CS 1550

Week 5 - Project 2

TA: Jinpeng Zhou jiz150@pitt.edu

CS 1550 – Project 2

- In Project 1, we have a modified Linux with our semaphore implementation, we also have a user-level program: trafficsim.

- In Project 2, you'll need to **write a specific user-level program**, which runs on the modified kernel from Project 1.
  - You can reuse your Project 1 Linux kernel
  - Or you can download a working kernel from Canvas

Build your user-level program on thoth, then run it on qemu virtual machine.

Project 2 - Museum Tour Simulation

- The specific user-level program is a museum tour simulation with some constraints, e.g.:
  - A visitor can tour the museum only if: (1) s/he has a ticket and (2) there's a guide on duty inside the museum. If no ticket is available, visitors will directly leave. If a visitor has a ticket but currently no guide on duty inside the museum, s/he will wait for a guide to be on duty.
  - A tour guide is on duty after arriving, and will be off duty after served 10 visitors or all tickets have been dispensed (i.e. there are no more visitors).
  - A tour guide cannot leave until s/he is off duty.
  - At most **two** tour guides can be in the museum at a time

  Assume the museum knows there will be K guides and M visitors in a given day, i.e., the museum has K*10 tickets available and it knows the number of sold tickets (the number of visitors to be served) is: min(K*10, M)

Please refer to the project pdf for detailed constraints.

Project 2 - Safe Museum Tour Problem

- Your program **should always** satisfy those **constraints,** without causing any **deadlock**

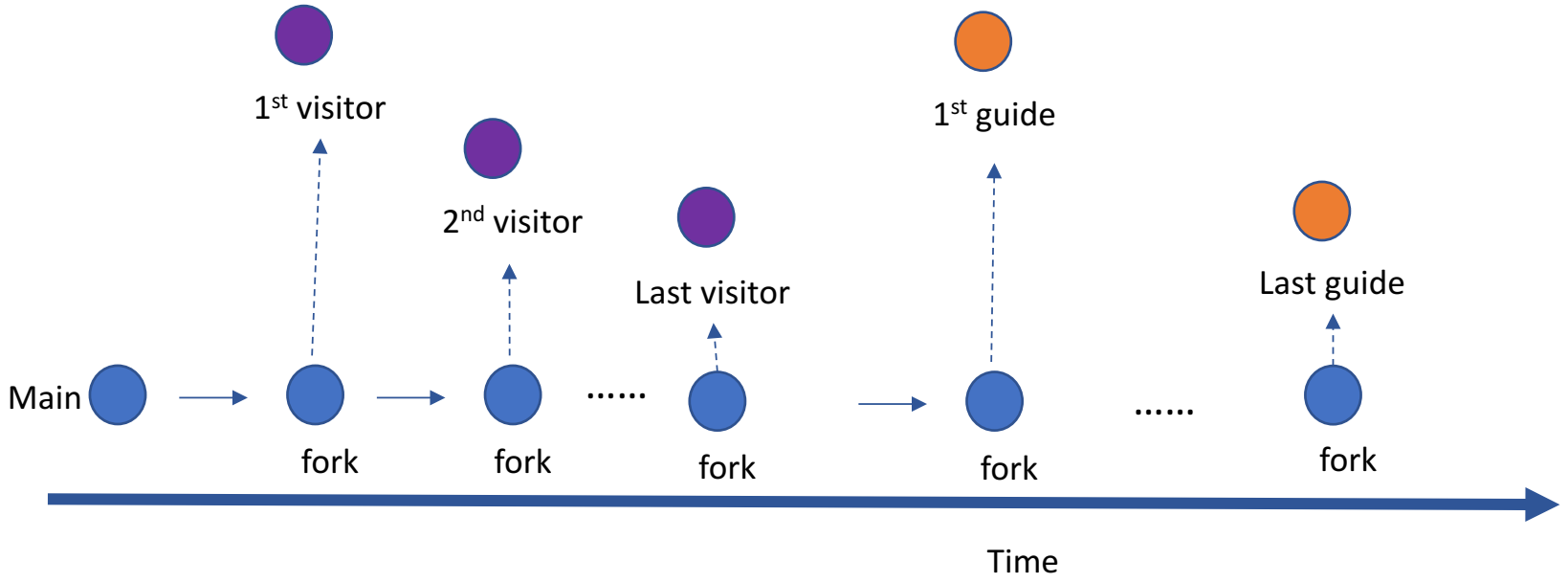
- Your program takes several command-line arguments, such as number of visitors, number of tour guides, etc.

