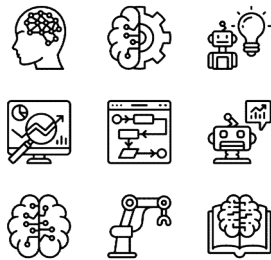


Computer Science for Practicing Engineers

Concurrency



TS. Huỳnh Bá Diệu
Email: dieuhb@gmail.com
Phone: 0914146868

Concurrency

Nội dung :

1. Introduction to concurrency
2. Models of concurrent systems and applications
3. Key challenges with concurrency
4. Strategies to manage concurrency

Concurrency

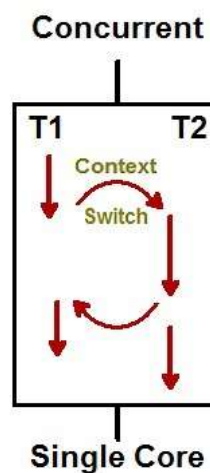
- *Concurrency* means that an application is making progress on more than one task at the same time (concurrently).
- If the computer only has one CPU the application may not make progress on more than one task at *exactly the same time*, but more than one task is being processed at a time inside the application.



Concurrency:
Multiple tasks makes progress at the same time.

Concurrency

....but more than one task is being processed at a time inside the application. It does not completely finish one task before it begins the next. Instead, the CPU switches between the different tasks until the tasks are complete.



Concurrency

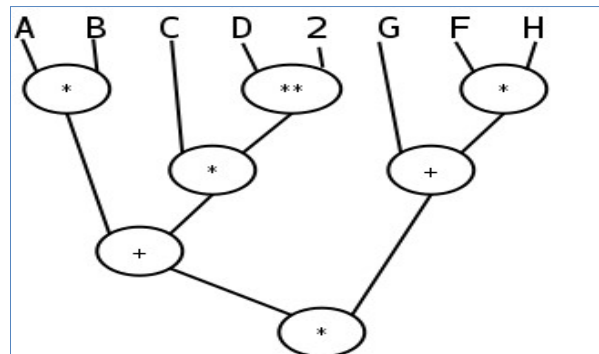
Các ví dụ về concurrency

- Railway Networks (note shared sections of track)
- Gardening (people rake, sweep, mow, plant, etc.)
- Machines in a factory
- Banking systems
- Travel reservation systems

Hoạt động hàng ngày của bạn có phải concurrency hay không???

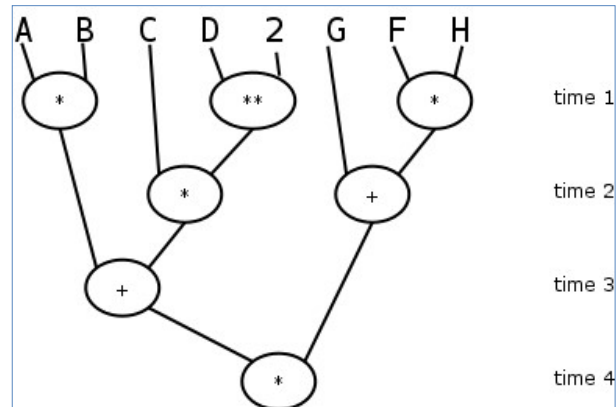
Concurrency

Có bao nhiêu cách tính cho biểu thức dưới đây???



Concurrency

The expression $(ab+cd^2)(g+fh)$ can be evaluated in four clocks



Concurrency

Many algorithms can be broken up into concurrent parts:

- Mergesort
- Quicksort
- Summing a list by summing fragments in parallel
- Operating on independent slices of an array

Concurrency vs Parallelism



Concurrent: 2 queues, 1 vending machine



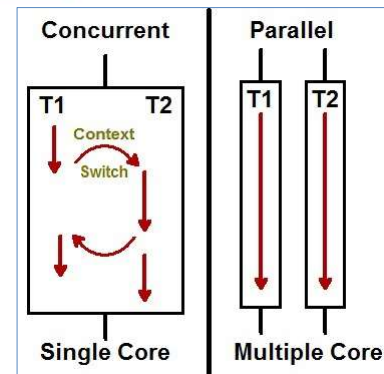
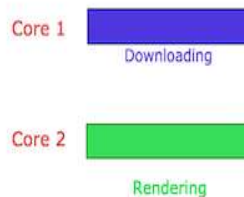
Parallel: 2 queues, 2 vending machines

Concurrency vs Parallelism

Concurrency



Parallelism



Concurrency vs Parallelism

- Concurrency is related to how an application handles multiple tasks it works on. An application may process one task at a time (sequentially) or work on multiple tasks at the same time (concurrently).
- Parallelism is related to how an application handles each individual task. An application may process the task serially from start to end, or split the task up into subtasks which can be completed in parallel.

Concurrency vs Parallelism

An application can be concurrent, but not parallel.

This means that it processes more than one task at the same time, but the thread is only executing on one task at a time. There is no parallel execution of tasks going in parallel threads / CPUs.

