



Operating Systems and Concurrency

Lecture 1: Introduction
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere, Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Goals

What Will be Covered?

- Introduction to the **fundamental concepts, key principles** and **internals** of (old and new) **operating systems** and **concurrency**
- Better understand how **application programs interact with the operating system**
- Basic understanding of writing **concurrent / parallel code** and **OS principles** related to concurrency

Lectures

When and Where?

- Through **lectures** on Mon, Wed, and Thu (4 weeks)
- Recordings will be available (will try to live stream over Echo360)
- Remember, studying is also about **interaction with peers** and **building contacts for life**

Lectures

Subjects We Will Discuss

Subject	#Lectures	By
Introduction to operating systems/computer design	3	GDM/DM
Processes, process scheduling, threading, ...	4	DM
Concurrency (deadlocks)	5	DM
Concurrency/coursework clarification/revision	1	GDM/DM
Memory management, swapping, virtual memory, ...	6	GDM
File Systems, file structures, management, ...	5	GDM
(Virtualisation & Cloud)	2	GDM
Revision	1	GDM

Table: Preliminary course structure

Labs

What and How?

- Labs are on Fridays (09:00 - 10:00) in A32, CS (from W/C 9th of October)
- The **labs** will teach you:
 - **OS concepts** (processes, schedulers, shared memory)
 - The use of operating system APIs & **implementation / coding** on Linux systems
 - The basics of **concurrency**
- **Lectures** will **introduce** these concepts

Coursework

Content

- The coursework focuses of a **OS structures, process scheduling, concurrency, and threads** (not **processes / fork**)
 - **Draft specification** will be available W/C 9th of October on Moodle (read this ASAP)
 - Follow the guidance to **break it down** in steps!
- It requires **C programming**

Submission

- The recommended submission date is the **12th of December** (latest date is **04/01/2024**)
- NO late submissions (unless you have ECs)!

Reading Material

My Favourite Books

- Seminal books:

- *Tanenbaum, Andrew S. 2014 Modern Operating Systems. 4th ed. Prentice Hall Press, Upper Saddle River, NJ, USA.*
- *Silberschatz, Abraham, Peter Baer Galvin, Greg Gagne. 2008. Operating System Concepts.* 8th ed. Wiley Publishing.
- *Stallings, William. 2008. Operating Systems: Internals and Design Principles.* 6th ed. Prentice Hall Press, Upper Saddle River, NJ, USA.

- Other sources:

- Daniel P. Bovet, Marco Cesati *Understanding the Linux Kernel.* 3rd ed. O'Reilly Media, November 2005
- Slides and recordings will be available on Moodle

Assessment

Exam & labs

- **In person ExamSys** (2 hours - TBC) that focuses on **knowledge, understanding, application**
 - The exam will have **3 out of 4 questions**, with 50% of the assessment on the exam
 - **Sample questions** from previous years are available on Moodle and are included in the lectures (**answers** are not available)
- Labs are part of the exam:
 - One or more (partial) questions in the **exam will be designed to evaluate the labs**
 - Help you with some aspects of the coursework (e.g. coding systems)

Assessment

- The coursework is an **individual** task and counts for 50%
 - **Git repositories** have been set up for you
 - **Only the final version** should be submitted in Moodle
 - Submit your code **regularly** (Git and Moodle- as many times as you like)
- **Academic misconduct** will be followed up on!

Assessment

Workload

- This is a **20 credit module - 200 hours** of work (5×40 hour week)
- The **coursework** should take approximately 100 hours
- **Lectures** take approximately 24 hours, **labs** 9 hours
- 67 hours of **revision**, approx 3 hours per **lecture**, 8 minutes per **slide**

Assessment

Start Revision Early

An E-mail Received Evening Before the Exam

Hi Geert,

*There is a **lot of information** covered during the course, hence **making revision very challenging**.*

*Do you have any guidance as to how to break the course down in to a **list of "main topics"** that are **essential to know in detail**?*

*Also, how will these **topics be split in to the 5 optional questions** in the exam?*

Thanks

...