Project 2 – Concurrent Execution

- Visitors and guides have different logics. Each visitor/guide is implemented as a child process.
- How can we create these child processes properly?

Intuitively, we may keep calling fork in the main(), e.g., as shown below. However, with this method, the visitors and guides are not created concurrently, due to a happen-before order: main creates all visitors before creating any guide.

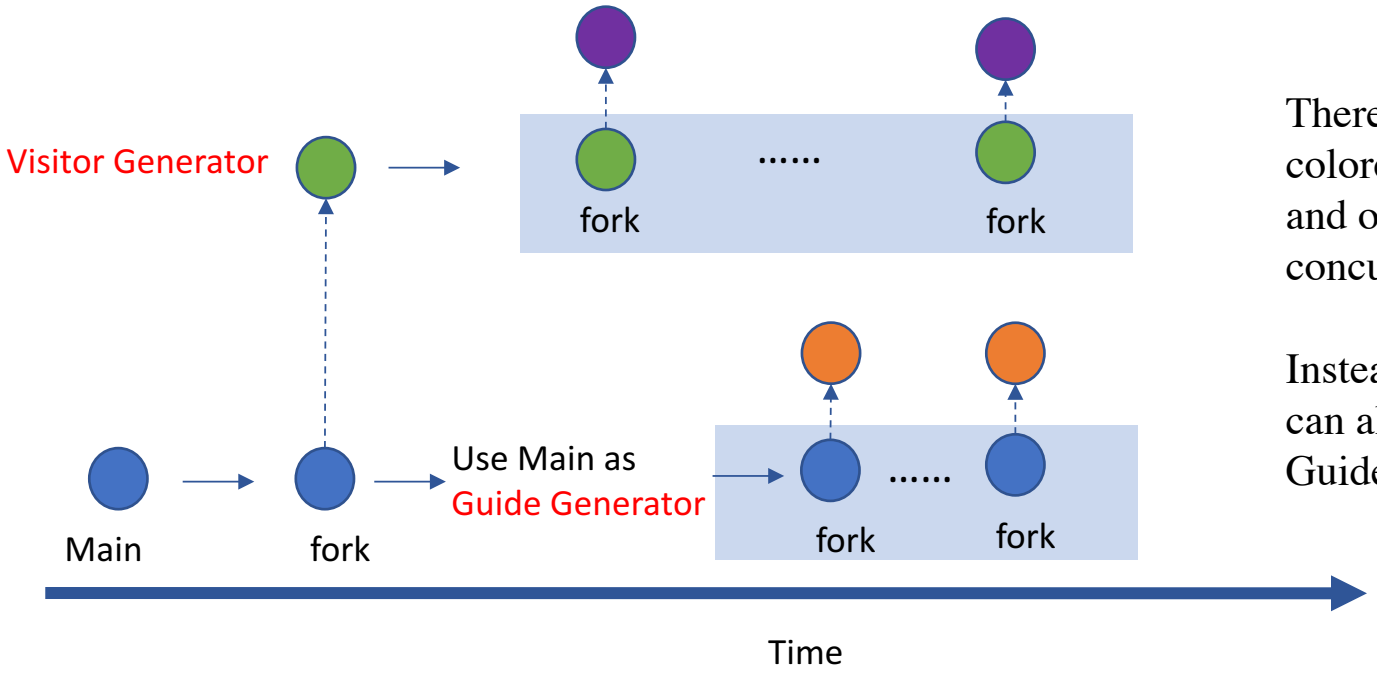
All forks in the Main are sequentially executed



Project 2 – Concurrent Execution

- Visitors and guides have different logics. Each visitor/guide is implemented as a child process.
- How can we create these child processes properly?

