Life in Deadlock IV
& Applications of Amdahl's Law

Lecture 14

Alma Rahat

CS-210: Concurrency

10 March 2021

Swansea University
Prifysgol Abertawe

What did we do in the last session?

- Deadlock handling: An App Developer's perspective.

In this lecture...

Learning outcomes.

1. To analyse resource allocation graphs and identify deadlocks during runtime.
2. To apply Amdahl's law in determining the potential speed up from program optimisation.

Outline.

1. Deadlock handling: an OS developer's perspective.
2. Resource allocation graphs.
3. Application of Amdahl's law.

Deadlock Handling: An Operating System Perspective
Dynamic or Run-time Schemes

Deadlock Avoidance

A dynamic (run-time) scheme.

- Do not start a process if its demands might lead to deadlock.
- Do not grant an incremental resource request to a process if this allocation may lead to deadlock.

Essentially, the following apply:

- Assume we know about the maximum resources required.
- Track allocations in real-time.
- When a request is made, only grant if guaranteed no deadlock even if all others take maximum resources.

Not very useful in practice, as we need to know a lot about the processes, and their needs a-priori.

Deadlock Detection and Recovery

Another dynamic scheme: probe programs regularly to construct a resource allocation graph, and see if there is at least one way to progress. If not, then we are deadlocked.

Recovery

Once detected, we have the following options:

- Abort all processes (common in many OSs).
- Back up to a predefined check point.
- Abort processes successively (i.e. one-by-one) until deadlock no longer exists.

Selection of processes for recovery may be based: processor time consumed, estimated time remaining, amount of output produced so far, etc.

Resource Allocation Graph (RAG)

P_i Process P_i

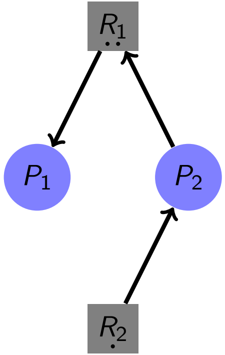
R_j Resource R_j with one instance (dot).

$P_k \longrightarrow R_k$ P_k *requesting* R_k (with two instances – two dots)

$P_l \longleftarrow R_l$ P_l *holding* R_l

Resource Allocation Graphs allow us to see the state of a system, and investigate whether deadlock is a possibility or not.

A process P_1 is holding R_1, and hopefully it would finish, and simultaneously P_2 can hold R_1 (two instances) and R_2 both and complete its tasks.

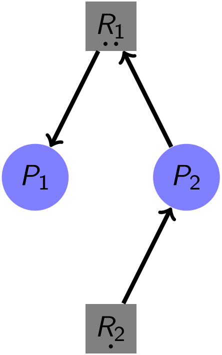


Resource Allocation Graph (RAG)

How to determine if deadlock exists?

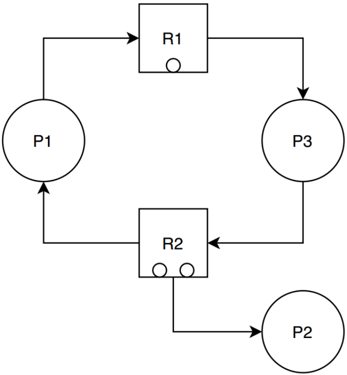
- No cycles $\equiv$ No deadlock.
- If there are cycles:
  - If only one instance per resource type, then deadlock.
  - If several instances per resource type, possibility of deadlock: further analysis required to establish if all processes can complete.

A process P_1 is holding R_1, and hopefully it would finish, and simultaneously P_2 can hold R_1 (two instances) and R_2 both and complete its tasks.



Example

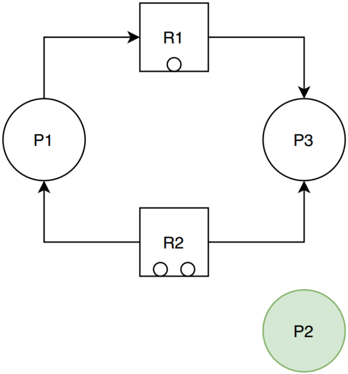
Consider the scenario: there are three processes $P1, P2$ and $P3$, and two resources $R1$ (with one instance) and $R2$ (with two instances). $P1$ has $R2$, but waiting on $R1$ which is held by $P3$. $P3$ is waiting on R_2, which is held by both P_1 and P_2.