Assessment

Start Revision Early

Response

Dear ... ,

*Unfortunately, I am **unable to provide any information** other than what was said during the lectures: the exam will try (as much as possible) to **assess all aspects covered in the module**. This is the only way in which a **fair exam** could be put together, since different students will find different topics easier/more difficult.*

*This is probably not the answer that you were hoping for, but if I would give **you a more detailed answer**, it **may be unfair to other students** who were not provided with this information.*

Best wishes,

Geert De Maere

About Us

Contact Details

- GDM's Contact details:
 - Name: Geert De Maere
 - E-mail: Geert.DeMaere@Nottingham.ac.uk
 - Office: C84
 - Office hour: Tuesdays 14:00 - 15:00 (confirm attendance by e-mail)
- DM's Contact details:
 - Name: Dan Marsden
 - Office: C41
 - E-mail: Dan.Marsden@Nottingham.ac.uk

About Me

Background

- Graduated in 2000, Bsc, Msc in Engineering
- Completed my PhD in CS in 2010 (Operational Research)
- Specific interest in **airline scheduling, airport operations** and **energy**
- I work together with **Institute for Aerospace Technology** (IAT), NATS, Heathrow Airport, etc.

About Me

My Background

- How does my research link in with operating systems?
 - I work on **scheduling** and **optimisation**
 - Exploit computer **architecture/design** and **common principles** in operating system design to:
 - Implement **sensible** parallelisations of algorithms
 - Speed up algorithms (caching, manipulate registers)
 - **Exploit similar principles** and **work on similar problems** in my daily work (e.g. caching, parallelisation, machine scheduling with sequence dependent setup times)
 - ...
- *“The ability to think independently while giving due weight to the arguments of others”*

Defining Operating Systems

What Can an OS Do For Me?

```
1 import java.io.FileWriter;
2 import java.io.IOException;
3 import java.io.PrintWriter;
4
5 public class Demo1 {
6     public static void main(String[] args) throws IOException {
7         FileWriter fw =
8             new FileWriter("C:/Program Files (x86)/test.txt");
9         PrintWriter pw = new PrintWriter(fw);
10        pw.close();
11    }
12 }
```

Defining Operating Systems

What Can an OS Do For Me?

- **File systems:** **where** is the file physically written on the disk and how is it **retrieved**?
- **Abstraction:** why looks the instruction the same **independent of the device**?
- **Concurrency:** what if multiple programs **access the same file simultaneously**?
- **Security:** why is the **access denied**?

Defining Operating Systems

What Can an OS Do For Me?

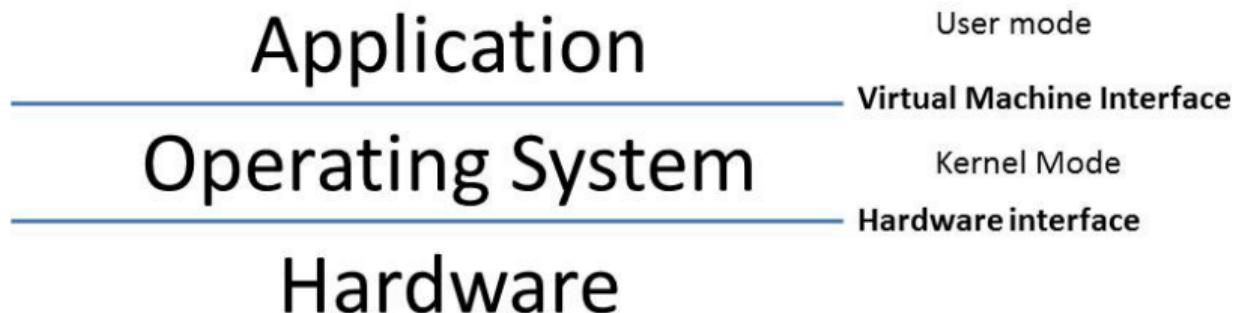
```
1 public class Demo2 {  
2     public static void main(String[] args) {  
3         long[] aLargeArrayOfLargeNumbers  
4             = new long[Integer.MAX_VALUE];  
5     }  
6 }
```

- **Where** in memory will the array be stored and how is it **protected** from unauthorised access?
- What if the array requires **more memory than physically available**?
- What if only **part of the array is currently in use** ?
- What if an **other process starts running**?