In order to concurrently create visitors/guides, we introduce **generator process**, e.g:

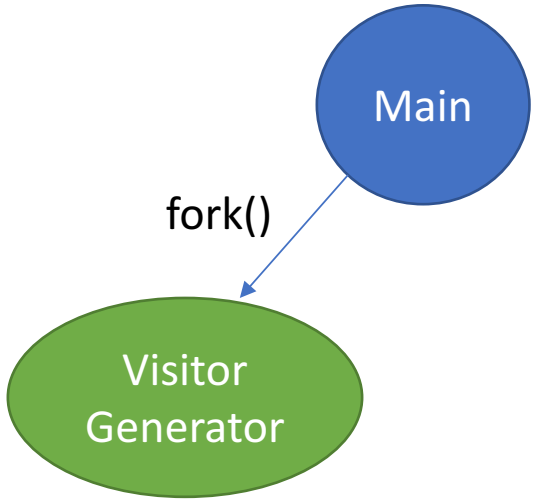


There's no synchronization between the blue-colored rectangles, so the purple nodes(visitors) and orange nodes(guides) can be forked concurrently (without any happen-before order)

Instead of using Main as Guide Generator, you can also explicitly create a new child process as Guide Generator.

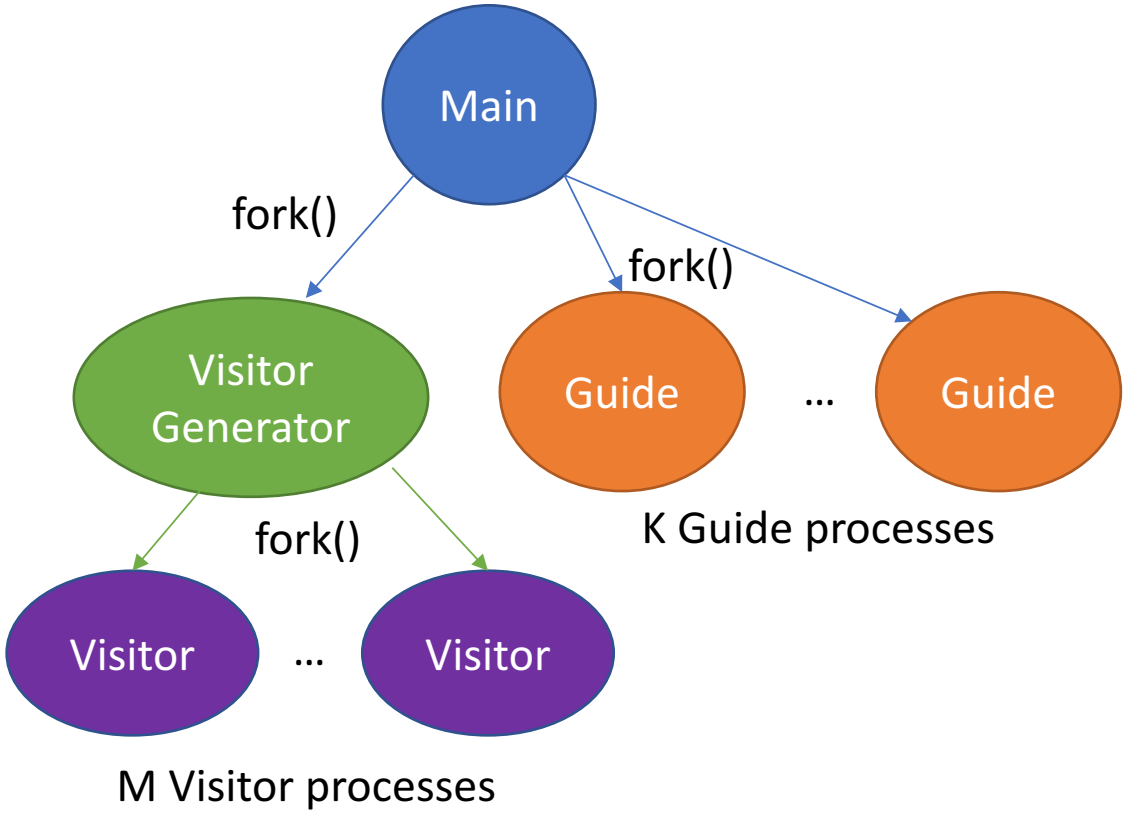
Process hierarchy

In our example, main process creates the visitor generator process.