Does deadlock occur here?

Example

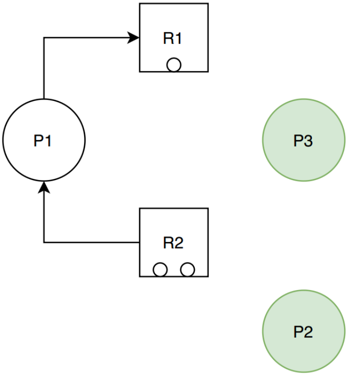
Consider the scenario: there are three processes $P1, P2$ and $P3$, and two resources $R1$ (with one instance) and $R2$ (with two instances). $P1$ has $R2$, but waiting on $R1$ which is held by $P3$. $P3$ is waiting on R_2, which is held by both P_1 and P_2.



$P2$ completes its tasks and lets $R2$ go. $P3$ therefore gets the resource, and now can finish its tasks.

Example

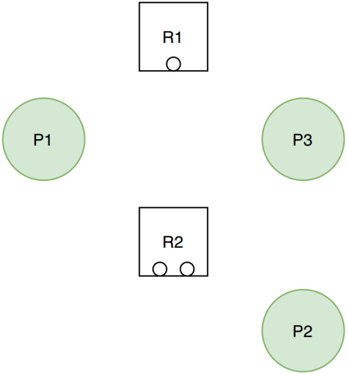
Consider the scenario: there are three processes $P1, P2$ and $P3$, and two resources $R1$ (with one instance) and $R2$ (with two instances). $P1$ has $R2$, but waiting on $R1$ which is held by $P3$. $P3$ is waiting on R_2, which is held by both P_1 and P_2.



$P3$ completes its tasks and lets $R1$ go. $P1$ therefore has all the resources it needs, and can complete its tasks.

Example

Consider the scenario: there are three processes $P1, P2$ and $P3$, and two resources $R1$ (with one instance) and $R2$ (with two instances). $P1$ has $R2$, but waiting on $R1$ which is held by $P3$. $P3$ is waiting on R_2, which is held by both P_1 and P_2.

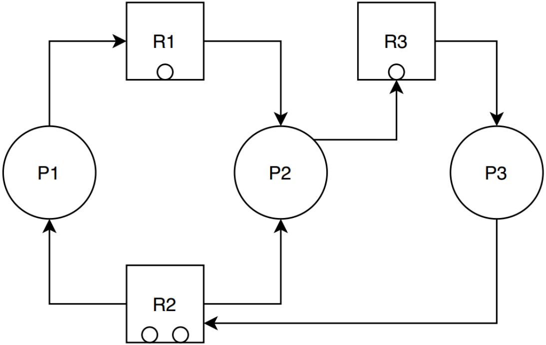


Each process finishes its tasks: no deadlock.

Any questions?

Exercise

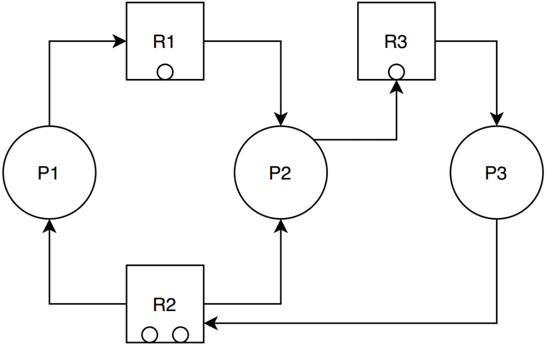
Is this system going to be deadlocked?



Please go to `www.menti.com` and use the code 7232 1306

Exercise

Is this system going to be deadlocked?



Please go to `www.menti.com` and use the code 7232 1306

$P3$ never gets $R2$, $P2$ never gets $R3$, and $P1$ never gets $R1$. The system is going to be deadlocked.

Any questions?

Program Optimisation

Program or software optimisation is a process of modifying a software system to make some aspect of it to work more efficiently (i.e. executes more rapidly) or to use fewer resources (e.g. memory, energy, etc.). There is usually a trade-off between efficiency and resource usages.