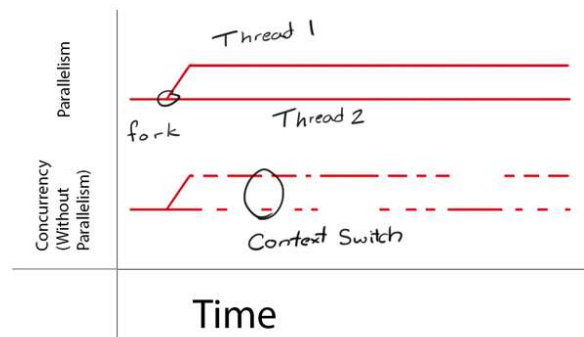
Concurrency vs Parallelism

An application can also be parallel but not concurrent.

This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel. However, each task (+ subtask) is completed before the next task is split up and executed in parallel.

Concurrency vs Parallelism

Additionally, an application can be neither concurrent nor parallel. This means that it works on only one task at a time, and the task is never broken down into subtasks for parallel execution.

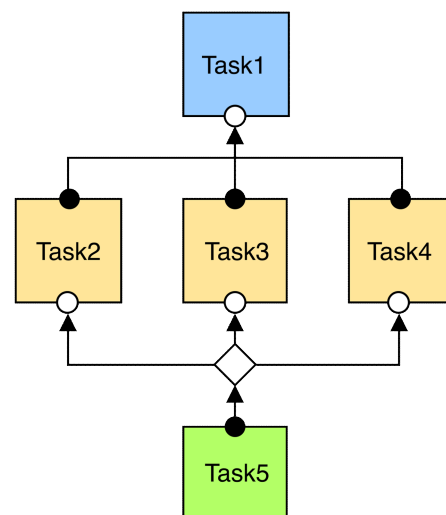


Concurrency vs Parallelism

Finally, an application can also be both concurrent and parallel, in that it both works on multiple tasks at the same time, and also breaks each task down into subtasks for parallel execution.

What is Concurrency

- Two or more tasks executing over the same time interval are said to execute concurrently
- Concurrency does not necessarily mean to imply that two tasks occur at exactly the same instant



What is Concurrency

The first task may execute for the first tenth of the second and pause, the second task may execute for the next tenth of the second and pause, the first task may start again, and so on

What is Concurrency (cont)

- Concurrent “tasks” can execute in a single or multiprocessing environment
- In a single processing environment, concurrent tasks exist at the same time and execute within the same time period by context switching

Tổng thời gian thực hiện xong các task có rút ngắn không???

What is Concurrency (cont)

- In a multiprocessor environment, if enough processors are free, concurrent tasks may execute at the same instant over the same time period
- The determining factor for what makes an acceptable time period for concurrency is relative to the application.

What is Concurrency (cont)

```

#include <iostream>
#include <thread>           ← 1

void hello()                ← 2
{
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello);    ← 3
    t.join();                ← 4
}

```

Why Concurrency?

- Simplifying the programming solution - sometimes it is cognitively easier to structure an application around a set of concurrent tasks
- Sometimes concurrency is used to make software faster or get done with its work sooner (same work, shorter period of time)

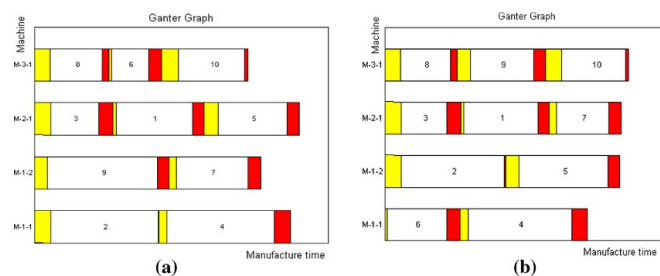
Why Concurrency?

- Sometimes concurrency is used to make software do more work over the same interval (more work, same period of time)
- Sometimes concurrency is used to increase capacity (more users, same work, same performance, same time period)

Challenges With Concurrency

Concurrency can offer enormous improvements in performance, capacity, throughput, and so forth, but can be challenging to get right...

- Resource Contention
- Mutual Exclusion Mechanisms
- Deadlock
- Scheduling



Resource Contention

Processors may have both read and write access to some resource

- Read access can be simultaneous
- Both concurrent and exclusive reads are permitted on the same piece of memory simultaneously
- Concurrent writes to memory are not trivial

Hai processes có thể ghi cùng một thời điểm không???

Resource Contention

Processors may have both read and write access to some resource

- Read access can be simultaneous
- Both concurrent and exclusive reads are permitted on the same piece of memory simultaneously
- Concurrent writes to memory are not trivial

Resource Contention

Processors may have both read and write access to some resource

-
- Concurrent writes to memory are not trivial
 - must ensure that no two processes write to the same memory at the same time
 - sophisticated algorithms allow multiple processors to write to memory

Mutual Exclusion (Mutex)

There are software and hardware solutions for mutual exclusion

- Hardware approaches on a single CPU system
- In multi CPU systems
- These tend to be low-level mechanisms,... we will focus on mutual exclusion in software.

