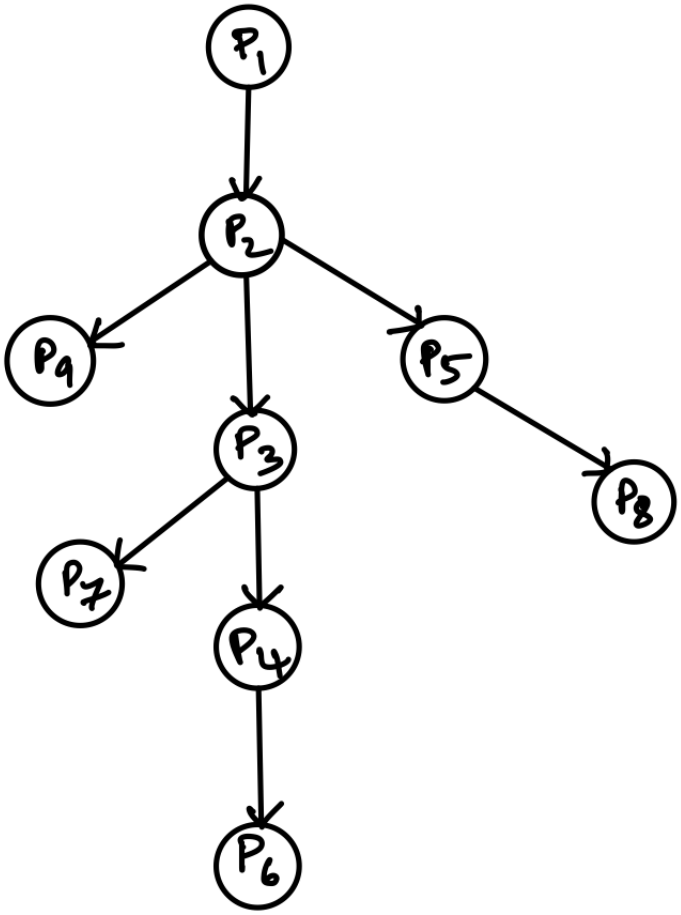
Lab 2 – Fork() and Wait() and Threads in Linux
Operating Systems COMP SCI 3SH3, FALL 2025
Prof. Neerja Mhaskar

1. You must show your working solution of this lab to the TA for a grade.
2. Working directly on your local Linux machine, instead of a virtual machine (VM), may lead to kernel panic if mistakes are made. This could require you to reinstall your operating system and possibly lose data. Therefore, it is advisable to work on your VM for this lab.
3. For Mac M1-4 chip users (all Macs with 64-bit ARM CPUs), you need to install UTM for virtualization: https://mac.getutm.app/
4. The TA will check your solution and will quiz you on your work. After which they will enter your mark and feedback on Avenue.
5. If you do not show your work to your Lab TA, you will get a zero (unless you provide an MSAF, in which case you will get 5 days extension to get this lab graded).
6. It is your responsibility to connect with your Lab TA to get a grade and ensure that your grade has indeed been posted on Avenue.

Outline: PART I

Write a C program that corresponds to the below process tree. You are to use the `fork()` system call to create child processes. Additionally, your program should ensure that the parent processes wait for their child processes to complete. As you can see there are a total of **9** processes (including the parent process).


P1
P2
P9
P3
P5
P8
P7
P4
P6

Output:

1. Your program must print the `pids` of all the processes created on the screen.
2. Your program must ensure that duplicate `pids` are not printed.
3. To compile the code use the below command
   `gcc -o PLfork PLfork.c`
4. To execute the code use the command: `./PLfork`

Sample Output: ./PLfork

7615
7616
7617
7618
7619
7620
7621
7622
7623

PART II

In this section you will learn how to create threads using the Pthreads (POSIX standard) API.

1. The C program shown on the next page demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread.
2. In a Pthreads program, separate threads begin execution in a specified function. In the code on the next page, this is the `runner()` function.
3. When this program begins, a single thread of control begins in `main()`.
4. After some initialization, main() creates a second thread that begins control in the `runner()` function. ***Both threads share the global data sum.***
5. All Pthreads programs must include the `pthread.h` header file.
6. The statement `pthread_t tid` declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The `pthread_attr_t attr` declaration represents the attributes for the thread. We set the attributes in the function call `pthread_attr_init(&attr)`. Because we did not explicitly set any attributes, we use the default attributes provided.
7. A separate thread is created with the `pthread_create()` function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the `runner()` function. Last, we pass the integer parameter that was provided on the command line, `argv[1]`.

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid;  /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  /* set the default attributes of the thread */
 pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

8. At this point, the program has two threads: the initial (or parent) thread in `main()` and the summation (or child) thread performing the summation operation in the `runner()` function. This program follows the thread *create/join* strategy, whereby after creating the summation thread, the parent thread will wait for it to terminate by calling the `pthread_join()` function.
9. The summation thread will terminate when it calls the function `pthread_exit()`.
10. Once the summation thread has returned, the parent thread will output the value of the shared data `sum`.

In your programs you may need to create more than one thread. A simple method for waiting on several threads using the `pthread_join()` function is to enclose the operation within a simple `for` loop.

```
#define NUM THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);
```

Lab question

Given a list of size 20 consisting of natural numbers, write a multithreaded C program for adding all the numbers in the list as follows: The list of numbers is divided into two smaller lists of equal size. Two separate threads (which we will term as ***summing threads***) add numbers in each sublist.

Because global data are shared across all threads, the easiest way to set up the data is to create a global array. Each ***summing thread*** will work on one half of this array. This lab will require passing parameters to each of the **summing threads**. It will be necessary to identify the starting index and ending index of the sublist in which each thread is to begin adding numbers. The parent thread will output the sum once all summing threads have exited.

You are to write a C program using `Pthreads` that contains the entire solution for this question. In particular your program needs to do the following:

1. To be able to create threads in your C program you need to include the `pthread.h` header file.
2. Each thread has a unique thread ID. To create thread IDs for your threads in your program you should use the `pthread_t` data type.
3. Thread attributes should be created/modified using `pthread_attr_t` structure.
4. Declare and code the function in which the thread begins control. For an example see the `runner()` function on the previous page.

5. To be able to identify the starting index and ending index of the sublist in which each thread begins adding numbers you can do the following:
    a. Create a structure to store the starting index and ending index of the sublist. For example:

```
typedef struct { int from_index;
int to_index; } parameters;
```

6. Since threads can share heap, you can simply create a variable of type `parameters`, allocate memory for it on the heap, and assign values to its members as follows:

```
parameters *data =
(parameters *) malloc (sizeof(parameters));
data->from_index = 0; data->to_index = (SIZE/2) - 1;
```

7. To create threads use `pthread_create()` function and pass in the necessary parameters.
8. For the parent thread to output the sum after all summing threads have exited, it is important that you use the `pthread_join()` function.
9. Whenever you dynamically allocate memory on the heap, it is important that you deallocate/free this memory when it is not required by the program using the `free()` function.
10. To compile your program, you need to use the `-pthread` option as follows:

```
gcc -pthread -o PLthreads PLthreads.c
```

Sample Output for the following list:

```
List={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}
./PLthreads
---------------------------------------------------------------------------------------------------
Sum of numbers in the list is: 210
```