**Question 1 (30 points)**

Use the **preemptive priority** scheduling algorithm to schedule the following set of processes (low numbers represent high priority):

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Arrival time | Burst time | Priority |
| P1 | 0 | 5 | 3 |
| P2 | 1 | 3 | 2 |
| P3 | 2 | 2 | 1 |
| P4 | 3 | 4 | 4 |

a. Draw the Gantt chart for the schedule. (15 points)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| P1 | P2 | P3 | | P2 | | P1 | | | | P4 | | | |

b. Calculate the average waiting time of these processes. (10 points)

**công thức: finish time – arrival time – burst time**

p1:

p2:

p3:

p4:

=> average waiting time =

c. Calculate the average turnaround time of these processes. (5 points)

**công thức: finish time – arrival time**

p1:

p2:

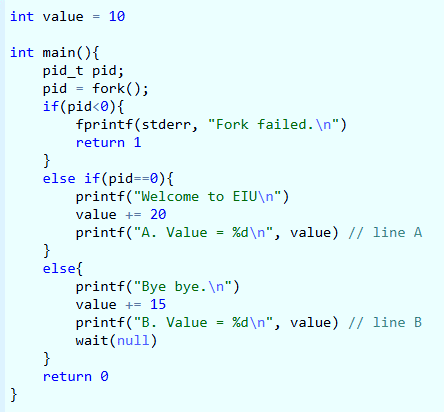
p3:

p4:

=> average turnaround time =

**Question 2 (20 points)**

a. Consider the following program. The parent process forked (created) a child process. What will be the values printed at line A and line B? Explain. **(10 points)**

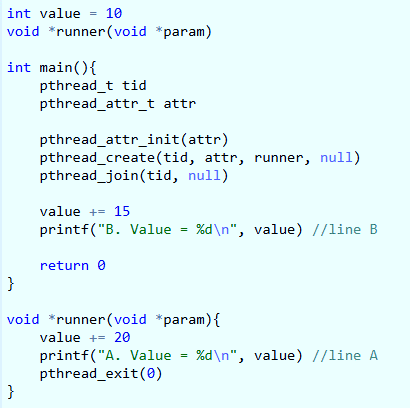


- Line A will print: A. Value = 30

- Line B will print: B. Value = 25

=> Because fork() creates 2 independent processes (parent-fork()>0 and child-fork()=0), the value in one process do not affect the other.

b. Consider the following program. In the program, a new thread is created by pthread\_create method. What will be the value printed at line A and line B? Explain. **(10 points)**



- Line A will print: A. Value = 30

- Line B will print: B. Value = 45

=> Because both thread share the same memory, the value update in child thread then the value update in main thread.

**Question 3 (20 points)**

Consider the following snapshot of a system:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Allocation** | **Max** |  | **Available** |
|  | **A B C D** | **A B C D** | **A B C D** |
| **T0** | 0 0 1 2 | 0 0 1 2 | 1 5 2 0 |
| **T1** | 1 0 0 0 | 1 7 5 0 |  |
| **T2** | 1 3 5 4 | 2 3 5 6 |
| **T3** | 0 6 3 2 | 0 6 5 2 |
| **T4** | 0 0 1 4 | 0 6 5 6 |

Answer the following questions using the banker's algorithm:

1. Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.

* Need matrix:
* Need(i)<=available -> new available = available + allocation(i)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Allocation** | **Max** | **Available** | **Need** |
|  | **A B C D** | **A B C D** | **A B C D** | **A B C D** |
| **T0** | 0 0 1 2 | 0 0 1 2 | ~~1 5 2 0~~ | 0 0 0 0 |
| **T1** | 1 0 0 0 | 1 7 5 0 | ~~1 5 3 2~~ | 0 7 5 0 |
| **T2** | 1 3 5 4 | 2 3 5 6 | ~~2 8 8 6~~ | 1 0 0 2 |
| **T3** | 0 6 3 2 | 0 6 5 2 | ~~2 14 11 8~~ | 0 0 2 0 |
| **T4** | 0 0 1 4 | 0 6 5 6 | ~~2 14 12 12~~ | 0 6 4 2 |
|  |  |  | 3 14 12 12 |  |

