Agorithmic Data Science: Processes and Concurrency

Ian Wakeman

School of Informatics
University of Sussex, Falmer
Brighton, UK

ianw@sussex.ac.uk

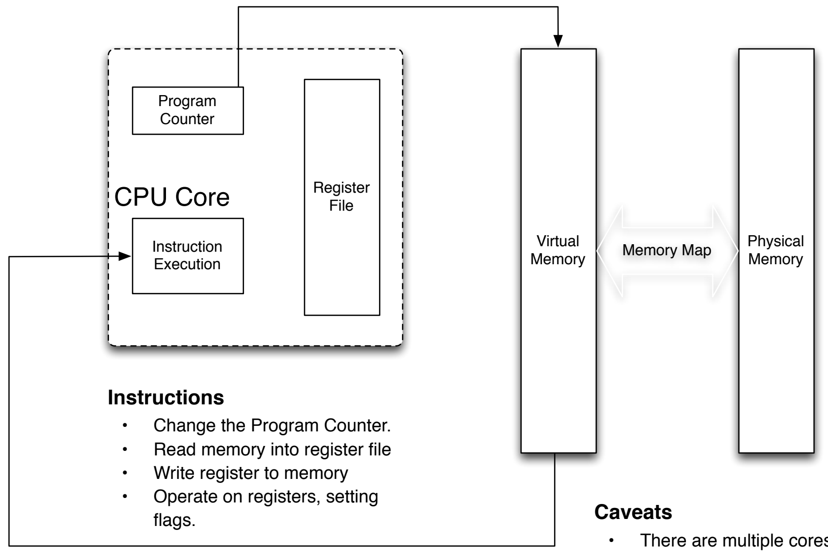
Introduction

- Big Data analysis can be greatly speeded up through parallel and distributed execution. . .
- But parallel and distributed programming is hard
- We will discuss why it is fundamentally hard
- Provide pointers as to how to make it easier

Processes and Concurrency

- The processor and program execution
- Definitions
- Control block
- Process creation
- Process scheduling
- Context switching
- Process termination

The Processor



Instructions

- Change the Program Counter.
- Read memory into register file
- Write register to memory
- Operate on registers, setting flags.

Caveats

- There are multiple cores
- Values read from memory are cached - caches live between memory and processor

Definitions

- Concurrency: The execution of different activities in parallel.
- The operating system must coordinate all activity on a machine, *e.g.* multiple users, I/O interrupts . . .
- This is a difficult task!
- We decompose the problem into smaller units of work — processes.

The process concept

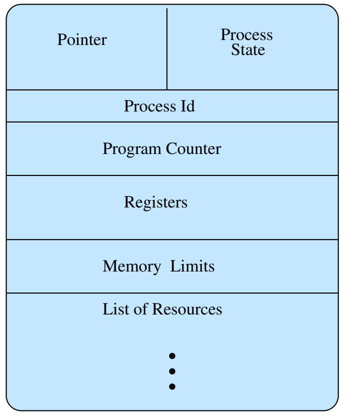
- A process is an abstraction which represents what is needed to run a single program.
- Two views:
  - Static — all of the resources required to run the program (including the code)
  - Dynamic — how the process' environment changes during sequential execution
- More concretely: a process contains information about the program it is to run, where in the execution sequence it is, the status of all the registers, a portion of memory allocated to it and a bunch of resources allocated to it.

Where do processes come from?

- Most operating systems provide some sort of Process Management module. User/application programs may make requests to this module using system calls.
- Have a guess what the *Win32* `CreateProcess` system call does?
- But user/applications programs must be running as a process before they can make a system call. How do they get created?
- The operating system creates a number of processes itself at boot time. Typically, one of these is a shell process.
- So what happens when a create process system call is requested?

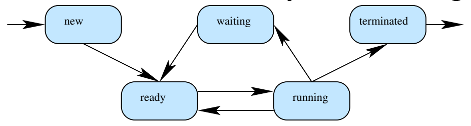
The Process Control Block

A data structure called the Process Control Block or PCB is created. This data structure contains all of the process' information:



Elements of the PCB

- The pointer is used to link processes together to form queues
- The process state is best described by the following diagram:



- The process Id is a unique integer identifier for the process.
- The program counter holds the address of the next instruction to perform.
- The registers are temporary storage used during execution (accumulators, stack pointers, etc).
- The memory limits determine the address space in which the process must run.

Process management — Creation

We examine the life cycle of a process:

Process creation

- Processes are always created by some parent process. The new process is referred to as a child process. The operating system always creates at least one process at start-up (often called `init`).
- The child inherits some, all, or none of its parent's resources.
- The parent and child can run concurrently (*cf.* Unix `fork`) or the parent waits on the child to terminate.
- The child's address space is a clone of the parent's (*cf.* Unix `fork`) or the child is loaded with a new program (VMS, Windows NT).