Mutual Exclusion (Mutex)

There are software and hardware solutions for mutual exclusion

- Hardware approaches on a single CPU system
- mutual exclusion is commonly achieved by disabling interrupts for instructions that might corrupt shared data
- this prevents interrupt code from running in the critical section

Mutual Exclusion (Mutex)

Software and hardware solutions for mutual exclusion

- **In multi CPU systems**

- a busy flip-flop is used as a flag to indicate that a memory (or any resource) is busy
- the flag is tested in a tight loop to wait until the busy flag is cleared

Mutual Exclusion (Mutex)

Software and hardware solutions for mutual exclusion

- **In multi CPU systems**

- •

- a test-and-set operation performs both operations without releasing the memory bus to another processor
- when the code leaves the critical section, it clears the flag

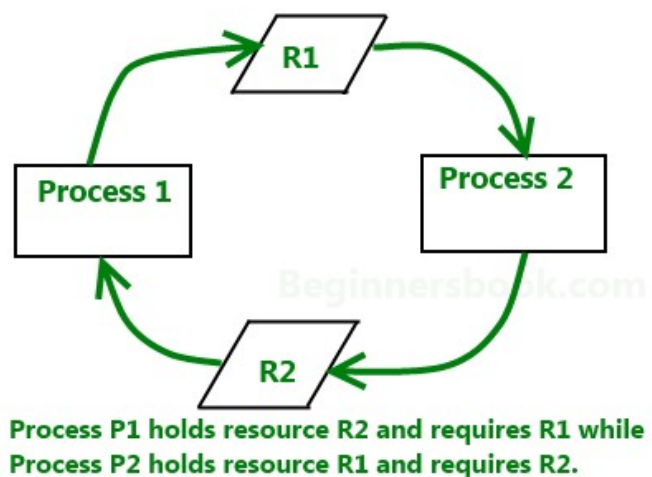
Mutual Exclusion (Mutex)

Software and hardware solutions for mutual exclusion

- These tend to be low-level mechanisms,... we will focus on mutual exclusion in software.

Deadlock and Livelock

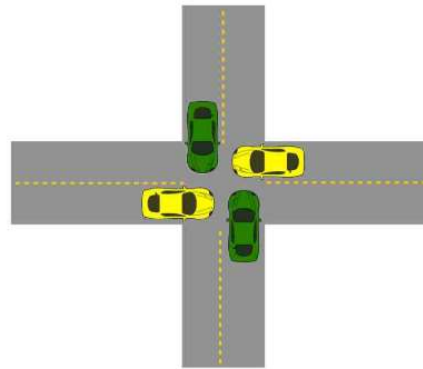
Deadlock refers to a specific condition when two or more processes are stopped, each waiting for another to release a resource.



Deadlock and Livelock

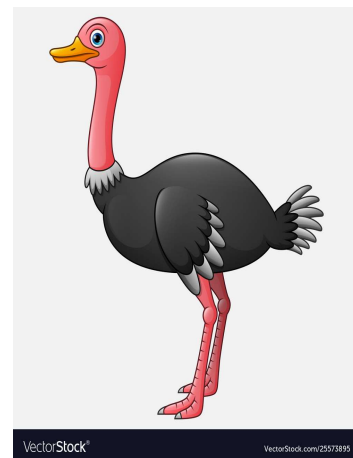
Livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another but no progress is made.

Livelock



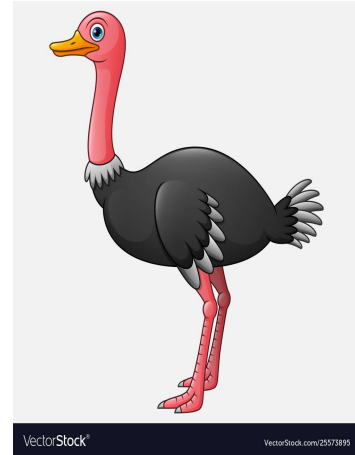
Ostrich Algorithm

- The *ostrich algorithm* is a strategy of ignoring potential problems on the basis that problems may be exceedingly rare
- Essentially you "stick your head in the sand" and ignore potential problems
-