Defining Operating Systems

A Virtual Machine Providing Abstractions

- In the early days, programmers had to **deal directly with the hardware**
 - Real computer **hardware is ugly**
 - Hardware is **extremely difficult** to manipulate/program
- An operating system is a layer of indirection on top of the hardware:
 - It provides **abstractions** for application programs (e.g., file systems)
 - It provides a **cleaner and easier interface to the hardware** and hides the complexity of “**bare metal**”
 - It allows the programmer to be lazy by using **common routines** :-)



Defining Operating Systems

Some Wisdom

David Wheeler (First PhD in Computer Science, 1951)

“All problems in computer science can be solved by another level of indirection”

Defining Operating Systems

A Resource Manager

- Many modern operating systems use **multi-programming** to **improve user experience** and **maximise resource utilisation**
 - Disks are slow: without multi-programming, CPU time is wasted while waiting for I/O requests
 - Imagine a **CPU** running at 3.2 GHz (approx. 3.2×10^9 instructions per second)
 - Imagine a **disk** rotating at 7200 RPM, taking 4.2 ms to rotate half a track
 - I/O is slow, we are **missing out on** $3.2 \times 4.2 \times 10^6$ **instructions** (13.44m)!
 - The implementation of **multi-programming** has important **consequences** for **operating system design**

Defining Operating Systems

A Resource Manager

- The operating system must **allocate/share** resources (including CPU, memory, I/O devices) **fairly and safely** between **competing processes**:
 - In time, e.g. CPUs and printers
 - In space, e.g., memory and disks
- The execution of **multiple programs** (processes) needs to be **interleaved** with one another:
 - This requires **context switches** and **process scheduling** ⇒ **mutual exclusion, deadlock avoidance, protection, ...**

Summary

Take-Home Message

- Summary:
 - Structure of the **module & assessment**
 - Introduction to **operating systems**
 - Operating systems in terms of **abstractions** and **resource managers**
- Tasks:
 - Revise your knowledge of C

Operating Systems and Concurrency

Introduction 2
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

- ① The **code** we write heavily **uses operating system functionality**
- ② The **operating systems provides abstractions** (e.g., it hides hardware details) and **manages resources**
- ③ **Resources must be carefully managed** in a multi-programming environment

Goals for Today

Overview

- **Hardware** from an operating system's point of view
- **Address spaces**

Quick Lecturer Introduction

Name: Dan Marsden

Email: dan.marsden@nottingham.ac.uk

Interests: Theoretician interested in category theory, logic and the foundations of computer science, working in the functional programming lab

Quick Lecturer Introduction

Name: Dan Marsden

Email: dan.marsden@nottingham.ac.uk

Interests: Theoretician interested in category theory, logic and the foundations of computer science, working in the functional programming lab

Previous life: Approximately 12 years working as a professional developer in industry. Co-author of two libraries in the Boost C++ library project www.boost.org

Computer Hardware

Computer hardware in a nutshell

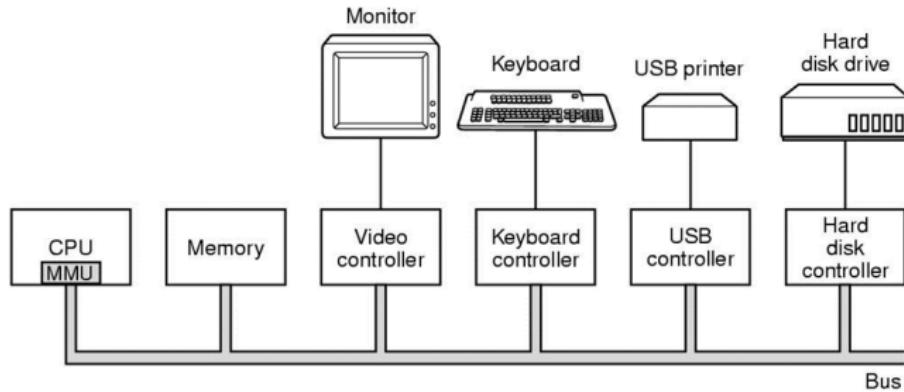


Figure: Simplified computer model (Tanenbaum, 2014)

