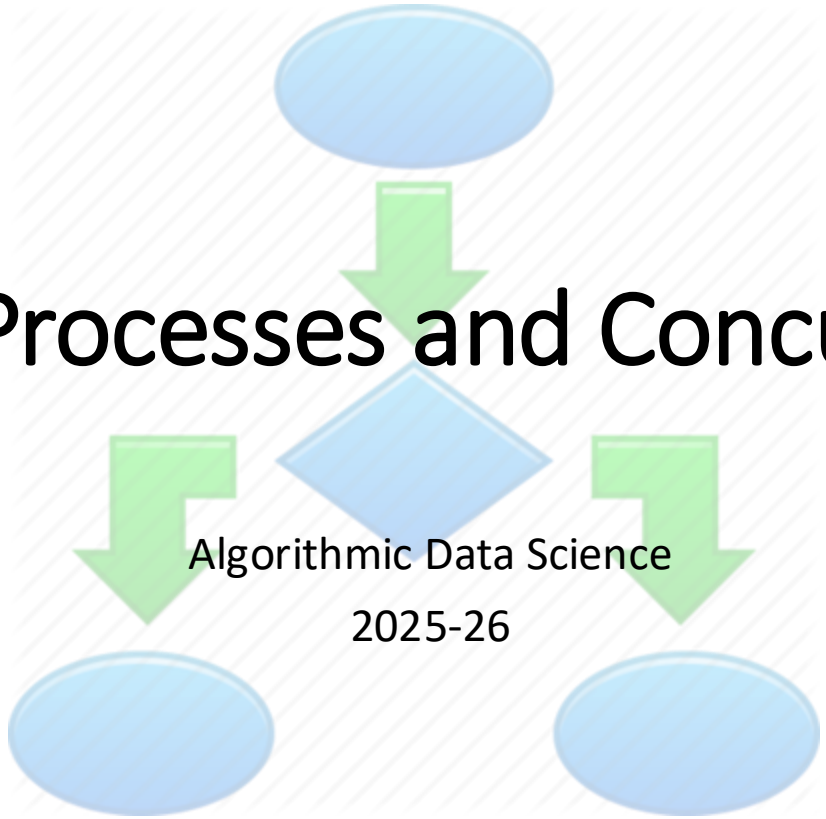
Week 4: Processes and Concurrency



Algorithmic Data Science

2025-26

UNIVERSITY OF SUSSEX

Dr Adam Barrett

Warm up

- Consider the list [7, 1, 8, 4, 3, 6]. If the algorithm *insertion sort* is applied to the list, the list will be updated several times, until the items on the list are arranged in ascending order. Write down each of these updated versions of the list.

[7, 1, 8, 4, 3, 6]

[1, 7, 8, 4, 3, 6]

[1, 7, 4, 8, 3, 6]

[1, 4, 7, 8, 3, 6]

[1, 4, 7, 3, 8, 6]

[1, 4, 3, 7, 8, 6]

[1, 3, 4, 7, 8, 6]

[1, 3, 4, 7, 6, 8]

[1, 3, 4, 6, 7, 8]

First assessment

- **Fri 31st Oct**, available from 9am-8pm on Canvas. 15 multiple choice questions. You have 1 hour from when you start the quiz. **You must finish the quiz before 8pm**.
  - This is worth 10% of your mark for this module.

Main topics per week

Week	Topic
1	Data structures and data formats
2	Algorithmic complexity. Sorting.
3	Matrices: Manipulation and computation
4	**Processes and concurrency**
5	Distributed computation
6	Similarity
7	Map/reduce
8	Graphs/networks
9	Graphs/networks, PageRank algorithm
10	Databases
11	*independent study*

Weeks 4 & 5: Introduction

- The analysis of big data can be greatly sped up through parallel and distributed execution
- But parallel and distributed computations are complicated.