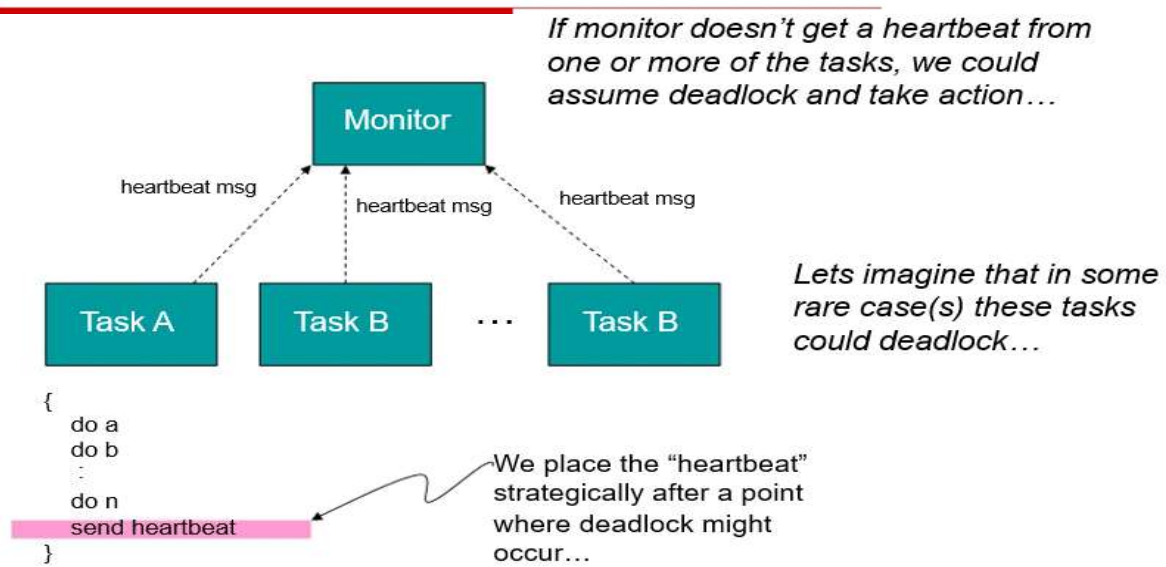


Ostrich Algorithm

-
- This approach may be used when deadlocks in concurrent programming can be shown to be “rare,” and if the cost of detection or prevention is high
- In using this approach, the designer must determine what “exceedingly rare” is and create mechanisms to detect these conditions, report and/or repair the condition



Example Ostrich Algorithm

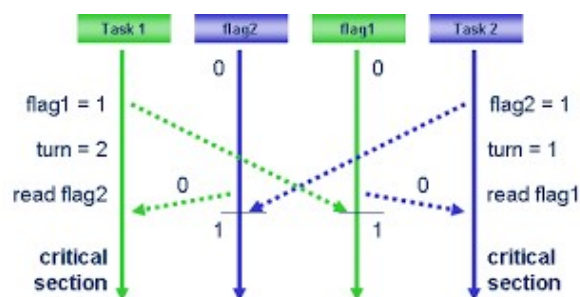


Dekker's Algorithm

- A concurrent programming algorithm for mutual exclusion that allows two threads to share a single-use resource without conflict, using shared memory
- If two separate processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is

Dekker's Algorithm

- If one process is already in the critical section, the other process will "*busy wait*" until the process exits
- Dekker's Algorithm is a fairness based algorithm



Example Dekker's Algorithm

```
f0 := false
f1 := false
turn := 0
```

Global
variables...

Process 0...

```
f0 := true
while f1 {
  if turn ≠ 0 {
    f0 := false
    while turn ≠ 0 { /* no op */ }
    f0 := true
  }
}
// begin critical section
:
// end critical section
turn := 1
f0 := false
```

Process 1...

```
f1 := true
while f0 {
  if turn ≠ 1 {
    f1 := false
    while turn ≠ 1 { /* no op */ }
    f1 := true
  }
}
// begin critical section
:
// end critical section
turn := 0
f1 := false
```

The Bakery Model - 1

Imagine a bakery with a numbering machine at the counter where each new customer gets a unique service number

- Service numbers increase by one for each customer
- A counter displays the number of the customer that is currently being served
- ...



The Bakery Model - 1

- ...
- Other customers wait until the baker finishes serving the current customer and the next number is displayed
- When done shopping, the customer loses their number and can then do whatever they want accept get service from the baker



The Bakery Model - 2

- The “customers” are threads that get some service number or “ticket”
- In some implementations it is possible that more than one thread gets the same number
 - *In these cases each thread also gets a priority #*
- ...

The Bakery Model - 2

- ...
- A lower ticket # means a higher priority. So threads with lower ticket numbers can access resources first
 - *If there are multiple threads with the same ticket #, threads with the same ticket# are granted access to resources in priority order*

Example - 1

NextServed = 0; CurrentTicket = -1; /* assign threads tickets as resources are requested */	
integer GetTicket() /* part of instantiation/declaration */ { if CurrentTicket > MaxTicket { CurrentTicket = 0; } else { CurrentTicket ++; } return CurrentTicket; } void EndService() { if NextServed == MaxTicket { NextServed = 0; } else { NextServed ++; } }	While NextServed != My_Ticket_Number { /* no op */ } // critical section : EndService(); // end critical section :

With Priorities

PriorityTable = array[0..MaxTicket] of Integers /* initialize with -1 */ NextServed = 0; CurrentTicket = -1; PriorityServed = 0; Ticket = {integer t, integer p}; /* where t = the ticket# and p = priority */	
ticket GetTicket() /* part of instantiation/declaration */ { ticket t; if CurrentTicket > MaxTicket { CurrentTicket = 0; } else { CurrentTicket ++; PriorityTable[CurrentTicket] ++; } t = {CurrentTicket, PriorityTable[CurrentTicket]}; return t; } void EndService(Ticket t) { if (t.p == PriorityTable[t.t]) { /* see if this is the lowest priority */ if (NextServed == MaxTicket) NextServed = 0; else NextServed ++; } else PriorityServed ++; } 	While NextServed != My_Ticket.t { /* no op */ } While PriorityServed != My_Ticket.p { /* no op */ } // critical section : EndService(My_Ticket); // end critical section :

The Banker's Algorithm

- The *banker's algorithm* is a resource allocation and deadlock avoidance algorithm that checks for potential deadlock
- The algorithm tests for deadlock conditions by simulating the allocation of pre-determined maximum resource allocations
- After allocating resources, it is possible to make a "safe-state" check to test for possible deadlock conditions for all other pending activities

The Banker's Problem - 1

Imagine there are four contractors with credit lines. Each draws down on the credit line as needed to do their work.