Computer Hardware

CPU Design: Instruction evaluation

- Naively, CPU's execute a sequence of instructions one at a time
- A CPU's basic cycle consist of **fetch**, **decode**, and **execute** running in a **pipeline**
- A **superscalar** CPU provides **instruction level parallelism**, evaluating multiple instructions in parallel

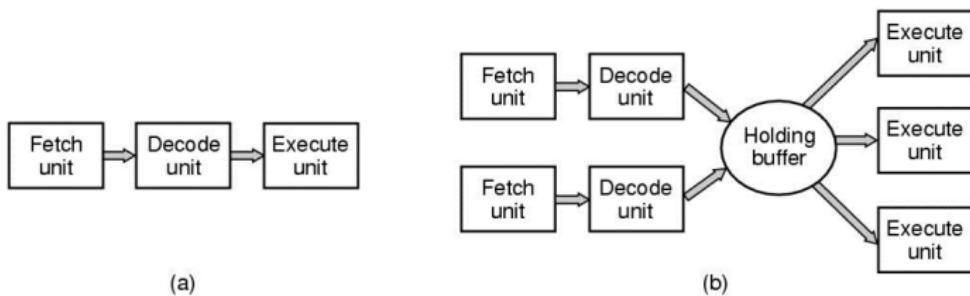


Figure: Processor Architectures (Tanenbaum, 2014)¹

¹ Computerphile info: https://www.youtube.com/watch?v=_qv0L8nhN4
©University of Nottingham

Computer Hardware

CPU Design: Advanced instruction evaluation

The search for further speed

- **Out-of-order execution** - instructions may not be evaluated in the order they appear in your code
- **Speculative evaluation** - speculative evaluation of instructions
- ...

Computer Hardware

CPU Design: Advanced instruction evaluation

The search for further speed

- **Out-of-order execution** - instructions may not be evaluated in the order they appear in your code
- **Speculative evaluation** - speculative evaluation of instructions
- ...

Take home message

You have to be very careful about the assumptions you make about CPU behaviour!^a

^aCompiler optimizations and memory architecture further complicate the picture

Computer Hardware

CPU Design: Registers

- Registers are small fast element of memory close to the CPU
- Registers have multiple purposes:
 - program counter (PC)** - which instruction should be run next
 - program status word (PSW)** - flags configuring the state of the CPU
 - general purpose registers** - storing operands for CPU instructions
- The **compiler/programmer** decides what to keep in the registers
- Registers** are part of the state of a running program

Computer Hardware

CPU Design: Moore's "law"

Moore's law

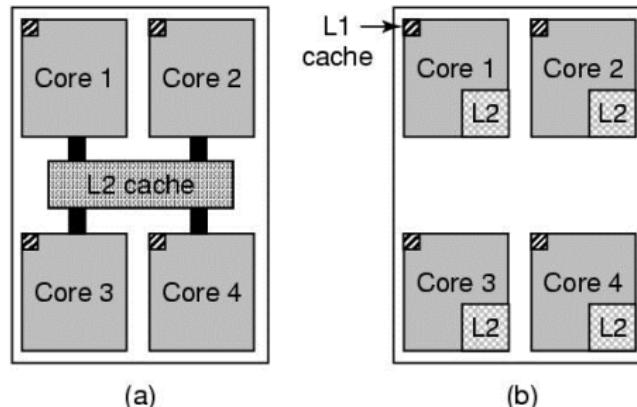
"The number of transistors on an integrated circuit (chip) doubles roughly every two years"

- Closely linked, but **not necessarily related to performance**
(performance roughly doubles until 2003)
- Moore's still continuing, but the "**power wall**" slows performance improvements of single core/single processor systems
 - A few cores for multiple "programs" is easy to justify
 - How to use **massively parallel** computers/CPUs/many core machines?

