Programmation parallèle: Exécution parallèle de processus.

Introduction

Un thread est une unité d'exécution faisant partie d'un programme. Cette unité fonctionne de façon autonome et parallèlement à d'autres threads. Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leurs exécutions "simultanées".

Sur une machine monoprocesseur, c'est le système d'exploitation qui alloue du temps d'utilisation du CPU pour accomplir les traitements de chaque threads, donnant ainsi l'impression que ces traitements sont réalisés en parallèle.

Sur une machine multiprocesseur, le système d'exploitation peut répartir l'exécution sur plusieurs coeurs, ce qui peut effectivement permettre de réaliser des traitements en parallèle.

Le système d'exploitation va devoir répartir du temps de traitement pour chaque thread sur le ou les CPU de la machine. Plus il y a de threads, plus le système va devoir switcher. De plus, un thread requiert des ressources pour s'exécuter notamment un espace mémoire nommé pile.

Cependant, l'utilisation de plusieurs threads améliore généralement les performances, notamment si la machine possède plusieurs coeurs, car dans ce cas plusieurs threads peuvent vraiment s'exécuter en parallèle. Il est aussi fréquent que les traitements d'un thread soient en attente d'une ressource : le système peut alors plus rapidement allouer du temps CPU à d'autres threads qui ne le sont pas.

Thread() in Processing

You are likely familiar with the idea of writing a program that follows a specific sequence of steps — setup() first then draw() over and over and over again! A Thread is also a series of steps with a beginning, a middle, and an end. A Processing sketch is a single thread, often referred to as the “Animation” thread. Other threads sequences, however, can run independently of the main “Processing” sketch. In fact, you can launch any number of threads at one time and they will all run concurrently.

Processing does this all the time, whenever you write an event callback, such as serialEvent(), or captureEvent(), etc. these functions are triggered by a different thread running behind the scenes, and they alert Processing whenever they have something to report. This is useful whenever you need to perform a task that takes too long and would slow down the main animation’s frame rate, such as grabbing data from the network (XML, database, etc.) If a separate thread gets stuck or has an error, the entire program won’t grind to a halt, since the error only stops that individual thread. To create independent, asynchronous threads, you can use the thread() function built into Processing.

```
void setup() {
  size(200,200);
  // This will tell someFunction() to execute now as a separate thread
  thread("someFunction");
}
```

```
void draw() {

}

void someFunction() {
  // This function will run as a thread when called via
  // thread("someFunction") as it was in setup!
}
```

The thread() function receives a String as an argument. The String should match the name of the function you want to run as a thread. While using the thread() function is a very simple way of getting an independent thread, it is somewhat limited.

Following is an example draws a loading bar in the “animation” thread that reports progress on another thread(). This is a nice demonstration, however, it would not be necessary in a sketch where you wanted to load data in the background and hide this from the user, allowing the draw() loop to simply continue.

```
/**
 * Thread function example
 * by Daniel Shiffman.
 *
 * This example demonstrates how to use thread() to spawn
 * a process that happens outside of the main animation thread.
 *
 * When thread() is called, the draw() loop will continue while
 * the code inside the function passed to thread() will operate
 * in the background.
 *
 * For more about threads, see: http://wiki.processing.org/w/Threading
 */

// This sketch will load data from all of these URLs in a separate thread
String[] urls = {
  "http://processing.org",
  "http://www.processing.org/exhibition/",
  "http://www.processing.org/reference/",
  "http://www.processing.org/reference/libraries",
  "http://www.processing.org/reference/tools",
  "http://www.processing.org/reference/environment",
  "http://www.processing.org/learning/",
  "http://www.processing.org/learning/basics/",
  "http://www.processing.org/learning/topics/",
  "http://www.processing.org/learning/gettingstarted/",
  "http://www.processing.org/download/",
  "http://www.processing.org/shop/",
  "http://www.processing.org/about/",
  "http://www.processing.org/about/people"
};

// This will keep track of whether the thread is finished
boolean finished = false;
```

```
// And how far along
float percent = 0;

// A variable to keep all the data loaded
String allData;

void setup() {
  size(640, 360);
  smooth();
  // Spawn the thread!
  thread("loadData");
}

void draw() {
  background(0);

  // If we're not finished draw a "loading bar"
  // This is so that we can see the progress of the thread
  // This would not be necessary in a sketch where you wanted to load data in the background
  // and hide this from the user, allowing the draw() loop to simply continue
  if (!finished) {
    stroke(255);
    noFill();
    rect(width/2-150, height/2, 300, 10);
    fill(255);
    // The size of the rectangle is mapped to the percentage completed
    float w = map(percent, 0, 1, 0, 300);
    rect(width/2-150, height/2, w, 10);
    textSize(14);
    textAlign(CENTER);
    fill(255);
    text("Loading", width/2, height/2+30);
  }
  else {
    // The thread is complete!
    textAlign(CENTER);
    textSize(24);
    fill(255);
    text("Finished loading. Click the mouse to load again.", width/2, height/2);
  }
}

void mousePressed() {
   thread("loadData");
}

void loadData() {
  // The thread is not completed
  finished = false;
  // Reset the data to empty
  allData = "";
```

```
  // Look at each URL
  // This example is doing some highly arbitrary things just to make it take longer
  // If you had a lot of data parsing you needed to do, this can all happen in the background
  for (int i = 0; i < urls.length; i++) {
    String[] lines = loadStrings(urls[i]);
    // Demonstrating some arbitrary text splitting, joining, and sorting to make the thread take longer
    String allTxt = join(lines, " ");
    String[] words = splitTokens(allTxt, "\t+\n <>=\\-!@#$%^&*(),.;:/?\"\'");
    for (int j = 0; j < words.length; j++) {
      words[j] = words[j].trim();
      words[j] = words[j].toLowerCase();
    }
    words = sort(words);
    allData += join(words, " ");
    percent = float(i)/urls.length;
  }

  String[] words = split(allData," ");
  words = sort(words);
  allData = join(words, " ");

  // The thread is completed!
  finished = true;
}
```