	Credit Used	Credit Line	Requested
Customer 1	0	10	0
Customer 2	0	20	0
Customer 3	0	10	0
Customer 4	0	25	0

Total Bank Cash: 30
Credit Used: 0
Total Bank Liability: 65

The bank's job is to keep enough cash in reserve so that they can meet their obligations and avoid a situation where customers cannot proceed because they cannot draw down their entire credit line.

The Banker's Problem - 2

	Credit Used	Credit Line	Requested
Customer 1	6	10	6
Customer 2	10	20	10
Customer 3	4	10	4
Customer 4	5	25	5

Total Bank Cash: 5
Credit Used: 25
Total Bank Liability: 65

	Credit Used	Credit Line	Requested
Customer 1	6	10	0
Customer 2	10	20	0
Customer 3	4	10	0
Customer 4	10	25	5

Total Bank Cash: 0
Credit Used: 30
Total Bank Liability: 65

The Banker's Problem - 3

	Credit Used	Credit Line	Requested
Customer 1	6	10	1
Customer 2	10	20	0
Customer 3	4	10	0
Customer 4	10	25	2

Total Bank Cash: 0
Credit Used: 30
Total Bank Liability: 65

The system is now in a deadlock situation. Customer 4 cannot continue until they get 2 units of cash. Nothing else can continue...

COMPLETE DEADLOCK!...

The Bankers Algorithm - 1

The Bankers Algorithm: "Give each requester all of the possible credit they could request."

	Credit Used	Credit Line	Requested
Customer 1	0	10	0
Customer 2	0	20	0
Customer 3	0	10	0
Customer 4	0	25	0

Total Bank Cash: 30
Credit Used: 0
Total Bank Liability: 65

This is not perfect, and results in implied pre-emption, prioritization, and scheduling.

The Bankers Algorithm - 2

	Credit Used	Credit Line	Requested
Customer 1	0	10	0
Customer 2	0	20	0
Customer 3	0	10	0
Customer 4	25	25	5

Total Bank Cash: 5
Credit Used: 25
Total Bank Liability: 65

In this example, no other resources are allocated until customer 4 finishes their work

	Credit Used	Credit Line	Requested
Customer 1	0	10	1
Customer 2	20	20	2
Customer 3	10	10	2
Customer 4	0	25	0

Total Bank Cash: 20
Credit Used: 10
Total Bank Liability: 65

The Bankers Algorithm - 3

	Credit Used	Credit Line	Requested
Customer 1	0	10	1
Customer 2	20	20	2
Customer 3	10	10	2
Customer 4	0	25	0

Total Bank Cash: 0
Credit Used: 30
Total Bank Liability: 65

No other resources are allocated until customer 2 or 3 finishes their work, and another customer can be found where the total credit line can be awarded.

The banker's algorithm is often implemented in operating systems and execute when a process requests resources.

This is more-or-less a fairness based approach...

Semaphores - 1

- In real life a semaphore is a system of signals used to communicate visually (flags, lights, or some similar mechanism)
- In software, a semaphore is a data structure used for synchronization.
- Semaphores are simple and very flexible



Semaphores - 1

- ...
- In software, a semaphore is a data structure used for synchronization. A semaphore is a data structure such that:
 - When you create a semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are *increment* and *decrement*.
 - The values of a semaphore should be hidden



Semaphores - 2

- In most programming environments, semaphores are part of the programming language or the operating system
This is because increment/decrement operations must be atomic and this can involve OS or even HW mechanisms
- If the value of a semaphore is positive, then it represents the number of threads that can execute and decrement without blocking

Semaphores - 2

- ...
- If it is negative, then it represents the number of threads that have blocked and are waiting
- If the value is zero, it means there are no threads waiting

Semaphores - 3

```

P(Semaphore s) // Acquire Resource
{
    wait until s > 0, then s := s-1; /* must be atomic */
}
V(Semaphore s) // Release Resource
{
    s := s+1; /* must be atomic */
}

```



P = decrement: computation blocks if the result is negative...

V = increment: continue computation if there are any waiting process...