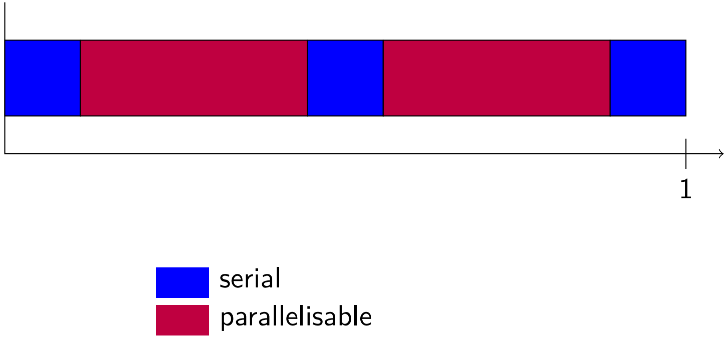
Levels of optimisation: Design, algorithms and data structures, source code, build, compile, assembly, run-time, and platform-based.

We should forget about small efficiencies, say about 97% of the time: premature optimization (prioritising performance over design) is the root of all evil. Yet we should not pass up our opportunities in that critical 3%

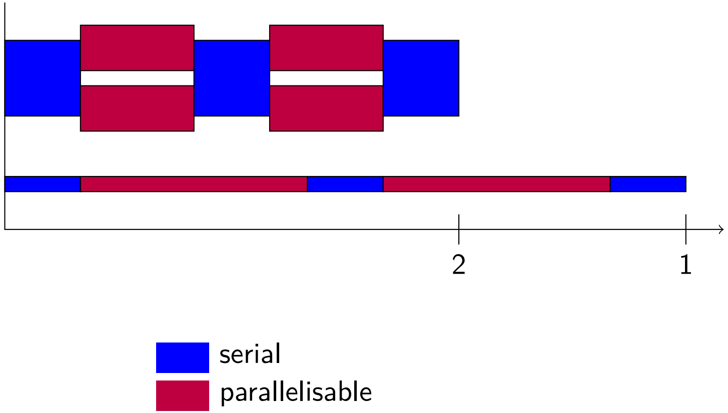
– Prof. Donald Knuth.

It takes effort and time to optimise a program. Often we can only spend finite time on a part of the program. How should we decide where to concentrate our efforts?

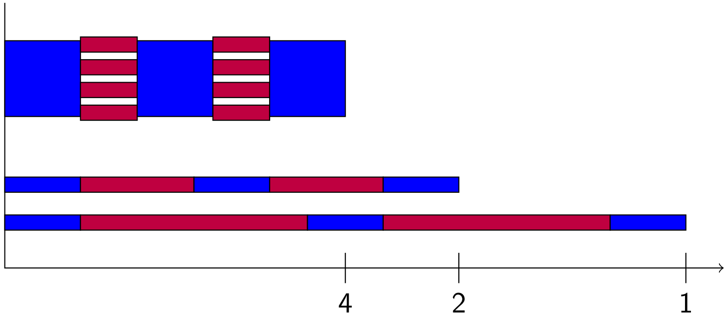
Execution time: single processor



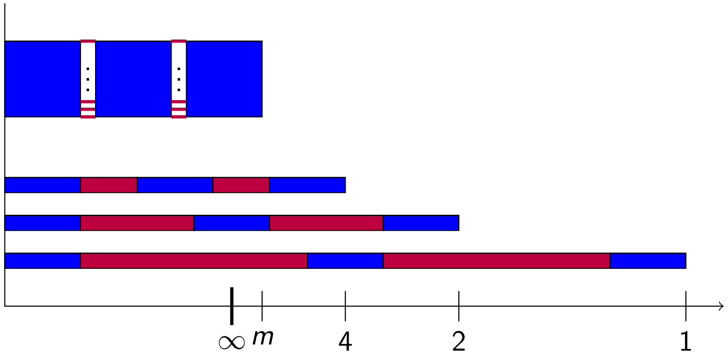
Execution time: two processors



Execution time: four processors



Execution time: many processors



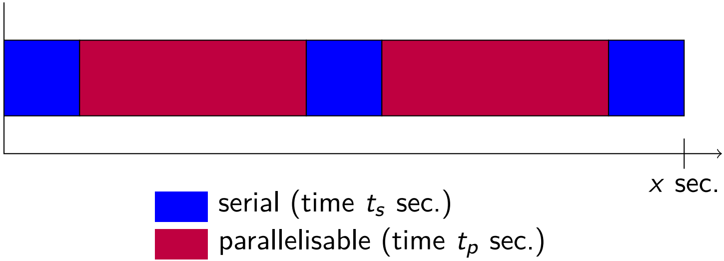
Amdahl's Law

Optimising code, either through parallelisation or code improvement, may lead to improving overall execution time of a program. There is, however, a limit to how much performance gain is achievable. Amdahl's law helps us compute such theoretical limits for tasks with **fixed workload**.

