# Process

## Overview

* A process is a program in execution
* A process is more than the program code, which is sometimes known as the text section
* A process also includes the current activity, as represented by the value of the program counter and the contents of the processor’s registers
* A process includes:
* Stack: contains temporary data such as function parameters, return addresses, and local variables
* Heap: memory that is dynamically allocated during process run time.
* Data: contains global and static variables.
* Text: includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

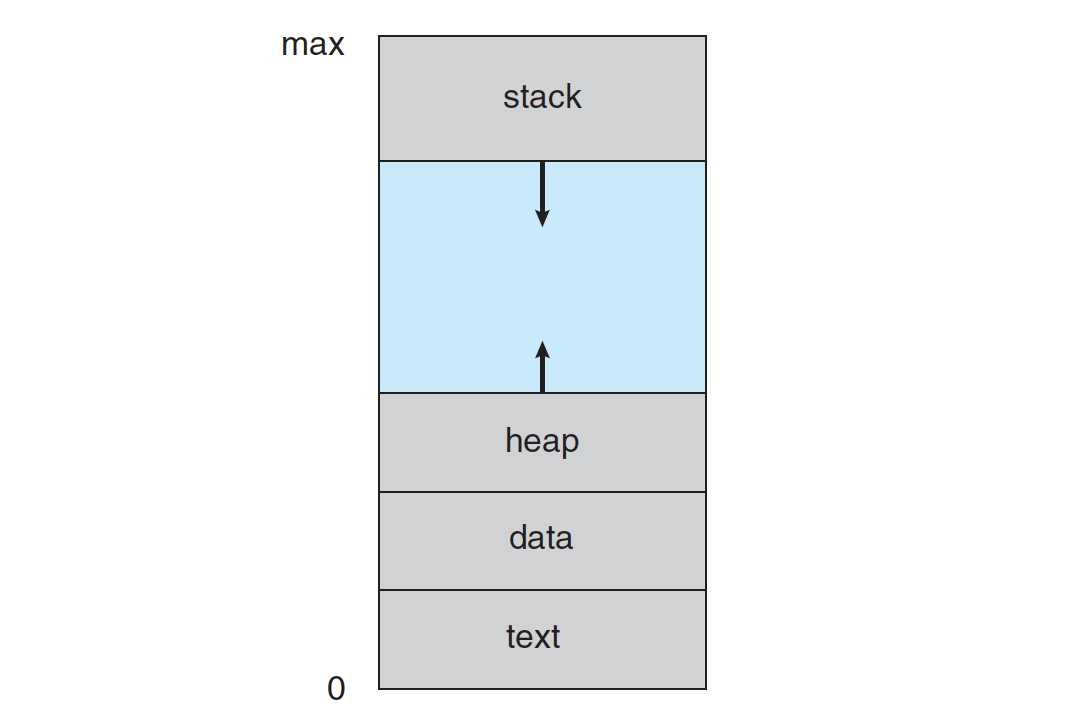


Figure 1. Process in memory

* A program is a ‘passive’ entity (executable file), process is a ‘active’ entity
* A program becomes a process when an executable file is loaded into memory