Rendezvous

```

P (Semaphore s) // Acquire Resource
{
    wait until s > 0, then s := s-1; /* must be atomic */
}
V (Semaphore s) // Release Resource
{
    s := s+1; /* must be atomic */
}

```

Thread 1 using Semaphore S1 = 0:

statements	S1=0, S2=0
V(S1); /* signal */	S1=1, S2=0 (waiting)
P(S2); /* wait */	S1=1, S2=1
statements	



Thread 2 using Semaphore S2 = 0:

statements	S1=0, S2=0
V(S2); /* signal */	S1=1, S2=1
P(S1); /* wait */	
statements	



Mutex

```
wait (Semaphore s) // Acquire Resource
{
    wait until s > 0, then s := s-1; /* must be atomic */
}
signal (Semaphore s) // Release Resource
{
    s := s+1; /* must be atomic */
}
```

Semaphore mutex = 1

Thread 1

```
:
wait(mutex)      Mutex = 1 (no wait)
// critical section
count = count + 1
signal(mutex)     Mutex = 1
// end critical section
:
```

Thread 2

```
:
wait(mutex)      Mutex = 0 (wait)
// critical section
count = count + 1
signal(mutex)     Mutex = 1 (resume)
// end critical section
:
```

Multiplexing

```
wait (Semaphore s)
{
    wait until s > 0, then s := s-1; /* must be atomic */
}
signal (Semaphore s) // Release Resource
{
    s := s+1; /* must be atomic */
}
```

Semaphore mutex = MaxConcurrentThreads

Thread 1..N

```
:
wait(mutex)      Mutex is granted if value is greater than zero.
:               Mutex = Mutex - 1
// critical section
:
signal(mutex)     Mutex = Mutex + 1
// end critical section
:
```

Producer-Consumer

```

wait (Semaphore s) // Acquire Resource
{
    wait until s > 0, then s := s-1; /* must be atomic */
}
signal (Semaphore s) // Release Resource
{
    s := s+1; /* must be atomic */
}

```

Producer-Consumer

*Semaphore ready = 0
Semaphore mutex = 1*

Producer:

```

statements
:
wait (mutex);
    // store data in buffer
signal (ready);
signal (mutex);
:
statements

```

Consumer:

```

statements
:
wait (ready);
wait (mutex);
    // get data from buffer
signal (ready)
signal (mutex)
:
statements

```

Producer-Consumer

```

wait (Semaphore s) // Acquire Resource
{
    wait until s > 0, then s := s-1; /* must be atomic */
}
signal (Semaphore s) // Release Resource
{
    s := s+1; /* must be atomic */
}

```

Producer-Consumer

*Semaphore ready = 0
Semaphore mutex = 1*

Producer:

```

statements
:
wait (mutex);
    // store data in buffer
signal (ready);
signal (mutex);
:
statements

```

Consumer:

```

statements
:
wait (ready);
wait (mutex);
    // get data from buffer
signal (ready)
signal (mutex)
:
statements

```


Readers and Writers

```
wait (Semaphore s) // Acquire Resource
{
    wait until s > 0, then s := s-1; /* must be atomic */
}
signal (Semaphore s) // Release Resource
{
    s := s+1; /* must be atomic */
}
```

```
int readers = 0;
Semaphore mutex = 1;
Semaphore empty = 1;
```

Readers:

Readers and Writers

Writer:

```
wait (empty)
    // critical section for writers
signal (empty)
```

Readers:

```
wait (mutex)
    readers += 1
    if readers == 1
        wait (empty.wait) // first reader in locks
signal (mutex)

    // critical section for readers
```

```
wait (mutex)
    readers -= 1
    if readers == 0
        signal (empty) // last reader out unlocks
signal (mutex)
```

Starvation

In each of these examples, deadlock is prevented, but there is a danger of starvation

For example: If a writer arrives while there are readers in the critical section, it might wait in queue forever while readers come and go

Starvation

As long as a new reader arrives before the last of the current readers departs, there will always be at least one reader in the room

- This situation is not a deadlock, because some threads are making progress, but it *may* not be desirable
- This might work as long as the load on the system is low, but as load increases so will starvation

Scheduling - 1

- To prevent starvation and satisfy timing requirements, scheduling algorithms or *scheduling policies* are required in concurrent systems
- Semaphores can be used, but ultimately special processes call *schedulers* are required and work with mutex mechanisms to ensure:
 - timing
 - fairness
 - adherence to deadlines and so forth

Scheduling - 2

- Scheduling policy may be as simple as accepting each task that is submitted and executing it to completion
- The scheduling policy that is ultimately used is derived from the operational requirements and *correctness often depends upon timeliness*

Scheduling - 2

- More advanced algorithms may be based on:
 - process priorities
 - coordinating shared resources
 - communication protocols,... and many other factors...

Schedulers - 1

- A scheduler is responsible for activating processes or threads at specific intervals for specific periods of time
- Schedulers are often operating system processes
- Scheduling policy and locking mechanisms often work hand-in-hand and can be complex

Schedulers - 1

- Schedulers typically runs when a:
 - timer, interrupt, or event occurs
 - process/thread blocks on a request
 - new process/thread is created or is terminated

Schedulers - 2

- At the heart of a *scheduler* is the *scheduling algorithm* which is an implementation of the *scheduling policy*
- The scheduler examines the set of candidate processes/threads and chooses one to execute based on scheduling algorithm.

Schedulers - 2

- The operation of the scheduling policy will vary and depend upon such goals as:
 - *Maximizing CPU utilization*
 - *Maximizing throughput (#jobs/time)*
 - *Minimizing average turnaround time in terms of the average execution or processing time*
 - *Minimizing response time*

Schedulers - 3

- The goal of a scheduling policy is to prevent starvation and finish tasks in a “timely” way *depending upon what “timely” means in the context of the application domain*
- Scheduling policies can be based on:
 - First come first serve
 - Round robin
 - Short jobs first
 - Highest priority or important jobs first

Schedulers - 3

- Scheduling policies can be based on:
 - First come first serve
 - Round robin
 - Short jobs first
 - Highest priority or important jobs first



Will these all prevent or even preclude starvation???