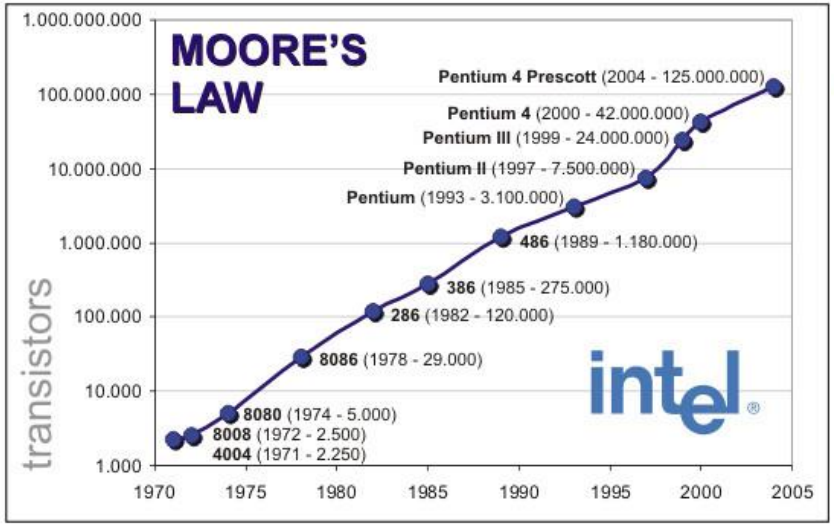
THIS WEEK:

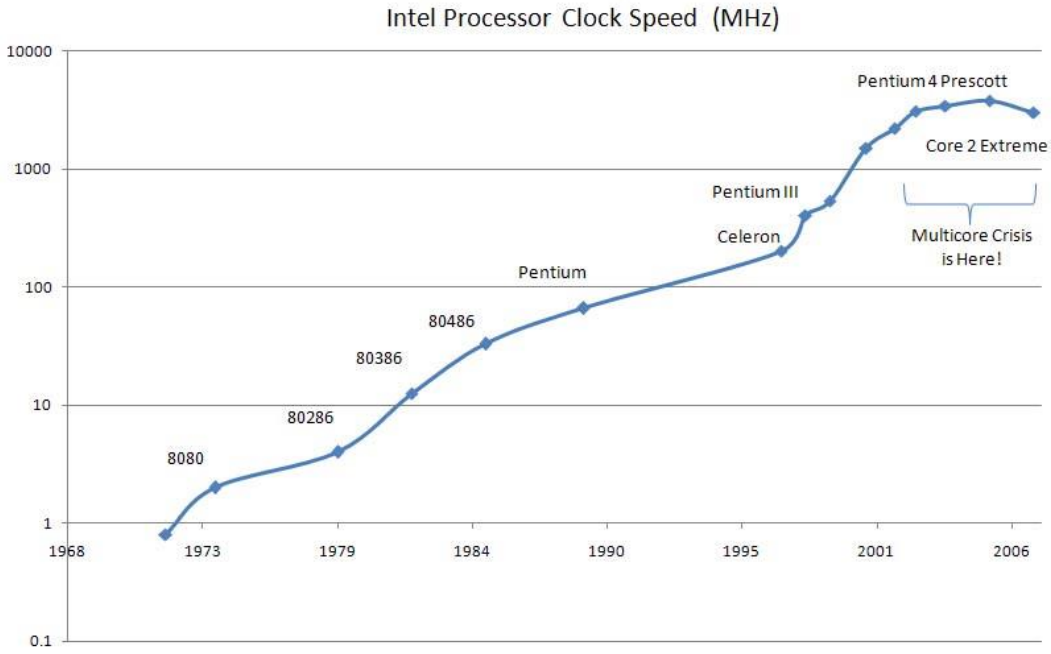
- CPUs, processes, threads and concurrency.

NEXT WEEK:

- Distributed computing and data centres.
- Security

Big data, with speed limits



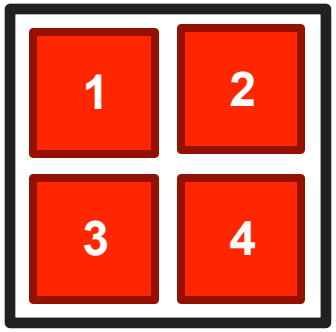


Picture from SmoothSpan

The anatomy of a processor

CPUs (central processing unit) nowardays come with multiple “cores”

The number of cores is the number of things the CPU can do *in parallel*



Central processing unit (CPU) cores

Have several components, e.g.:

Control Unit (CU): Controls flow of data and instructions within the CPU and to and from other devices.