## Process State

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

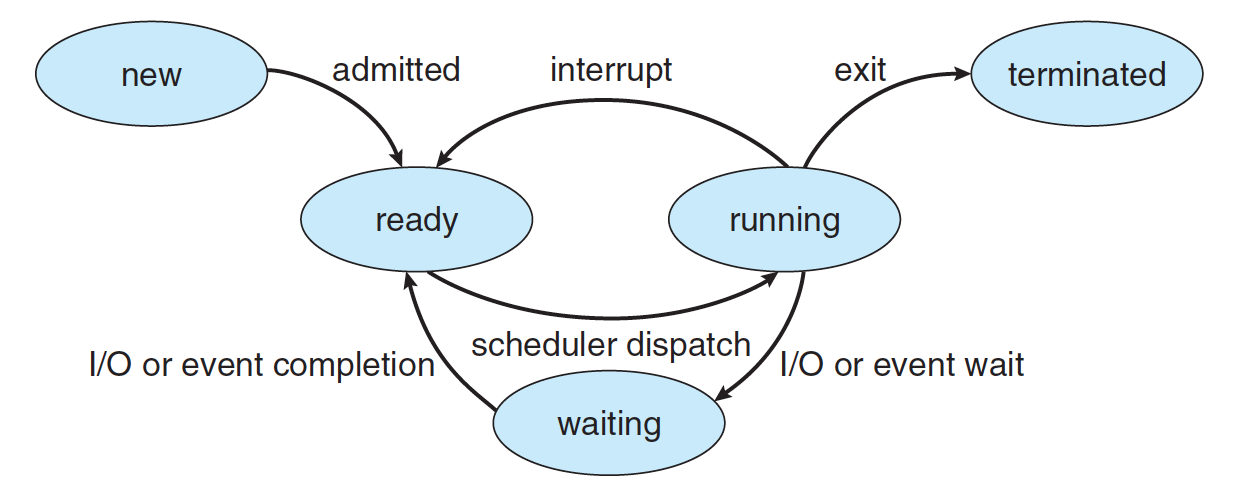


Figure 2. Diagram of process state

|  |  |
| --- | --- |
| **New** | The process is being created. |
| **Running** | Instructions are being executed. |
| **Waiting** | The process is waiting for some event to occur (such as an I/O completion or reception of signal). |
| **Ready** | The process is waiting to be assigned to a processor. |
| **Terminated** | The process has finished execution. |

## Process Control Block

Each process is represented in the operating system by a process control block (PCB).

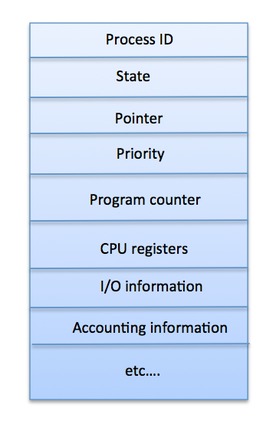


Figure 3. Process Control Block

|  |  |
| --- | --- |
| **Process State** | The current state of the process i.e., whether it is ready, running, waiting, or whatever |
| **Process privileges** | This is required to allow/disallow access to system resources. |
| **Process ID** | Unique identification for each of the process in the operating system. |
| **Pointer** | A pointer to parent process. |
| **Program Counter** | Program Counter is a pointer to the address of the next instruction to be executed for this process |
| **CPU registers** | Various CPU registers where process need to be stored for execution for running state. |
| **CPU Scheduling Information** | Process priority and other scheduling information which is required to schedule the process. |
| **Memory management information** | This includes the information of page table, memory limits, Segment table depending on memory used by the operating system |
| **Accounting information** | This includes the amount of CPU used for process execution, time limits, execution ID etc. |
| **IO status information** | This includes a list of I/O devices allocated to the process. |

## Process Scheduling

### What is Process Scheduling?

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

### Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

<dịch: HĐH duy trì tất cả PCB trong Hàng đợi lập lịch quy trình. HĐH duy trì một hàng đợi riêng cho từng trạng thái quy trình và PCB của tất cả các tiến trình trong cùng trạng thái thực thi được đặt trong cùng một hàng đợi. Khi trạng thái của một tiến trình được thay đổi, PCB của nó sẽ bị hủy liên kết khỏi hàng đợi hiện tại và được chuyển sang hàng đợi trạng thái mới>

* **Job Queue** – This queue keeps all the processes in the system.
* **Ready queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
* **Device queues** – The processes which are blocked due to unavailability of an I/O device constitute this queue.



Figure 4. Process Schedule Queue

The OS can use different policies to manage each queue:

* FIFO
* Round Robin
* Priority
* Etc.

### Context Swich

Context switch is the mechanism to store and restore the state or context of a CPU in Process Control Block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

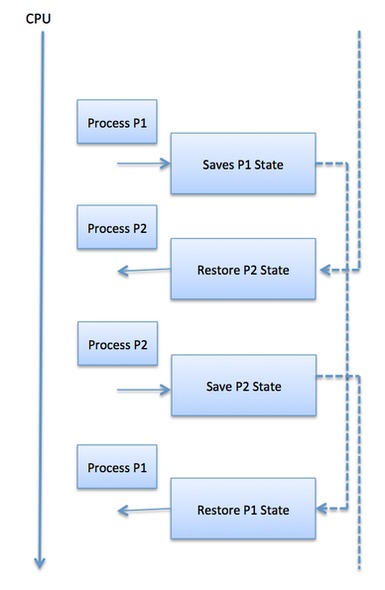


Figure 5. Illustration of Context Switch

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use:

* Program Counter
* Scheduling information
* Base and limit register value
* Currently used register
* Changed State
* I/O State information
* Accounting information