Computer Hardware

CPU Design: Multiple cores and threads

- Modern CPUs contain **multiple cores** and are often support multiple hardware threads within a core
- **Evolution in hardware** has implications on **operating system design**
 - Older operations systems such as Windows XP did not support multi processor architectures
 - Modern operating systems, along with other infrastructure such as compilers, **need to make good use of hardware concurrency** to fully exploit hardware advances



Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
    while(iVar++ < 10) {
        printf("Addr:%p; Val:%d\n", &iVar, iVar);
        sleep(1);
    }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
    while(iVar++ < 10) {
        printf("Addr:%p; Val:%d\n", &iVar, iVar);
        sleep(1);
    }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
    while(iVar++ < 10) {
        printf("Addr:%p; Val:%d\n", &iVar, iVar);
        sleep(1);
    }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
    while(iVar++ < 10) {
        printf("Addr:%p; Val:%d\n", &iVar, iVar);
        sleep(1);
    }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
    while(iVar++ < 10) {
        printf("Addr:%p; Val:%d\n", &iVar, iVar);
        sleep(1);
    }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
    while(iVar++ < 10) {
        printf("Addr:%p; Val:%d\n", &iVar, iVar);
        sleep(1);
    }
}
```

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
    while(iVar++ < 10) {
        printf("Addr:%p; Val:%d\n", &iVar, iVar);
        sleep(1);
    }
}
```

- If we run this multiple times, will the **same or different addresses** be displayed for `&iVar`?

Computer Hardware

CPU Design: Memory Management Unit

```
#include <stdio.h>
#include <unistd.h>

int iVar = 0;

int main() {
    while(iVar++ < 10) {
        printf("Addr:%p; Val:%d\n", &iVar, iVar);
        sleep(1);
    }
}
```

- If we run this multiple times, will the **same or different addresses** be displayed for `&iVar`?
- If we run two copies simultaneously, will the **same or different addresses** be displayed for `&iVar`?

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

Addr: 0x10eac2000; Val: 1 ...

Instance two output

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

Addr:0x10eac2000; Val:1 ...
Addr:0x10eac2000; Val:2 ...

Instance two output

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

Addr:0x10eac2000; Val:1 ...

Addr:0x10eac2000; Val:2 ...

Addr:0x10eac2000; Val:3 Addr:0x10eac2000; Val:1

Instance two output

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

Addr:0x10eac2000; Val:1 ...
Addr:0x10eac2000; Val:2 ...
Addr:0x10eac2000; Val:3 ...
Addr:0x10eac2000; Val:4 ...

Instance two output

Addr:0x10eac2000; Val:1 ...
Addr:0x10eac2000; Val:2 ...

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

Addr:0x10eac2000; Val:1 ...
Addr:0x10eac2000; Val:2 ...
Addr:0x10eac2000; Val:3 ...
Addr:0x10eac2000; Val:4 ...
...

Instance two output

Addr:0x10eac2000; Val:1
Addr:0x10eac2000; Val:2
...

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

Addr:0x10eac2000;	Val:1	...
Addr:0x10eac2000;	Val:2	...
Addr:0x10eac2000;	Val:3	Addr:0x10eac2000; Val:1
Addr:0x10eac2000;	Val:4	Addr:0x10eac2000; Val:2
...		...

Instance two output

...	
...	

- Both processes display 0x10eac2000, the same address.
- The value of `iVar` is different in each run of the program.

Computer Hardware

CPU Design: Memory Management Unit

Instance one output

Addr:0x10eac2000; Val:1 ...
Addr:0x10eac2000; Val:2 ...
Addr:0x10eac2000; Val:3 ...
Addr:0x10eac2000; Val:4 ...
...