Process queues

- Once a process has been created the Process Manager places it in the ready queue.
- There are a number of different queues in the operating system.
- job queue — a list of all of the processes in existence (including those on disk)
- ready queue — a list of all of the processes currently in memory and ready to execute immediately
- device queues — a list of all of the processes waiting for I/O service from a particular device

How do processes move between these queues?

Process management — Scheduling

- Short term — this scheduler allocates the CPU to processes in the ready queue. Must be incredibly fast as this is invoked very frequently.
- Long term — sometimes, in large systems, not all processes are currently in memory. They are stored temporarily on hard disk. This scheduler chooses processes from the job queue which are ready to execute but not yet in memory and places them in the ready queue. This can be very slow as disk transfer rates are slow.
- Executing processes may interrupt, request I/O, fork a child process, or terminate. Any of these will cause the process to be yield the CPU and rejoin the appropriate queue.

Process management — Scheduling

- Device controllers interrupt the CPU to inform the scheduler to move processes from device queues back to the ready queue
- Processes can be
  - I/O bound — perform a lot of I/O
  - CPU bound — perform a lot of computation
- A good long-term scheduler chooses a suitable mix of the two.

Context switch

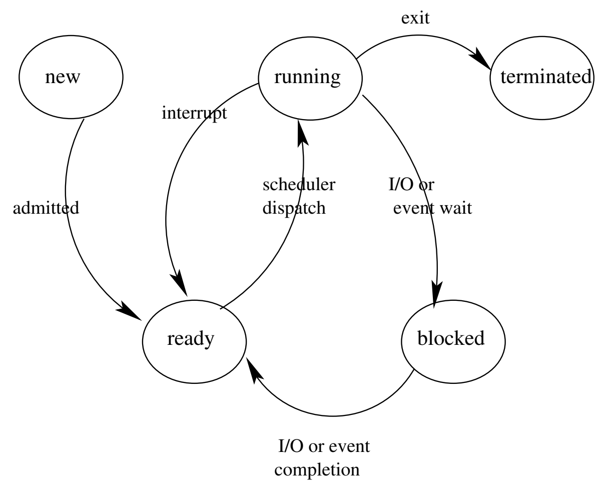
- How does the scheduler actually allocate the CPU to a process in the ready queue?
- Save the context of the currently executing process in its PCB. That is, store the value of registers, process state, program counter etc
- Load the context of the new process from its PCB and let the CPU point at the new process
- This is referred to as Context switching. Note that this operation is pure overhead. The actual time this takes relies heavily upon the hardware support (multiple register sets, high-speed cache) and memory management. Typically the context switch will take between 1 and 1000 microseconds.

Process management — Termination

There are two main ways for processes to terminate

- A process can terminate itself by executing a termination system call (in *Unix* this is `exit`).
- Parents may terminate their child processes (in Unix by using the `abort` system call with the child's process id).
- But what happens to the child processes when the parent is terminated?
- The children are also terminated — known as cascading termination
- Another parent (*e.g.* grandparent, or *init*) is allocated to them

Process State Transitions



The Kitchen Processor

Can we regard a shared kitchen as a process scheduling problem?
Yes!

Kitchen Programs

ThirstyStudent

1. Boil kettle
2. Add teabag to cup
3. Add water to cup
4. Wait till drawn
5. Add milk

DepressedStudent

1. Add half pint milk to bowl
2. Add Angel Delight to bowl
3. While not ready, whisk

HungryStudent

1. Add bread to toaster
2. Wait
3. Spread butter on toast

Simple Process Context Switching

Context switching for simple processes - see the powerpoint animation for simple processes.

The need for threads

- Suppose the user invokes a *word processor*.
- This is a complicated task involving many roles: *e.g.*
  - input
  - rendering
  - justification
  - spell checking
- Using a single sequence of instructions is inefficient and difficult to program for.
- Could run multiple small programs, each performing one of these roles — the OS could sequence these concurrently. But these programs should be grouped together some how.

Two process views

Recall that we said that we could view processes statically:

- Related resources are grouped together

and dynamically:

- The sequence, or *thread* of execution is recorded

This suggests

> *Processes are used to group resources together; threads are the entities scheduled for execution on the CPU*

Why not allow processes to have *multiple* threads of control?