Arithmetic Logic Unit (ALU): Performs arithmetic operations (addition, subtraction, multiplication, division) and logical operations (AND, OR, NOT) on data.

Registers: Small, high-speed memory units within the CPU.

- Store data, instructions, and intermediate results during processing.

Cache Memory: High-speed memory that stores frequently used data and instructions to speed up access.

- Helps reduce the time taken to fetch data from the main memory.

Kernel memory: Operating system stuff and process control blocks.

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Parallelism vs concurrency

Parallelism

Example: Walking and thinking at the same time

Each action driven by a different resources…

Concurrency

Example: Eating and talking “at the same time”

Each action driven by the same resources…

Definitions

Parallelism: the ability of a multi-core CPU to run multiple instructions *at the same time*.

Concurrency: the ability of a single CPU core to run multiple programs at the same time.

The OS must coordinate all activity on a machine (e.g. multiple users, I/O interrupts, etc), which is really hard.

We decompose the problem into smaller units of work — **processes**.

The idea of a *process*

A **process** is an abstraction which represents what is needed to run a single program.

A computer program is a passive collection of instructions typically stored in a file on disk.

A process is the execution of those instructions after being loaded from the disk into memory

Check the processes and cores on your computer

Windows (lab computers):

Ctrl-Shift-Esc, then Performance tab

Mac:

Core info-
About this Mac from Apple menu.
More Info.
System Report.

Processes: Activity Monitor in app menu.

A process contains information about:

- the program it is to run,
- where in the execution sequence it is,
- the status of all the registers,
- the portion of memory allocated to it,
- any other resources allocated to it,
- security attributes (owner and permissions)

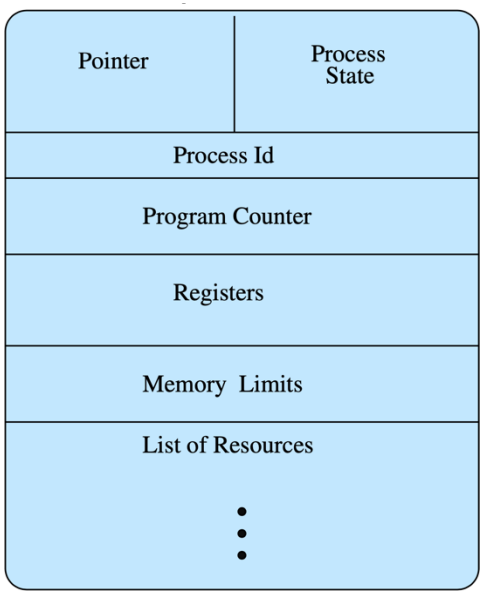
Where do processes come from?

- Most operating systems provide a **Process Management module**, which can create, run, and terminate processes.
- Users or programs make requests to this module using **system calls**.
- Only a process can make a system call. Hence, processes are created by other processes
- At boot time, the OS creates a number of initial processes. One of them is typically a **shell process** — via which users interact with the OS.

The Process Control Block

A *create process* system call creates a “Process Control Block” (PCB).

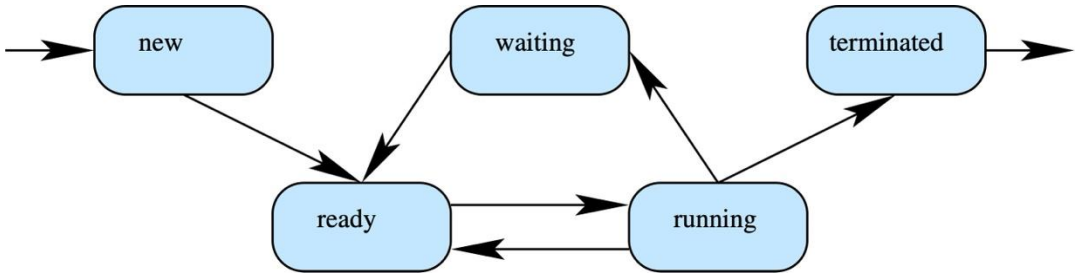
The PCB is a data structure that contains all the process’ information:



Elements of the PCB

The **pointer** is used to link processes together to form queues.

