000 [7915]: 3 children: 001 [7916], 010 [7917], 100 [7918]
+++ Process 100 [7918]: 0 children
+++ Process 010 [7917]: 1 children: 110 [7919]
+++ Process 011 [7920]: 1 children: 111 [7922]
+++ Process 001 [7916]: 2 children: 011 [7920], 101 [7921]
+++ Process 101 [7921]: 0 children
+++ Process 110 [7919]: 0 children
+++ Process 111 [7922]: 0 children
--- Process 100 [7918]: Going to terminate
    Process 000 [7915]: Child 100 [7918] terminated
--- Process 101 [7921]: Going to terminate
--- Process 110 [7919]: Going to terminate
    Process 001 [7916]: Child 101 [7921] terminated
--- Process 111 [7922]: Going to terminate
    Process 010 [7917]: Child 110 [7919] terminated
--- Process 010 [7917]: Going to terminate
    Process 000 [7915]: Child 010 [7917] terminated
    Process 011 [7920]: Child 111 [7922] terminated
--- Process 011 [7920]: Going to terminate
    Process 001 [7916]: Child 011 [7920] terminated
--- Process 001 [7916]: Going to terminate
    Process 000 [7915]: Child 001 [7916] terminated
--- Process 000 [7915]: Going to terminate
```

Submit two C source files. Do not use global/static variables.