Instance two output

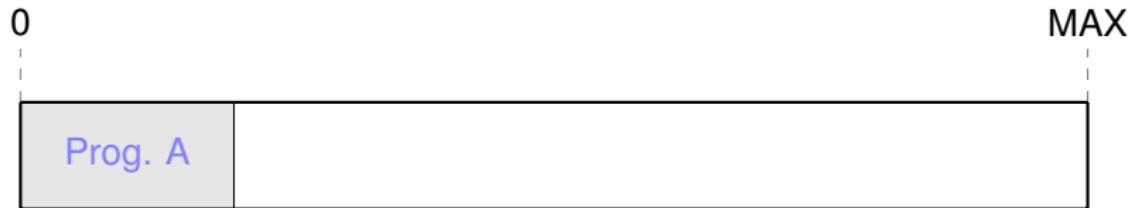
Addr:0x10eac2000; Val:1 ...
Addr:0x10eac2000; Val:2 ...
...

- Both processes display 0x10eac2000, the same address.
- The value of iVar is different in each run of the program.

Different behaviour on Mac due to **Address Space Layout Randomization (ASLR)**.

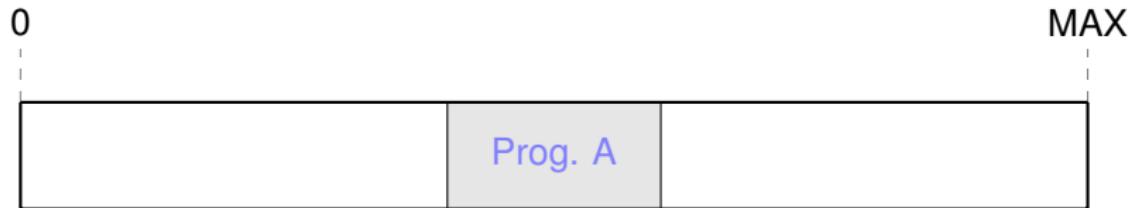
Computer Hardware

CPU Design: Memory Management Unit



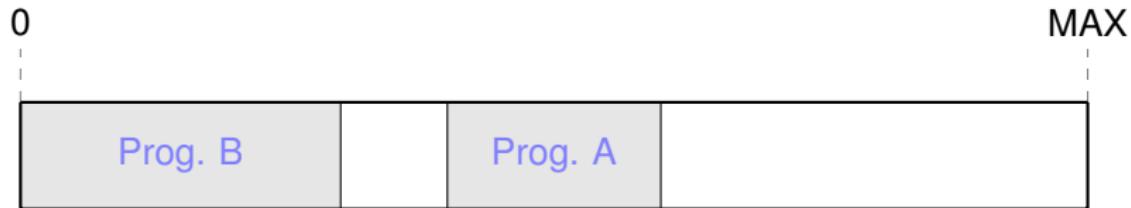
Computer Hardware

CPU Design: Memory Management Unit



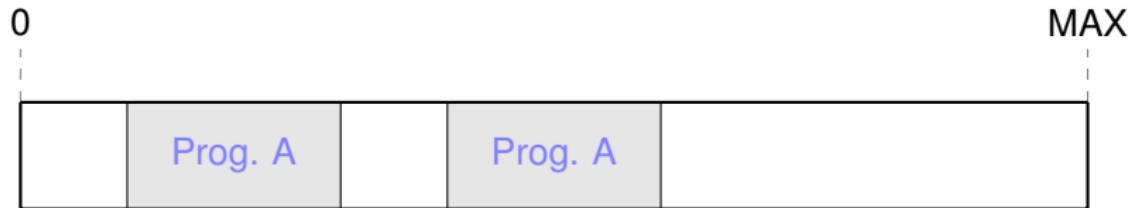
Computer Hardware

CPU Design: Memory Management Unit



Computer Hardware

CPU Design: Memory Management Unit



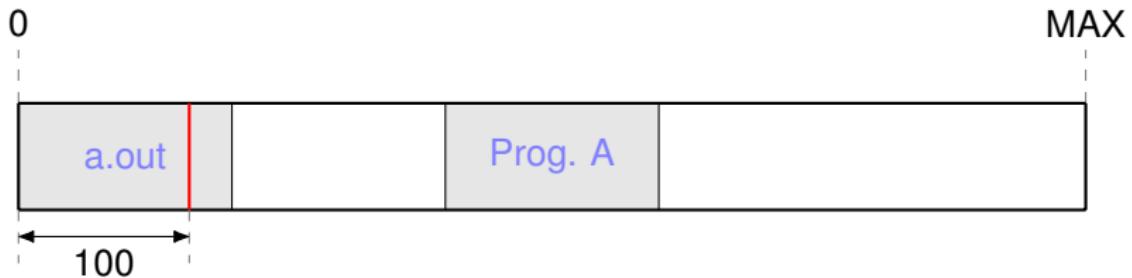
Computer Hardware

Memory Management Unit

- One cannot know **where in memory an executable will run**
 - Might be influenced by what other code is running on the machine
 - Multiple instances of the same program require distinct portions of memory
- When coding: variables have memory associated with them - they **need to have an address**