T0: 0 0 0 0 <= 1 5 2 0 -> true -> available = 1 5 2 0 + 0 0 1 2 = 1 5 3 2

T1: 0 7 5 0 <= 1 5 3 2 -> false

T2: 1 0 0 2 <= 1 5 3 2 -> true -> available = 1 5 3 2 + 1 3 5 4 = 2 8 8 6

T3: 0 0 2 0 <= 2 8 8 6 -> true -> available = 2 8 8 6 + 0 6 3 2 = 2 14 11 8

T4: 0 6 4 2 <= 2 14 11 8 -> true -> available = 2 14 11 8 + 0 0 1 4 = 2 14 12 12

T1: 0 7 5 0 <= 2 14 12 12 -> true -> available = 2 14 12 12 + 1 0 0 0 = 3 14 12 12

=> [ T0, T2, T3, T4, T1]

1. If a request from thread T₁ arrives for (0, 4, 2, 0), can the request be granted immediately? Explain.

T₁ arrives for (0, 4, 2, 0), the request can grante immediately. Explain:

- Request(i) <= need(i): 0 4 2 0 <= 0 7 5 0 -> true

- Request(i) <= available: 0 4 2 0 <= 1 5 2 0 -> true

- New available = available - request(i): 1 5 2 0 – 0 4 2 0 = 1 1 0 0

- New allocation(i) = allocation(i) + request(i) = 1 0 0 0 + 0 4 2 0 = 1 4 2 0

- New need(i) = need(i) - request(i): 0 7 5 0 – 0 4 2 0 = 0 3 3 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Allocation** | **Max** | **Available** | **Need** |
|  | **A B C D** | **A B C D** | **A B C D** | **A B C D** |
| **T0** | 0 0 1 2 | 0 0 1 2 | ~~1 1 0 0~~ | 0 0 0 0 |
| **T1** | 1 4 2 0 | 1 7 5 0 | ~~1 1 1 2~~ | 0 3 3 0 |
| **T2** | 1 3 5 4 | 2 3 5 6 | ~~2 4 6 6~~ | 1 0 0 2 |
| **T3** | 0 6 3 2 | 0 6 5 2 | ~~2 10 9 8~~ | 0 0 2 0 |
| **T4** | 0 0 1 4 | 0 6 5 6 | ~~2 10 10 12~~ | 0 6 4 2 |
|  |  |  | 3 14 12 12 |  |

T0: 0 0 0 0 <= 1 1 0 0 -> true -> available = 1 1 0 0 + 0 0 1 2 = 1 1 1 2

T1: 0 3 3 0 <= 1 1 1 2 -> false

T2: 1 0 0 2 <= 1 1 1 2 -> true -> available = 1 1 1 2 + 1 3 5 4 = 2 4 6 6

T3: 0 0 2 0 <= 2 4 6 6-> true -> available = 2 4 6 6+ 0 6 3 2 = 2 10 9 8

T4: 0 6 4 2 <= 2 10 9 8 -> true -> available = 2 10 9 8 + 0 0 1 4 = 2 10 10 12

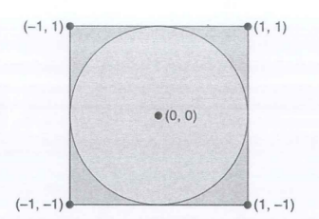
T1: 0 3 3 0 <= 2 10 10 12-> true -> available = 2 10 10 12 + 1 4 2 0 = 3 14 12 12

=> [ T0, T2, T3, T4, T1]

**Question 4 (30 points)**

An interesting way of calculating is to use a technique known as Monte Carlo, which involves randomization. This technique works as follows:

Suppose you have a circle inscribed within a square, as shown in the following figure. (Assume that the radius of this circle is 1)



1. First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle.

2. Next, estimate z by performing the following calculation:

Write a multithreaded version of this algorithm that the main thread creates several threads to generate a number of random points. The threads will count the number of points that occur within the circle and store that results in a shared variable. When these threads have exited, the parent thread will calculate and output the estimated value of . It is worth experimenting with the number of random points generated. As a general rule, the greater the number of points, the closer the approximation to .

**=> Bài 4 – lab 2**