COMPSCI 340 S2 C 2015
SOFTENG 370 S2 C 2015
Operating Systems
Assignment 1 – A Stack Dispatcher
7% of course mark
Due Monday 17th of August 9:30 pm

Introduction

There are many types of dispatcher. That is the component of the process scheduler which selects from amongst the runnable processes which process should be allocated to which processor. This assignment deals with a stack based (LIFO) dispatcher. It came from an idea for a simple operating system for a single user on a very old, very slow computer. The requirement was for the scheduling of processes to be as simple as possible, without even the benefit of timeslices. On the assumption that the most recently started processes were the ones the user wanted to give priority to, the list of ready to run processes was maintained as a stack.

Details

All processes which are ready to run (i.e. they are not waiting for any resource except the processor) are maintained in a stack. The process on the top of the stack is the process currently running. When the currently running process needs to wait for a resource or it finishes running it leaves the stack. This means the new process at the top of the stack becomes the running process.

When a new process is created it gets pushed on to the top of the stack, becoming the new currently running process. This means that the process that was previously running must be paused. It remains on the stack, and the new process starts running.

Under this scheme it is easy for a process never to finish. As long as there is a stream of newly arriving processes the processes on the bottom of the stack may never run. Of course if this happens it means that there is too much work for the system to handle and many other types of dispatcher exhibit this behaviour as well. In this case, as there is only going to be one person using the system at any time the user can be given direct control over the scheduling of processes. So if the user really wants a particular process in the stack to be the currently running one he or she can select the process and it gets moved to the top of the stack. In this way the dispatcher is not truly a stack, it only works as a stack without the user's intervention.

The scheduling is actually pre-emptive, but not by clock interrupts. The only things that can stop a happily running process from continuing are actions by the user.

All interactions with the program are carried out via a simple menu system. Each menu option consists of a single letter e.g. "n" starts the "(n)ew" operation. This is designed so that the program can be tested by redirecting commands from a file. In the rest of this document such commands are simply written as (n)ew.

User actions causing dispatching

The user can start a new process running. The way this is done is by selecting (n)ew and then either (i)nteractive or (b)ackground. These two process types, background and interactive are explained later. The new process is created and immediately pushed on the top of the stack, displacing any process that is currently at the top of the stack and hence running.

The user can pause a running process and continue with another process by selecting (t)op and typing the process identifier for a process on the stack. This causes the currently running process to be paused and the selected process to move from its position to the top of the stack. All elements further up the stack move towards the bottom of the stack to fill in the gap made by the newly running process.

Another user action that causes a new process to be dispatched is typing the return/enter key after typing in data to a process waiting for input. The process to receive input is selected using the (f)ocus command. The process that was waiting for input moves to the top of the stack and becomes the currently running process.

And lastly if the user selects (k)ill the currently running process is killed and removed and the next process on the stack resumes running.

Process types

In a real operating system there may well be many different types of processes. In this system there are just two types depending on whether a process requires input from the user or not.

Both types of process in this assignment simulate doing some work by printing a "*" to the screen and by going to sleep (see the main_process_body method of the Process class). All processes in this assignment are implemented using Python threads (or pthreads if you really want to do the assignment in C).

Background process

A background process cycles through the main_process_body method a random number of times (between 10 and 160 inclusive). It requires no input from the user and will always stay in the ready to run stack until it is finished (or killed).

Interactive process

An interactive process is very similar to a background process but instead of executing for a random number of times it repeatedly asks the user to type in the number of times it should call the main_process_body method before waiting for more input.

The input cycle

The interactive processes call the IO_sys read method to get input from the user. This call must stop the process from running and remove it from the stack moving it to the first empty position in the list of waiting processes. The process does not automatically get the focus of the keyboard input. In order to choose the interactive process for keyboard input there is the (f)ocus command. This command asks the user to type in the process identifier of the process.

The process currently receiving keyboard focus is shown by having the input cursor visible in its window panel.

When a process has the keyboard focus each character typed is sent to a buffer for the process (and also appears in the display area of the process). When the user types a newline (enter or return) the process gets put back on the top of the runnable stack (both visually and logically). The process then checks the value returned from the read. If the value is -1 the process finishes. If the value is greater than -1 the process runs for that many calls to the main_process_body before asking the user for more input. Throughout the assignment you may assume that only valid data will be input.

The menus

There are two menus and extra prompts which may appear at the top of the screen. The menu options are described here.

The main menu

(n)ew, (f)ocus, (t)op, (k)ill, (h)alt, (p)ause, (w)ait, (q)uit:

(n)ew
Create a new process. This command changes the menu to the create menu described below.