$$L(k, n) \leq \frac{t_i}{t_o(k, n)},$$

where, $L =$ upper bound of speed up in latency (proportional improvement in execution time),
$k =$ Improvement factor in the sequential part,
$n =$ Improvement factor in the parallelisable part,
$t_o =$ Optimised execution time,
$t_i =$ Execution time before improvement $t_i = t_o(1, 1)$.

Derivation



Initial total time for execution, t_i
$=$ time for serial part $+$ time for parallelisable part
$= t_s + t_p$

Therefore, $t_p = t_i - t_s$.

Derivation

If we use, n cores for the parallelisable part, then the execution time will be:

$$t'_p = \frac{t_p}{n} = \frac{t_i - t_s}{n}.$$

Now k times improvement in the sequential part means, the sequential execution time will be:

$$t'_s = \frac{t_s}{k}.$$

Therefore, the combined execution time is:

$$\begin{aligned} t_0(k, n) &= t'_s + t'_p \\ &= \frac{t_s}{k} + \frac{t_p}{n} \end{aligned}$$

Derivation

Now, the speed up bound is given by:

$$
\begin{aligned}
L(k,n) &\leq \frac{t_i}{t_o(k,n)} \\
&= \frac{t_s + t_p}{\frac{t_s}{k} + \frac{t_p}{n}} \\
&= \frac{1}{\frac{1}{k}\frac{t_s}{t_s+t_p} + \frac{1}{n}\frac{t_p}{t_s+t_p}} && \text{;divide both numerator} \\
& && \text{and denominator by } (t_s + t_p) \\
&= \frac{1}{\frac{1}{k}\tilde{t_s} + \frac{1}{n}\tilde{t_p}}
\end{aligned}
$$

Note that $\tilde{t_s} = \frac{t_s}{t_s+t_p}$ is the proportion of the time spent in executing the sequential part, and $\tilde{t_p} = \frac{t_p}{t_s+t_p}$ is the proportion of the time spent in executing the parallelisable part in the original program. Thus, $\tilde{t_p} = 1 - \tilde{t_s}$.

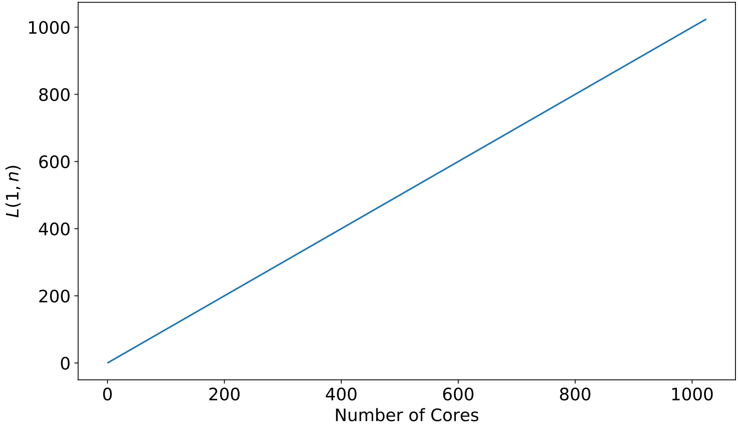
Any questions?

Important Observations

Given that: $L(k,n) \leq \frac{1}{\frac{1}{k}\tilde{t_s}+\frac{1}{n}\tilde{t_p}}$, what happens if $\tilde{t_s} = 0$?

Important Observations

Given that: $L(1, n) \leq \frac{n}{\tilde{t}_p}$, a linear increase in speed up.



Any questions?

Summary

- Resource allocation graphs can help OS programmers detect deadlocks.
- Amdahl's law helps us quickly determine the potential performance gain for our optimisation efforts, and this is defined as the proportion between original program execution time and the execution time of the optimised program.