Schedulers - 3

Will these all prevent or even preclude starvation???

- *it might even depend upon how you define starvation*

First-Come-First-Serve

- Based on a FIFO queue where jobs are scheduled for service in the order they arrive - common in batch processing systems
- When used, it tends to be non-preemptive – but this is not a rule.

Some systems utilize "job fences" that tells the scheduler that certain jobs must wait while other jobs execute

First-Come-First-Serve

- ...
- The current job gets all the system resources and runs until its done where "done" can mean end of CPU burst or completion of job
- All jobs treated equally and starvation is generally eliminated

Issues

Can lead to poor overlap of I/O and CPU utilization

- If each task runs until they are done or block for I/O they can get a "convoy effect"
- Task with long CPU burst executes that do not require devices, force other tasks that need devices to wait to complete (it is generally a good idea to keep resources as busy as possible).
- Small jobs wait behind long running

Can result in high average job turn-around rates

Issues

Can lead to poor overlap of I/O and CPU utilization

- If each task runs until they are done or block for I/O they can get a "convoy effect"
- Task with long CPU burst executes that do not require devices, force other tasks that need devices to wait to complete (it is generally a good idea to keep resources as busy as possible).
- Small jobs wait behind long running

Can result in high average job turn-around rates

Round-Robin

Round-robin schedulers where each process is given a specific time slice or a relative time quantum

- in a time slice policy, each process is allocated the exact same amount of time during each task execution cycle
- in time quantum policy, each process is allocated a stochastic value between some min- max amount of time during each task execution cycle

Issues

- A simple time slice policy can be too naïve – some tasks need and should receive more time than others.
- Choosing the time quantum can be tricky
- There is no way to express jobs of a higher priority through the system

Issues

-
- Choosing the time quantum can be tricky
 - If too small, then tasks spend all their time context switching and very little time making progress
 - If large, then it will be a while between the times a given task is scheduled to execute, leading to poor response time, throughput, and/or poor overall systemic performance
 - If too large, then the round-robin policy will morph into a first-come first-serve policy
-

Shortest-Job-First

- Similar to the express lane at the market - this policy prevents
 - "short running tasks" from waiting behind long jobs
 - long job queues
- Overtime it minimizes the average waiting time
- Can be implemented as a queue system where the tasks in the "short jobs queue" are done before tasks in the "long job queue."

Issues

- It can be difficult to know or estimate the running time of some tasks
- Historical data must be collected and running time estimated over time
 - dynamic allocation of jobs to queues results in better performance, but are also much more complex
- Poor long-job/short-job estimates may lead to starvation for long-running jobs

Most Important Job First

- A most important job first policy is a policy where “important” jobs are scheduled to run first
- Assign the highest priorities to important jobs and run the job with the highest priority next
- The policy may be implemented with multiple “priority queues” instead of single ready queue (Basically the policy is that we run all jobs on highest priority queue first)

Most Important Job First

- This policy may also incorporate pre-emption where:
 - higher priority jobs will cause processing on lower priority jobs to suspend
 - the higher priority job is given the CPU and it executes until it completes or is preempted by a higher priority task
 - the suspended job then resumes

Issues

- First and foremost: Once we have identified the tasks, it can be difficult to:
 - decide what is most important and what is of secondary importance and so on
 - determine the most optimal scheduling scheme
 - know how to assign priorities to tasks
- Improperly assigned priorities can cause starvation and deadlock

Scheduling Basics - 1



- In a priority based system, how do you :
 - assign priorities?
 - know if you can schedule all your tasks?
 - order the execution of your tasks so they can all meet their deadlines?

Task A

- *Most Important*
- Period: 250ms
- Duration: 150ms

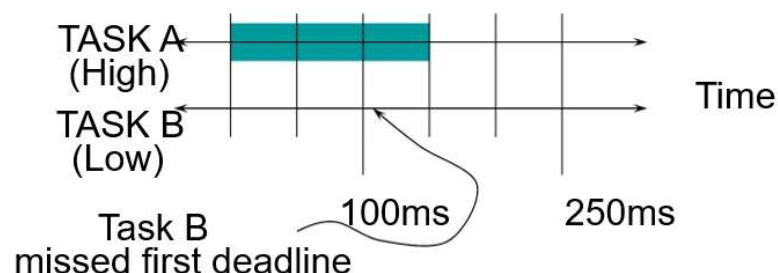
Task B

- *Least Important*
- Period: 100ms
- Duration: 50ms

Given these two tasks,
how would you assign
priority to them???