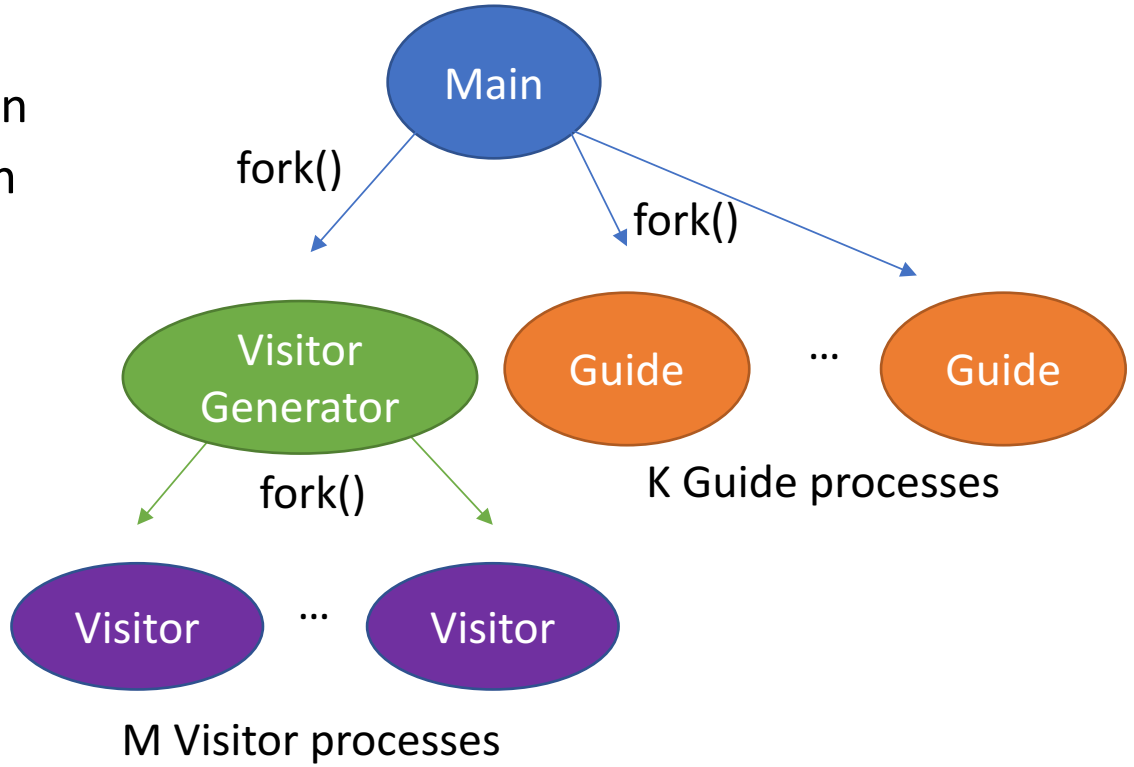
Process hierarchy

In our example, main process creates the visitor generator process.
Then main process works as the guide generator process, and generate all guides.



