Processes, threads and synchronizations

Master in computer science of IP Paris

Master CHPS of Paris Saclay

Gaël Thomas

Definition of a process

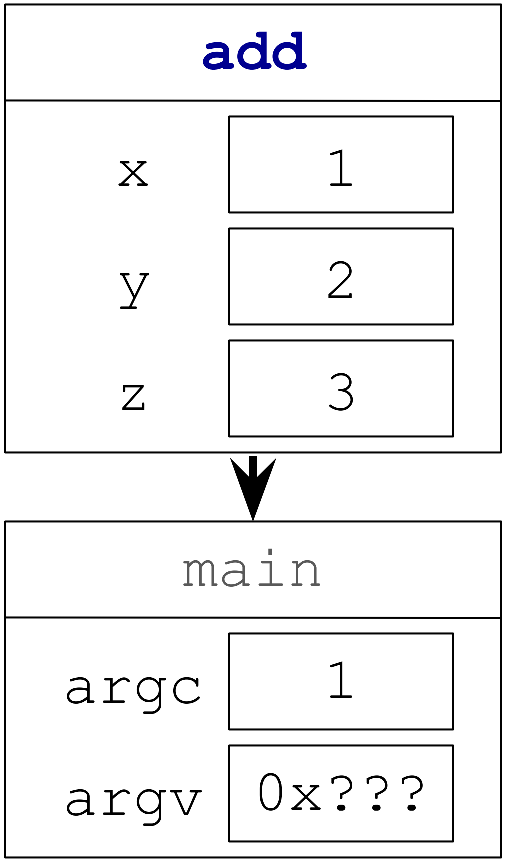
- A process is a **running instance** of a program
  - Allow the execution of different programs in parallel (e.g., fortnite and chrome)
  - Allow the execution of the same program multiple times (e.g., two instances of emacs for two different users)

- The operating system is in charge of
  - Managing the **life cycle** of the processes (start, stop)
  - Allowing processes to **communicate** (signals, pipes, sockets…)
  - (Regularly) **running** the processes on the processors
  - **Isolating** the processes (no shared memory by default)

- A process is roughly a virtualization of a complete machine

From the call frame to the thread

- During execution, when it starts a function, the process creates a **call frame**
  - Contains
    - the arguments of the functions
    - its local variables
    - a link to the caller
  - Frees the call frame at the end of the call



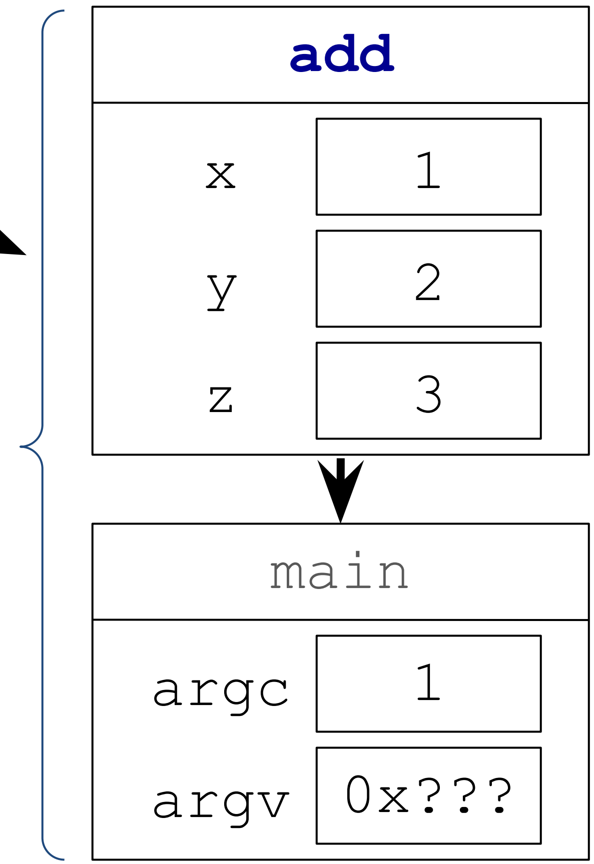

```
int add(int x, int y) {
  int z = x + y;
  return z;
}

int main(int argc, char** argv) {
  printf("%d\n", add(1, 2));
}
```

From the call frame to the thread

- During execution, when it starts a function, the process creates a **call frame**
  - Contains
    - the argu
    - its local v
    - a link to the caller
  - Frees the call frame at the end of the call

Stack
(of call frames)



Next instruction to be executed

```
int add(int x, i
  int z = x + y;
  return z;
}

int main(int argc, char** argv) {
  printf("%d\n", add(1, 2));
}
```

From the call frame to the thread

- During execution, when it starts a function, the process creates a **call frame**
  - Contains
    - the arguments of the functions
    - its local variables
    - a link to the caller
  - Frees the call frame at the end of the call

- A thread is an execution context executable by a CPU
  - A stack of call frames (e.g., main calls add)
  - The next instruction to be executed (e.g., the return z)

- An operating system schedules the threads on the CPUs

Processes and threads

- A process contains
  - A memory (data, code, heap)
  - One or more threads (each with its stack and its next instruction)
- A process always starts with a single thread
- A process may create more threads to increase parallelism
  - The operating system can then schedule the multiple threads on the multiple CPUs in parallel
- A process dies when its last thread terminates