The Thread concept

A thread is a "lightweight process."
It has its own

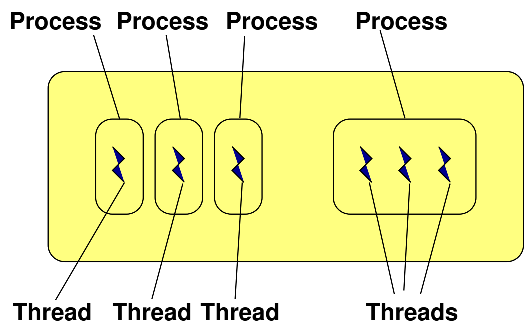
- Identifier, Program counter, Register set, Execution stack

but shares within a single process its

- code section, data section and resources

with its peer threads.
The notion of process described in the last lecture contained only a single thread of control. We extend the notion to include processes with multithreading and sometimes refer to these as "heavyweight processes."

Multithreaded processes depicted



Threading in Python

```
import threading
import time

def loop1_10():
    for i in range(1, 11):
        time.sleep(1)
        print(i)

threading.Thread(target=loop1_10).start()
```

Don't use threading in Python

- The Python runtime manages multiple threads *under the hood*
- Implementing threads directly will likely interfere with this implementation
- Use higher level abstractions for parallelisation and distribution
- Other languages (Java, Erlang, Scala) are very good for threads

Introduction to Concurrency

- Motivation
- Critical Section
- Locks
- Semaphores
- Synchronization in Python

The Producer Consumer Problem

- Producer puts things in a shared buffer, consumer takes them out.
- Need synchronisation between processes for access to the shared finite buffer.
- *Example* Coke machine - producer is delivery person, consumers are students and faculty.

Coke Machine Code

```
import multiprocessing
def producer:
    while True:
        if(!machine.full())
            machine.put(coke)

def consumer:
    while True:
        if(!machine.empty())
            machine.get().drink()

deliveryPerson = Process(target=producer,args=(machine,))
student = Process(target=consumer,args=(machine,))
```

What goes wrong?

- If students get switched out between checking to see if machine is empty and getting coke, they can be disappointed.
- If delivery man is switched out between checking machine isn't full, and inserting coke, may get overflow

Correctness Constraints

- Consumer must wait for producer to fill buffer if they are empty. (Scheduling constraint)
- Producer must wait for consumer to empty buffer if full (scheduling constraint).
- Only one thread can manipulate buffer at a time (mutual exclusion).

The Critical Section Problem

The Producer/Consumer problem is actually an instance of a more general problem:

Give *n* concurrent threads, $T_1, \ldots, T_n$ of the form:

```
repeat
  entry code
    critical section
  exit code
    remainder section
until
```

ensuring the following criteria:

Critical section problem — constraints

- Mutual exclusion — If thread T_i is executing in its critical section, then no other threads can also be executing in their critical sections.
- Progress — If no thread is executing in its critical section and there exist some threads that wish to enter theirs, then the selection of the thread that will enter the critical section next cannot be postponed indefinitely.
- Bounded Waiting — A bound must exist on the number of times that other threads are allowed to enter their critical sections after a thread has made a request to enter its critical section and before that request is granted.

We assume that all threads execute at some non-zero speed but make no assumptions concerning the relative speeds of the threads.

Synchronization Tool 1: Locks

- A lock is a variable which can be free or held.
- When the lock is free, a thread attempting to get the lock will get and hold the lock.
- When the lock is released, if one or more threads are blocked on the lock, a thread is given the lock and becomes ready.
- When the lock is already held by another thread, a thread attempting to get the lock will join in a queue of threads blocked on the lock, and move from the ready state to the blocked state.
- Queue of blocked threads is generally first come, first served

Synchronization Tool 2: Semaphores

- *Semaphores* provide atomic operations to *increment*, and to *decrement or block*
- The *V* or *release* operation increments an integer counter
- The *P* or *acquire* operation will decrement the integer counter or block until the counter can be decremented
- Other synchronization tools are available, such as monitors and so on

Threaded Producer Consumer with Semaphores

```
def producer:
    while True:
        fullSemaphore.acquire()
        lock.acquire()
        machine.put(coke)
        lock.empty()
        emptySemaphore.release()

def consumer:
    while True:
        emptySemaphore.acquire()
        lock.acquire()
        machine.get().drink()   # Shouldn't do long
        lock.release()          # things in critical
                                # sections
        fullSemaphore.release()
```

Synchronized Data Structures

- Correct synchronization is hard. . .
- So let experts do it for you
- Language runtimes and libraries are built to be thread safe, such as the *synchronized Queue* class, or the *Multiprocessing Queue* class in Python
- Previous code has the locks and sempahores built into the Queue class

Conclusions

- Processes are the base abstraction for running code, with multiple threads of control per process
- The Process life cycle is key to understanding parallel programs
- Concurrency is hard to get correct and should be left to the experts