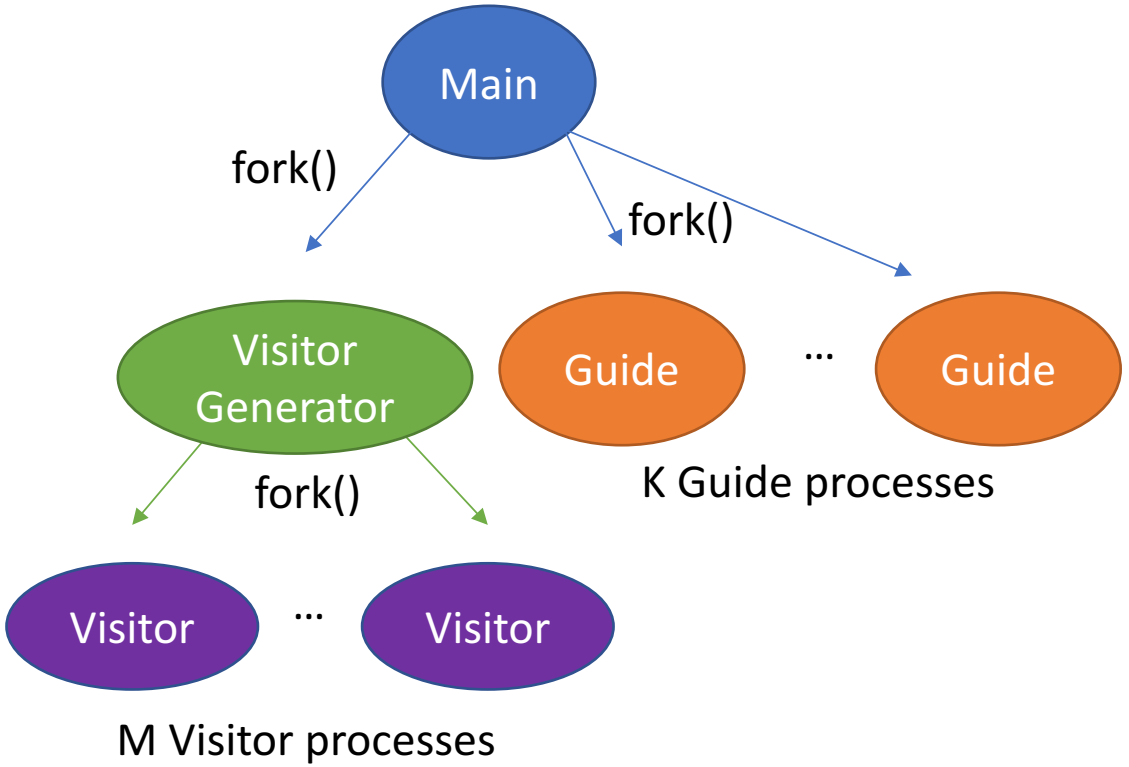
wait() and exit()

- A parent process must call “wait()”, waiting for its child to terminate, such that the resource occupied by the child process can be released. Otherwise:
  - If child terminates before parent, it becomes a “zombie”, still occupying resource after termination
  - The parent may terminate while child is running. In this case, the child becomes an ”orphan”, and will be adopted by the Init process (the ancestor of all processes)



wait() and exit()

- In our example:
  - Main is the parent of visitor generator
    - Main should call one wait() for visitor generator
  - Main is the parent of all guides
    - Main should call K wait() for K guides
  - Visitor generator is the parent of all visitors
    - Visitor generator should call M wait() for M visitors

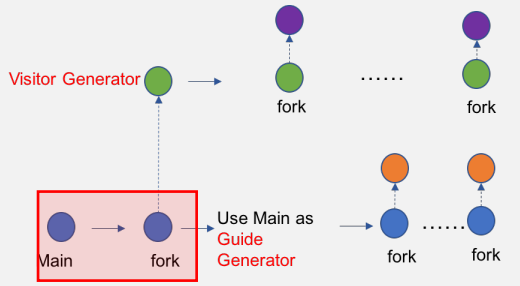


Simulating visitors' & guides' arrival

- Here's one possible implementation:

```
Main process: // herein we ignore cases such as failed fork for simplicity
        /* Main process creates the visitor generator process */
        pid = fork();
        if pid is 0:
                /* Visitor generator process */
        else if pid > 0:
                /* Main process  (reused as Guide generator process) */

                // A straightforward way is to reuse the main process as guide generator,
                // or you can do a fork here to actually create the guide generator
```