The **process state** determines its current condition, and evolves as follows:



The **process id** is a unique integer identifier for the process.

The **registers** are temporary storage used during execution.

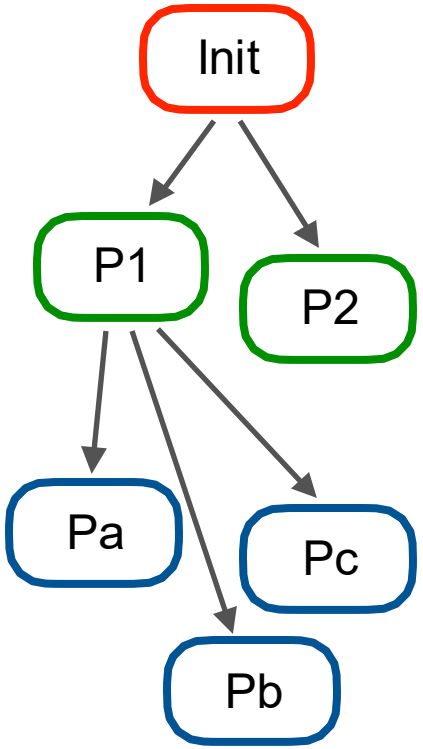
The **memory limits** determine the address space in which the process must run.

Process management — Creation

We examine the life cycle of a process:

Process creation

- Processes are always created by some **parent** process, with the new process being referred to as a "**child process**."
- The OS always creates at least one process at start-up — often called *init*.
- The child inherits some, all, or none of its parent's resources.
- Two modes: either the parent and child can run concurrently, or the parent can wait until the child terminates.



Process queues

There are a number of different queues in the operating system

- **Job queue** — a list of all the process in existence (including those in disk)
- **Ready queue** — a list of all 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.

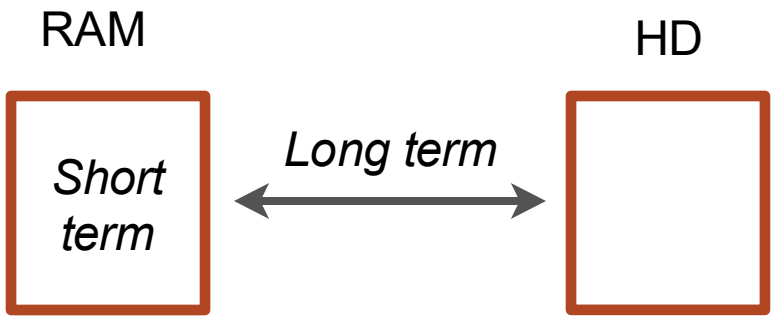
How do processes move between queues?

Process management — Scheduling

Scheduling of processes is done via a number of instances:

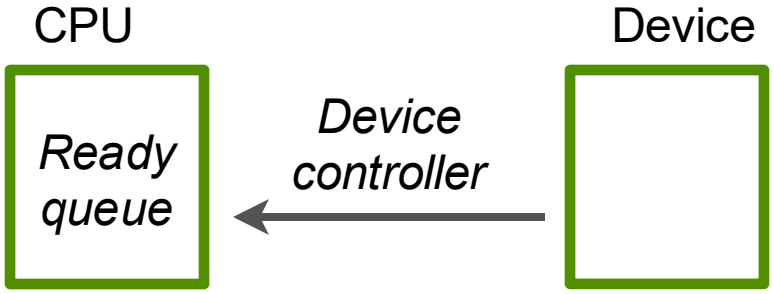
Short term — this scheduler allocates the CPU to processes in the *ready queue*. Is **very** fast, as it is involved frequently.