(f)ocus
Allocate the keyboard focus to a particular panel (and corresponding process). This command causes the prompt "Enter the number of the input process:" to appear at the top of the screen and the program input then waits for the user to enter the process identifier of a waiting interactive process, followed by the return/enter key. The cursor then jumps to the panel of that process and stays there until the user has typed a number followed by the return/enter key. After this the main menu is once again displayed at the top of the screen.

(t)op
Move a runnable process (already in the stack) to the top of the stack. This usually causes other processes to move towards the bottom of the stack to occupy the place where the selected process was. It obviously also causes the selected process to resume running as it is now at the top of the stack. This command causes the prompt "Enter the number of the process to move:" to appear at the top of the screen and the program input then waits for the user to enter the process identifier of a runnable process. The process is moved when the user presses the return/enter key. After this the main menu is once again displayed at the top of the screen.

(k)ill
Kill a process. As with (t)op and (f)ocus this command causes a prompt and waits for the user to enter the number of the process. The prompt is "Enter the number of the process to kill:". After the number is entered the process is killed and removed from the system. If the process was runnable this may require the stack to be rearranged. If the process was waiting it is just removed from the waiting set. After this the main menu is once again displayed at the top of the screen.

(h)alt
Halts the entire system for 10 seconds. This pauses both the currently running process and also stops the program from receiving input commands for 10 seconds. The reason for this command is to freeze the system when receiving commands from redirected files, thus enabling you and the markers to examine output.

(p)ause
Pause from receiving further commands for 10 seconds. Unlike (h)alt this allows runnable processes to keep running but it pauses input from redirected files so the processing can be observed.

(w)ait
Allow all currently runnable processes to run to completion (or until they ask for further input) and then stop the system. After this command the program stops when there are no more processes in the runnable queue.

(q)uit
Shut the system down. The program stops regardless of the current states of the processes.

The create menu

When the user selects (n)ew in the main menu this menu is displayed.

(i)nteractive, (b)ackground, (c)ancel

(i)nteractive
The process being created is interactive. After creating the process on the top of the runnable stack the main menu is displayed. Interactive processes don't stay on the top of the stack for long because they wait almost immediately.

(b)ackground
The process being created is a background process. After creating the process on the top of the runnable stack the main menu is displayed.

(c)ancel
Don't create a new process. Return to the main menu.