### OS Scheduling Algorithms

* First-Come, First-Serve
* Shortest Job First
* Priority
* Round Robin

# Thread

## What is Thread?

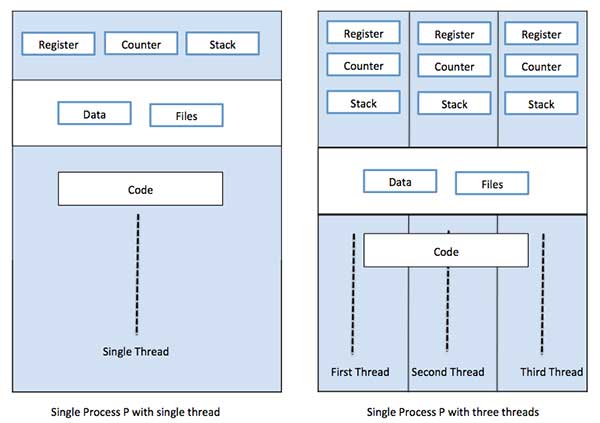


Figure 6. Single-thread vs Multi-thread process

# Goroutine

## Growable stacks

Start with small stack, typically 2KB, grows and shrinks as needed. Size limit for a goroutine stack may be as much as 1GB

## Goroutine Scheduling

OS thread are scheduled by the OS kernel. The operation is slow due to its poor locality and the number of memory accesses required.

Go runtime contains its own scheduler that uses a technique known as ***m:n scheduling***. Multiplexes (or schedules) m goroutines on n OS threads. The job of Go scheduler is analogous to that of the kernel scheduler, but it is concerned only with the goroutines of a single Go program.

## GOMAXPROCS

To determine how many OS threads may be actively executing Go code simultaneously.

## Goroutines have no identity

# Problems in concurrent programming

* Resource synchronization
* Activity synchronization
* Inter-thread communication

## Race conditions

A race condition is an undesirable situation that occurs when two or more threads can access shared data and they try to change it at the same time. The most common type of race condition is **check-then-act**.

# race condition example in Java: <https://www.netjstech.com/2015/06/race-condition-in-java-multi-threading.html>


## Deadlocks

A deadlock occurs when a thread enters a waiting state because a requested resource is held by another waiting thread, which in turn is waiting for another resource held by another waiting thread. If a thread is unable to change its state indefinitely because the resources requested by it are being used by another waiting thread, then the system is said to be in a deadlock.

# deadlock example in Java: <https://www.tutorialspoint.com/java/java_thread_deadlock.htm>





## Starvation

Starvation occurs when a thread is perpetually denied access to resources it needs in order to make progress; the most commonly starved resource is CPU cycles. Starvation happens when greedy threads make shared resources unavailable for long periods.

# starvation example in Java: <https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/thread-starvation.html>

## Livelock

Livelock is a form of liveness failure in which a thread while not blocked still cannot make progress because it keeps retrying an operation that will always fail. It often occurs in transactional messaging applications, where the messaging infrastructure rolls back a transaction if a message cannot be processed successfully, and puts it back at the head of the queue.

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

# livelock example in Java: <https://www.baeldung.com/java-deadlock-livelock>

# Solutions

## Atomic

An atomic operation is an operation that will always be executed without any other process being able to read or change state that is read or changed during the operation.

# Example in Java: <https://www.baeldung.com/java-atomic-variables>

import java.util.concurrent.atomic.AtomicInteger;  
  
class SafeCounter implements Runnable {  
// private final AtomicInteger c = new AtomicInteger(0);  
 private int c = 0;  
 public int getValue() {  
// return c.get();  
 return c;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 1000000; i++) {  
// c.addAndGet(1);  
 c++;  
 }  
 }  
}  
  
public class Atomic {  
 public static void main(String[] args)  
 throws InterruptedException {  
 SafeCounter counter = new SafeCounter();  
 Thread t1 = new Thread(counter, "T1");  
 Thread t2 = new Thread(counter, "T2");  
 t1.start();  
 t2.start();  
 t1.join();  
 t2.join();  
 System.*out*.println(counter.getValue());  
 }  
}

## Spinlock

Nếu critical resource chỉ là một số nguyên (hay một bit), và việc truy cập vào critical resource chỉ là thay đổi giá trị, thì kỹ thuật atomic là phù hợp. Nhưng trong nhiều trường hợp, critical resource là một cấu trúc dữ liệu phức tạp, và việc truy cập critical resource gồm nhiều thao tác, tức là critical section gồm nhiều câu lệnh. Khi đó, kỹ thuật atomic không phù hợp.

### Introduction

Spinlock là một cấu trúc dữ liệu, được dùng để ngăn chặn race condition xảy ra trên các cấu trúc dữ liệu khác. Nói nôm na, spinlock đảm bảo rằng: tại một thời điểm bất kì, chỉ có tối đa một thread được phép sử dụng critical resource.