Long term — in large systems not all processes are kept in memory, but some 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 slow (as disk transfer is usually slow).



Process management — Scheduling

- The scheduler will switch processes from time to time.
- The execution of a process may arrive to interrupts, request I/O, fork a child process, or terminate. Any of these will cause the process to yield the CPU and rejoin the appropriate queue.
- 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 input/output
  - **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 allocate the CPU to a new process in the ready queue?

- First, saves the **context** of the currently executing process in its PCB (i.e. it stores the value of registers, process state, program counter, etc)
- Then, it loads the context of the new process from its PCB, and makes the CPU point at the new process

This is known as "**context switching**."

Note: 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

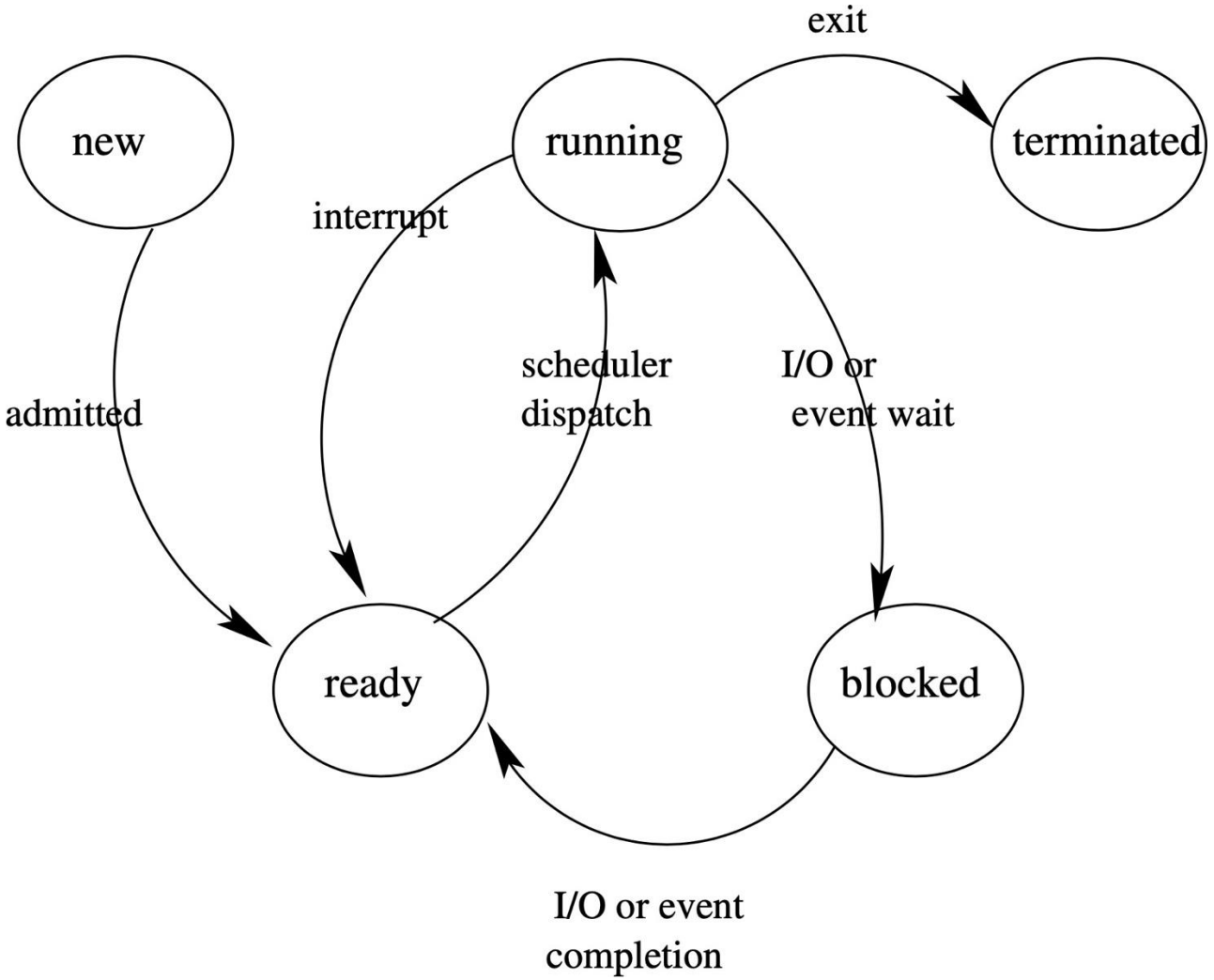
There are two main ways for a process to terminate:

- 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? There are two options:

- The children are also terminated — known as "cascading termination"
- Another parent (e.g., grandparent, or init) is allocated to them

Process State Transitions



Metaphorical example: 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

Context switching in lab exercises (Unit 2 lab)

- I wanted you to run insertion sort for long lists, to see the large n behaviour, but if you do it for short lists you can see the effects of context switching:

The need for threads

Consider the workings of a *word processor.*

- This is a complicated task involving many roles, including:
  - Receiving input
  - Rendering images
  - Calculating justification
  - Performing spell checking
- Using a single sequence of instructions is inefficient, and difficult to program for.
- Another option is to have multiple small processes, each performing one of these roles — the OS could sequence them concurrently.
- However, these programs should be grouped together somehow…

The *Thread* concept

We allow process to have multiple “threads”. A **thread** is a lightweight process.

It has its own:

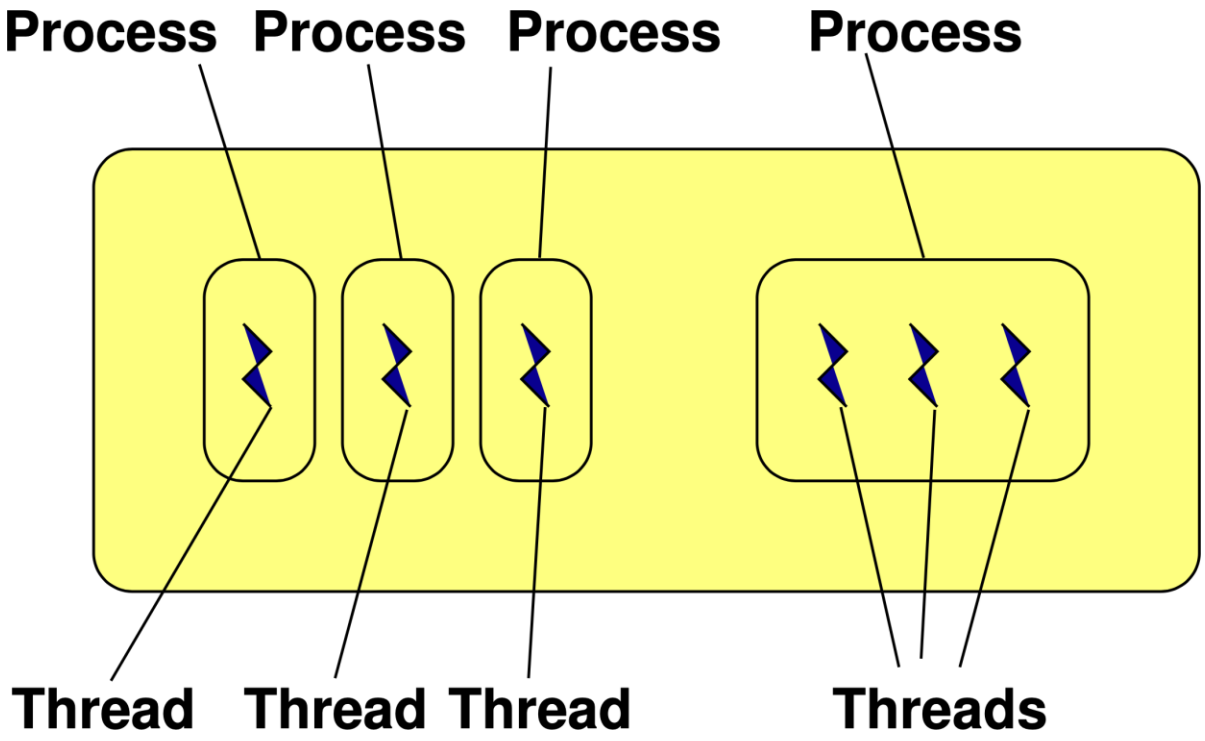
- Identifier
- Program counter
- Register set
- Execution stack

Nonetheless, it shares with its peer threads:

- Code
- Data
- Resources

Some processes have a single thread; others have multiple which leads to **multithreading**.