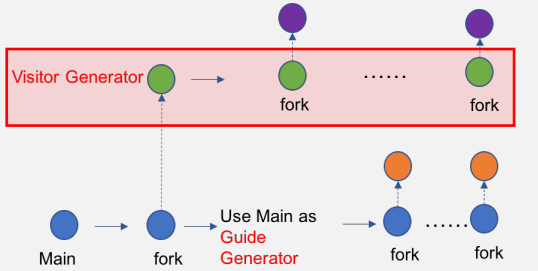


Simulating visitors' & guides' arrival

- Visitor generator process (guide generator process has a similar logic)

```
Visitor generator process: // herein we ignore cases such as failed fork for simplicity
        /* Creates M visitors */
        loop with M iterations:
                pid = fork();
                if pid is 0:
                        /* Visitor process: implement your visitor tour logic here */
                        . . .
                else:
                        /* Generator process: proceed to generate next visitor */

                        // Implement the probabilistic delay before generating next visitor
```



Simulating visitors' & guides' arrival

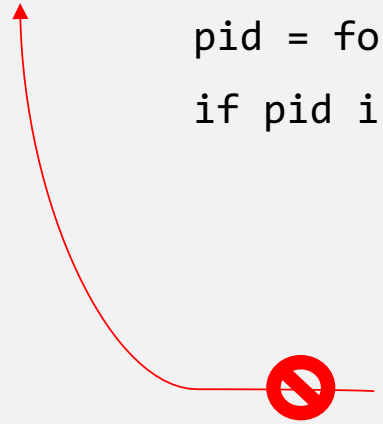
- Probabilistic delay example:

```
srand(seed);     // set seed for the generator
. . .
int value = rand() % 100 + 1;     //random number between 1 and 100
if (value > 30) {        // 70% chance to delay; 30% chance to create next visitor immediately
    sleep(delay_seconds);
}
```

Simulating visitors' & guides' arrival

- Visitor process should exit after the visitor leaves the museum.

```
Visitor generator process: // herein we ignore cases such as failed fork for simplicity
        /* Creates M visitors */
        loop with M iterations:
                pid = fork();
                if pid is 0:
                        /* Visitor process: implement your visitor tour logic here */
                        . . .
                        /* This visitor should not go back to loop after he finishes tour.
                           Otherwise he will be creating visitors */
                        . . .
                else:
                        /* Generator process: proceed to generate next visitor */
```



Share data among processes

- To synchronize visitors and guides, we'll need to check the current status, such as the the current number of visitors/guides inside museum, etc.

- These data should be shared among different processes.
  - Using global variable, local variable, or heap variable won't work, as a child process has its own copies of global/stack/heap regions:

```
pid = fork();
/* child and parent both have a variable named "pid", but in different physical mem addr */
if pid is 0:
        /* child: "pid" equals 0 */
else if pid > 0:
        /* parent: "pid" equals the actual process_id of child */
```

Share data among processes

- Use mmap() to allocate shared memory by using MAP_SHARED flag

#include <sys/mman.h>
void *mmap(void **addr*, size_t *length*, int *protection*, int *flags*, int *fd*, off_t *offset*);

- Assume we'd like to share two 32-bit integers (2*4Bytes = **8 bytes** in total):

```
void *ptr = mmap(NULL, 8, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, 0, 0);  // assume mmap returns 0x100
/* the first 4 bytes (0x100~0x103) is the 1st  integer, the next 4 bytes (0x104~0x107) is the 2nd integer */
int *first =  (int*) ptr;     // set the first int pointer to the start of allocated memory, so the value of first is 0x100
int *second = first + 1;  // adding 1 to an int pointer means adding sizeof(int) bytes, so the value of second is 0x104
```

Alternatively, you can define a struct that contains two ints, and use mmap to allocate mem for a struct object.