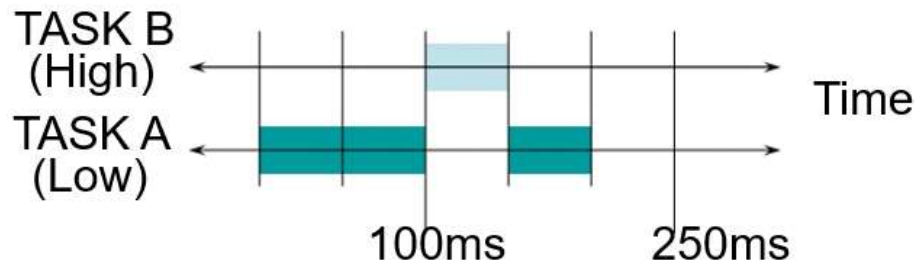
Scheduling Basics - 2

- You could schedule the most important task with the highest priority



Scheduling Basics - 3

- An alternative is to schedule the task with the highest frequency (shortest period) first



Rate Monotonic Scheduling

- Rate monotonic scheduling says:
 - assign the process/thread/task with the shortest period, the highest priority...
 - then assign the next highest priority with the next shortest process/thread/task with the nexthighest priority,... and so on
 - priorities are assigned based on period, not perceived importance

Rate Monotonic Scheduling

- Rate Monotonic Scheduling (RMS) can help:
 - assign priorities
 - ensure that each process/thread/task executes and finishes on time

Rate Monotonic Analysis

- Rate Monotonic Analysis (RMA) can be used to determine if a set of processes/threads/tasks can be scheduled (that is, all meet their deadlines)
- Simple RMA can be used for periodic tasks using a RMS policy

Rate Monotonic Analysis

- Simple RMA can be used for periodic tasks using a RMS policy, but this theory has been extended for:
 - Non-zero task switching times and variable context switching times
 - Pre-period deadlines
 - Interrupts and aperiodic processes
 - Distributed processing
 - Overload and Latency
 - Task Synchronization

Thảo luận nhóm

Cách viết một chương trình trên máy tính sử dụng concurrent programming!

Summary

- Concurrent programming is hard. Key issues include:
 - coordinating access to resources
 - ensuring mutual exclusion
 - coordinating operation between tasks
 - scheduling tasks
 - and a combination of these is almost always required in practice!

Tài liệu đọc thêm

Michel Raynal, "Algorithms for Mutual Exclusion"

Allen B. Downey, "The Little Book of Semaphores"

M. Ben-Ari, "Concurrent Programming"

Tài liệu đọc thêm

Michel Raynal, "Algorithms for Mutual Exclusion"

Allen B. Downey, "The Little Book of Semaphores"

M. Ben-Ari, "Concurrent Programming"

Tài liệu đọc thêm

https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/#reading_17_concurrency

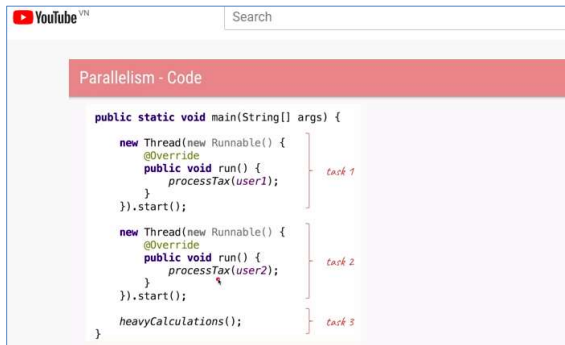
<https://cs.lmu.edu/~ray/notes/introconcurrency/>

<https://text.relipasoft.com/2016/12/dong-thoi-khong-phai-la-song-song-concurrency-is-not-parallelism/>

Link YouTube

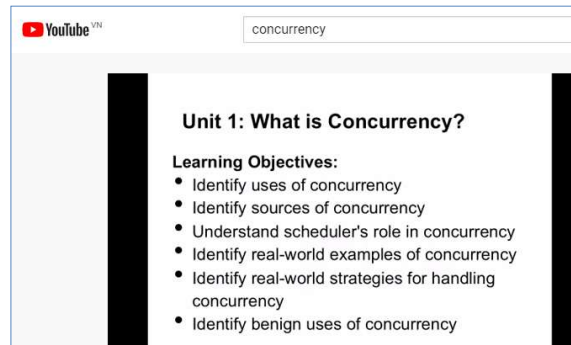
<https://www.youtube.com/watch?v=FChZP09Ba4E>

<https://www.youtube.com/watch?v=iKtvNJQoCNw>



The screenshot shows a YouTube video player with a search bar at the top. Below the search bar, there is a red header with the text "Parallelism - Code". The main content area displays a Java code snippet for a main method that creates three threads. The first two threads are annotated with "task 1" and "task 2" respectively, and the third thread is annotated with "task 3".

```
public static void main(String[] args) {  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            processTax(user1);  
        }  
    }).start();  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            processTax(user2);  
        }  
    }).start();  
    heavyCalculations();  
}
```



The screenshot shows a YouTube video player with a search bar at the top. Below the search bar, there is a slide titled "Unit 1: What is Concurrency?". The slide lists "Learning Objectives:" followed by five bullet points.

Unit 1: What is Concurrency?

Learning Objectives:

- Identify uses of concurrency
- Identify sources of concurrency
- Understand scheduler's role in concurrency
- Identify real-world examples of concurrency
- Identify real-world strategies for handling concurrency
- Identify benign uses of concurrency