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

Coordinating processes — concurrency

The Producer Consumer Problem:

- A producer puts things in a shared buffer, a 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 can go wrong?

What are possible problems with this setup:

- 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.

How to fix this?

- 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

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

Given *n* concurrent threads, $T_1, T_2, \ldots, T_n$, of the form:

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

ensuring the following criteria:

Critical section problem — constraints

Criteria:

- **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.

Synchronisation 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.

Threaded Producer-Consumer with Locks

```
def producer:
   while True:
      lock.acquire()
      if(!machine.full())
          machine.put(coke)
      lock.release()
def consumer:
   while True:
       lock.acquire()
       if(!machine.empty())
          machine.get().drink()
       lock.release()
```

Basic python example

```
import threading
import time

my_lock = threading.Lock()

def loop1_15():
        for i in range(1, 11):
                time.sleep(1)
                print(i)
        with my_lock:
                for i in range(11, 16):
                        time.sleep(1)
                        print(i)

thread1 = threading.Thread(target=loop1_15).start()
thread2 = threading.Thread(target=loop1_15).start()
```

SUPPLEMENTARY

Synchronization Tool 2: Semaphores

Semaphores provide atomic operations to increment, to decrement, or block a process. They have two operations:

- The **release operation** increments an integer counter.
- The **acquire operation** will decrement the integer counter, or block until the counter can be decremented.

Semaphores are similar to locks, more flexible but more complicated.

Other more complex synchronisation tools are available, such as monitors.

SUPPLEMENTARY

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()
```

Synchronised data structures

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

  A computer typically executes many processes “at the same time” — which can be either in **parallel** (multicore) or **concurrently**.

- **Concurrency** is hard to get right; is not a bad idea to leave it to the experts.

A library has a large collection of n books. It has information about all the books in a dictionary, where the titles of the books are the keys and the values contain author, publisher, year, edition and ISBN number.

i. Information needs to be retrieved for 10 specific books. In O notation, what computation time is required to do this if the dictionary is stored in a hash table with a hash function for which there are no collisions?

(a) O(1) (b) O(log(n)) (c) O(n) (d) O(10)

ii. Information needs to be retrieved for 10 specific books. In O notation, what computation time is required to do this if the dictionary is stored as a binary search tree?

(a) O(1) (b) O(log(n)) (c) O(n) (d) O(10)

iii. Suppose that the data have been restructured into a list of lists. Now what is the computation time required to retrieve the information for 10 specific books?

(a) O(1) (b) O(log(n)) (c) O(n) (d) O(10)

A library has a large collection of n books. It has information about all the books in a dictionary, where the titles of the books are the keys and the values contain author, publisher, year, edition and ISBN number.

i. Information needs to be retrieved for 10 specific books. In O notation, what computation time is required to do this if the dictionary is stored in a hash table with a hash function for which there are no collisions?

(a) ***O(1)*** *(b)* *O(log(n))* *(c)* *O(n)* *(d)* *O(10)*

ii. Information needs to be retrieved for 10 specific books. In O notation, what computation time is required to do this if the dictionary is stored as a binary search tree?

(a) *O(1)* *(b)* ***O(log(n))*** *(c)* *O(n)* *(d)* *O(10)*

iii. Suppose that the data have been restructured into a list of lists. Now what is the computation time required to retrieve the information for 10 specific books?

(a) *O(1)* *(b)* *O(log(n))* *(c)* ***O(n)*** *(d)* *O(10)*