More info: http://man7.org/linux/man-pages/man2/mmap.2.html

Print messages

- You'll need to print specific messages, e.g.:
  - When a tour arrives you should print
    - **Tour guide %d arrives at time d%**
  - When a visitor arrives you print
    - **Visitor %d arrives at time %d**

- Use these calls to print messages in multi-process programs:

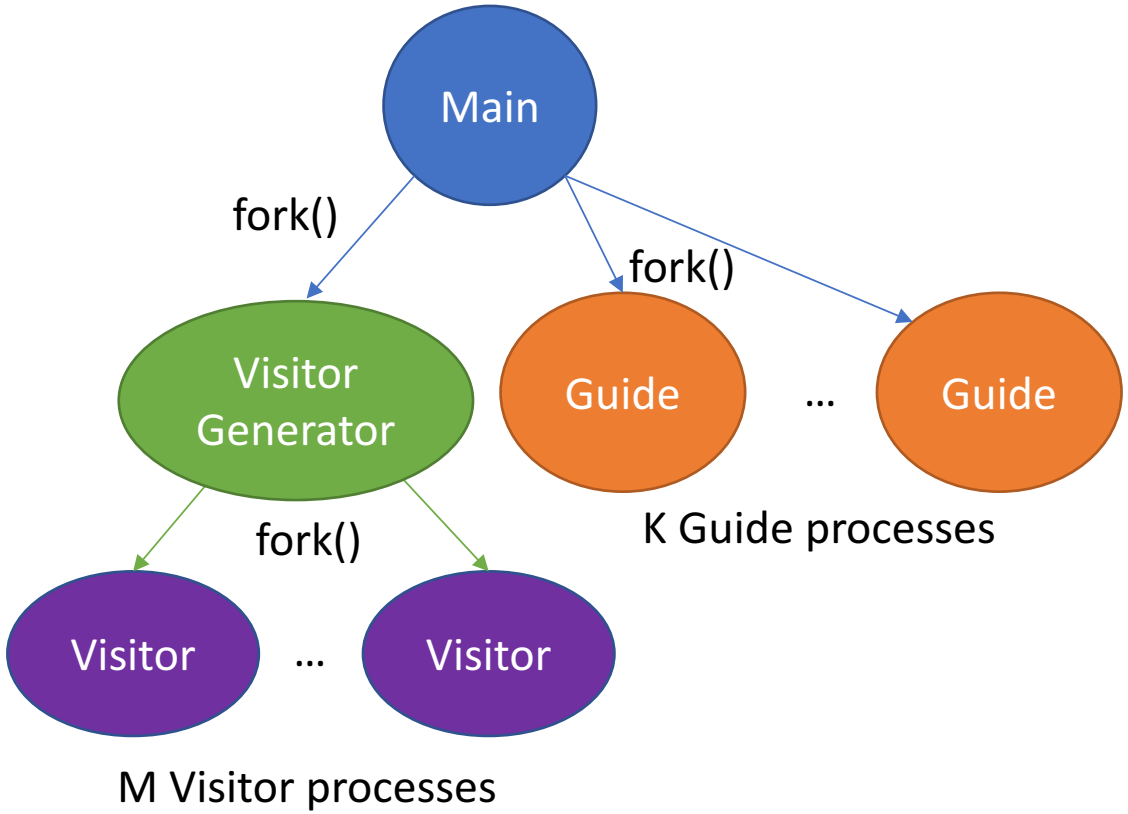
```
printf(<print content here>);
fflush(stdout);
```

OR

```
fprintf(stderr,<print content here>);
```

Tips

- First, try to fork()/wait() processes correctly for M visitors and K guides, without worrying about the detailed sync logic in visitor/guide.



Tips

- Second, use mmap to create some shared variables
  - It should be done before creating processes.

```
void *ptr = mmap(NULL, 8, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, 0, 0);
int *first =  (int*) ptr;
int *second = first + 1;
```

Tips

- Then, think about detailed implementation of visitor and guide. We'll further discuss this next week.

```
Visitor generator process:
        /* Creates M visitors */
        loop with M iterations:
                pid = fork();
                if pid is 0:
                        /* Visitor process: implement your visitor tour logic here */
                        . . . // synchronize, print, exit
                else:
                        /* Generator process: proceed to generate next visitor */
```