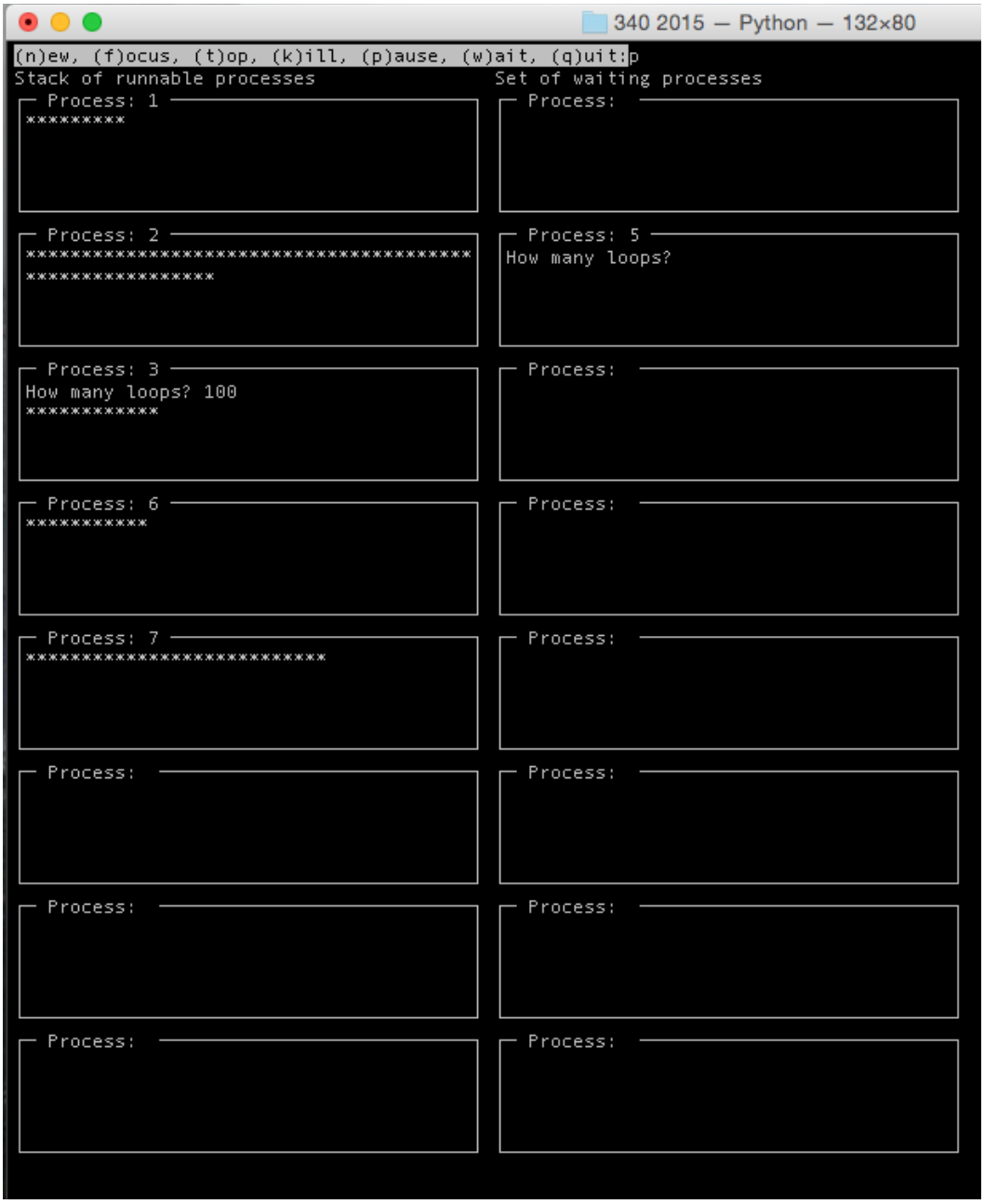

figure 1

The screen

When the program is run it shows two columns of panels (see figure 1). The left side panels show the stack of runnable processes (these processes are not waiting for input, they only need the processor). The right side panels display all interactive processes currently waiting for input from the user.

The currently running process is process 7 since it is at the top of the runnable stack. If you were watching this run it would be obvious as it is the only process adding more stars to the screen. This demonstrates that the stack grows downwards in this assignment so the top of the stack is actually at the bottom.

There are four background processes in the figure, processes 1, 2, 6 and 7, and two interactive processes, processes 3 and 5. Process numbers are allocated sequentially so Process 4 must have already finished. Process 3 is an interactive process which is in the runnable stack because it is not waiting for input. You can see the user has typed in 100 and the process is working on that. Process 5 however is waiting for user input and is in the set of waiting processes. When an interactive process is made to wait it is allocated the first screen panel in the right hand column which is currently not being used. So we can see that when process 5 was moved to the waiting set there must have been another interactive process (probably 3) already waiting.

What you have to do

Write a program which works as described in this document. You can do this in any language you like which runs on Linux in the labs but since I provide you with a number of Python modules which already provide some of the functions including all of the user interface it will be much easier if you do it in Python.

Feel free to modify any of the distributed code.

SOFTENG 370 students

SoftEng370 students have an extra component. You must dispatch two processes at a time, simulating a dual core machine. Everything should work the same as described above but with the top two processes in the runnable stack both running.

Useful info

To run the program you type: `python3 a1.py`

Alternatively if you make a1.py executable then you can run it with: `./a1.py`

It is likely that your program will occasionally crash leaving the terminal window in an inconsistent state. When this happens type `reset`. This will fix your terminal window.

When logging into Linux in the lab I recommend choosing "Ubuntu" at the login window, you may need to click on the GNOME footprint icon to the right of the login panel.

It looks as though you may be limited to Xterm or UXterm as terminal programs and gedit as an editor (or vi for the particularly brave). Of course on your own version of Linux you can use any terminal programs and editors that you like.

Apart from the places in the released code you are NOT allowed to use the Python `time.sleep` function.

Questions

Answer the following questions. Put the answers in a simple text file called `a1Answers.txt`.

1. Look at the list of processes, arrival times and burst times on page 270 of the text. Draw a Gantt chart (ASCII art is suitable) showing the order of execution for these processes if they were scheduled by a LIFO stack-based dispatcher. What would the average waiting time be in this case?
2. The code commented out in Process.main_process_body method checks the process' state. A real implementation wouldn't require this. Explain why a real system doesn't require this and why a Python thread implementation does. Alternatively explain how you could get rid of this even when using Python threads.
3. (For SOFTENG370 students only). Does the solution of maintaining one stack shared by multiple processors scale well? Explain.

Submitting the assignment

Make sure your name and upi is included in every file you submit.

Use the assignment drop box to submit.

Any work you submit must be your work and your work alone – see the information on academic integrity http://www.auckland.ac.nz/uoa/home/about/teaching-learning/honesty.

Marking guide

The markers will redirect data from text files as the standard input. Your program must work with input both from the keyboard and from redirected input. You will not receive any marks for a section which does not produce output (even if it works perfectly behind the scenes).

A breakdown of the marking guide will be available shortly.