### How it work?

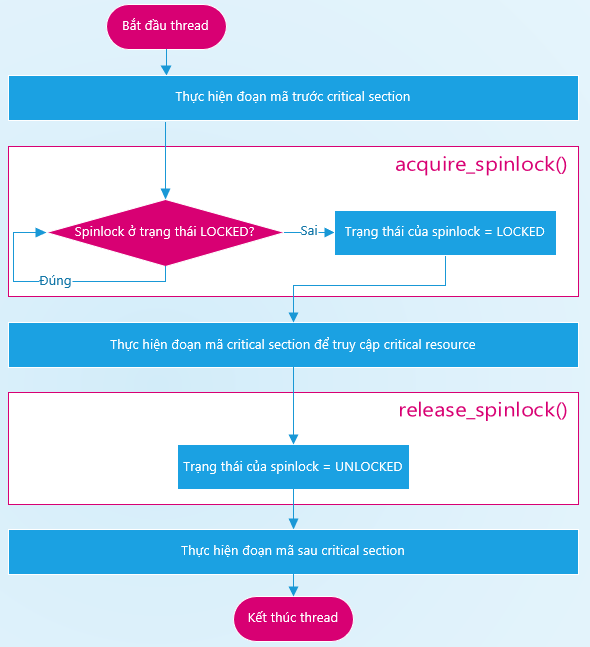


Figure 7. Hoạt động của spinlock

Khi spinlock đang ở trạng thái UNLOCKED, nếu một thread gọi hàm acquire\_spinlock, thì spinlock bị chuyển sang trạng thái LOCKED. Ta nói rằng, thread đã khóa spinlock lại (hay thread đã chiếm dụng spinlock). Sau đó, CPU bắt đầu thực thi critical section của thread (nói theo ngôn ngữ của CPU), hay thread đang sử dụng critical resource.

Khi spinlock đang ở trạng thái LOCKED, nếu một thread gọi hàm acquire\_spinlock, thì CPU liên tục thực thi hàm acquire\_spinlock cho tới khi spinlock chuyển sang trạng thái UNLOCKED. Tình trạng này được gọi là giao tranh (contended), còn cơ chế chờ đợi này được gọi là chờ bận (busy-waiting) => lãng phí thời gian của CPU nếu phải chờ đợi quá lâu. Do đó, spinlock thuộc loại khóa busy lock.

Khi spinlock đang ở trạng thái LOCKED, nếu một thread gọi hàm release\_spinlock, thì spinlock chuyển sang trạng thái UNLOCKED. Ta nói rằng, thread đã mở ổ khóa spinlock (hay thread đã giải phóng spinlock). Nếu vẫn còn một vài thread đang chờ đợi để chiếm dụng spinlock, thì một trong số các thread đó sẽ chiếm được spinlock và bắt đầu sử dụng critical resource. Khi ấy, spinlock lại trở về trạng thái LOCKED.

### How it protect critical resource?

#### Trường hợp 1: T1 muốn truy cập R trong khi T2 đang sử dụng R

Trước khi thực thi các lệnh trong critical section của T1, CPU0 sẽ thực thi hàm acquire\_spinlock và thấy rằng L đang ở trạng thái LOCKED. Khi đó, CPU0 sẽ liên tục thực thi hàm acquire\_spinlock để xem khi nào L trở về trạng thái UNLOCKED.

Sau khi thực thi xong critical section của T2, CPU1 thực thi tiếp hàm release\_spinlock để chuyển L sang trạng thái UNLOCKED. Lúc này, T1 sẽ chiếm lấy L và CPU0 tiếp tục thực thi T1.

#### Trường hợp 2: Cả T1 và T2 đồng thời muốn truy cập R

Khi đó, cả 2 thread đồng thời thực thi hàm acquire\_spinlock. Tuy nhiên, do hàm này dùng thao tác atomic để thay đổi spinlock, nên chỉ có một trong hai thread chiếm được L.

Thread nào chiếm được L trước thì sẽ sử dụng R trước. Thread nào không chiếm được L thì sẽ chờ bận cho đến khi thread đầu tiên sử dụng xong R.

## Mutex

Mutex is a data structure that enforces a **mut**ual **ex**clusion in a program. Only one process can access to shared resources.

Mutex is a lock mechanism.

## Semaphore

Semaphore is a signaling mechanism.

Semaphore use 2 atomic operations, wait() and signal()

# Example in Golang

## sync.Mutex

## atomic

References:

<https://medium.com/technofunnel/understanding-golang-and-goroutines-72ac3c9a014d>

<https://codeburst.io/why-goroutines-are-not-lightweight-threads-7c460c1f155f>

<https://medium.com/swlh/race-conditions-locks-semaphores-and-deadlocks-a4f783876529>

<https://www.tutorialspoint.com/operating_system/os_multi_threading.htm>

<https://vimentor.com/vi/lesson/giai-phap-mutex>

Code golang example for race condition

<https://medium.com/learning-the-go-programming-language/encoding-data-with-the-go-binary-package-42c7c0eb3e73> : BTL