Thread identification

- Type that can hold a thread identifier: `pthread_t`
- Identifier of the running thread: `pthread_t pthread_self()`

Thread creation

```
int pthread_create(pthread_t* tid, pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)
```

- Create and start a new thread
- The new thread starts in the function `start_routine`
- The `start_routine` function receives the argument `arg`
- `pthread_create` fills `*tid` with the identifier of the new thread
- `pthread_attr_t` gives attribute (scheduling, stack pointer…)

```
void* f(void* arg) { printf("f is running\n"); return NULL; }

int main(int argc, char** argv) {
  pthread_t tid;
  pthread_create(&tid, NULL, f, NULL);
  printf("main is running in parallel with f\n");
}
```

Thread termination

- After an explicit call to `pthread_exit(void* retval)`
- At the end of the `start_rountine`
- The system also terminates **all** the threads of a process when:
  - The main function returns
  - One of the threads of the process calls `exit`

Waiting the termination of a thread

- `int pthread_join(pthread_t thread, void** pretval);`

```
void* f(void* arg) {
  printf("f is running\n");
  return (void*)0x42;
}

int main(int argc, char** argv) {
  pthread_t tid;
  void* retval;
  pthread_create(&tid, NULL, f, NULL);
  printf("main is running in parallel with f\n");
  pthread_join(tid, &retval);
  printf("f terminated with retval %p\n", retval);
}
```

Detached mode

- By default, a thread is in the **joinable mode**
  - When the thread dies, the system keeps its return value, which consumes system resources
  - Another thread can use `pthread_join` to retrieve this value

- In **detached mode**
  - The system immediately frees all the system resources used by a thread when it exits
  - It is impossible to retrieve its return value

- You can change the mode of a thread to detached
  - Through a call to `pthread_detach(pthread_t tid)`
  - By using the `pthread_attr_t` in `pthread_create`

Shared variables and inconsistencies

- The threads of a process share the same memory
  - When a thread modifies a variable, the other threads see the modification
  - Concurrent accesses may lead to inconsistencies

```
int balance = 1000;
```

Thread 1

```
a. void credit() {
b.   int tmp = balance;
c.   tmp = tmp + 100;
d.   balance = tmp;
e. }
```

Thread 2

```
f. void debit() {
g.   int tmp = balance;
h.   tmp = tmp - 1;
i.   balance = tmp;
j. }
```

- Possible schedule: fg abcde hij => the credit of 100 is lost!

Principle to avoid inconsistencies

- Prevent two sections of code that access the same shared variables to execute at the same time
  - We say that the sequences of instructions are in **mutual exclusions**
- Definition: **a critical section** is a section of code in mutual exclusion
  - Critical sections execute entirely one after the other
  - We say that a critical section executes **atomically**
- A critical section is often in mutual exclusion with itself

Implementation of mutual exclusion

- Mutex: a lock in mutual exclusion
  - Two possible states: busy or free
  - At each time, only one thread can own (have marked as busy) the mutex

- A mutex provides two operations
  - Lock acquisition: waits if the lock is busy and then changes its state from free to busy
  - Lock release: marks the lock as free

- The two operations seem to execute atomically

Implementation of mutual exclusion

- Implementation:
  - `pthread_mutex_lock`: acquire a mutex
  - `pthread_mutex_unlock`: release a mutex

```
int balance = 1000;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Thread 1

```
void credit() {
  pthread_mutex_lock(&m);
  int tmp = balance;
  tmp = tmp + 100;
  balance = tmp;
  pthread_mutex_unlock(&m);
}
```

Thread 2

```
void debit() {
  pthread_mutex_lock(&m);
  int tmp = balance;
  tmp = tmp - 1;
  balance = tmp;
  pthread_mutex_unlock(&m);
}
```

Monitor

- Allows a thread to wait for a certain condition to become true
  - Built with a mutex and a variable condition

```
char* msg = NULL;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

Thread 1

```
void send() {
  pthread_mutex_lock(&m);
  msg = "Hello!";
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

Thread 2

```
void recv() {
  pthread_mutex_lock(&m);
  while(msg == NULL)
    pthread_cond_wait(&c, &m);
  printf("Message: %s\n", msg);
  pthread_mutex_unlock(&m);
}
```

Monitor

- Allow a thread to wait for a certain condition to become true
  - Built with a mutex and a variable condition

- Interface
  - Release `mutex`, sleep on `cond`, and re-acquire `mutex`
    `pthread_cond_wait(&cond, &mutex)`
  - Wake up one thread that sleeps on `cond`
    `pthread_cond_signal(&cond):`
  - Wake up all the threads that sleep on `cond`
    `pthread_cond_broadcast(&cond):`

To take away

- Thread life cycle
  - `pthread_create`: create a thread
  - `pthread_self`: return the thread identifier
  - `pthread_exit`: quit a thread
  - `pthread_join`: wait for the termination of a thread

- Synchronization
  - `pthread_mutex_lock`: take a lock
  - `pthread_mutex_unlock`: release a lock
  - `pthread_cond_wait`: wait on a condition variable
  - `pthread_cond_signal`: wake up a thread that waits on a condition variable
  - `pthread_cond_broadcast`: wake up all the threads that wait on a condition variable