 - But the exact **physical addresses** cannot be known at compile time
 - The compiler just **assumes** that the program will **start at address 0**
 - If the program does not run at physical address 0 but at, e.g. 1000, just **add the offset** (1000) to every “compiled address”

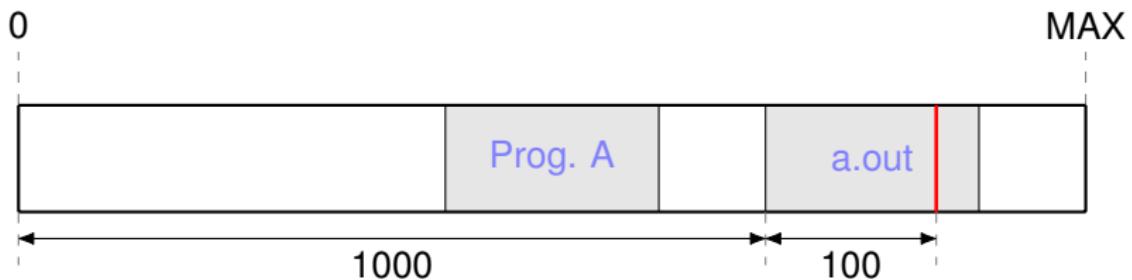
Computer Hardware

CPU Design: Memory Management Unit



Computer Hardware

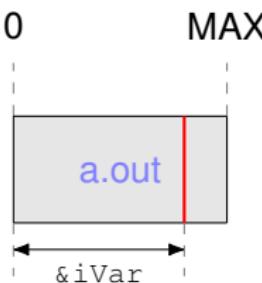
CPU Design: Memory Management Unit



Computer Hardware

Memory Management Unit

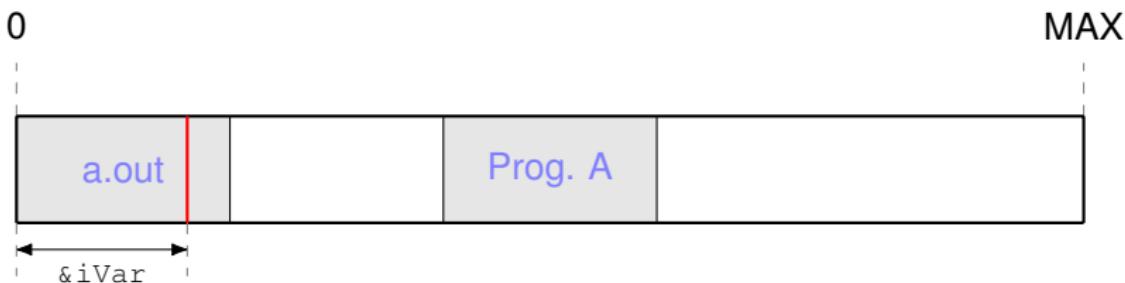
- There are two different address spaces:
 - the **logical address space** seen by the program and used by the compiler
 - the **physical address space** seen by the hardware/OS
- When compiling code, **memory addresses must be assigned to variables** and instructions, the compiler does not know what **memory addresses will be available** in physical memory
- It will just assume that the code will **start running at address 0** when generating the machine code



Computer Hardware

Memory Management Unit

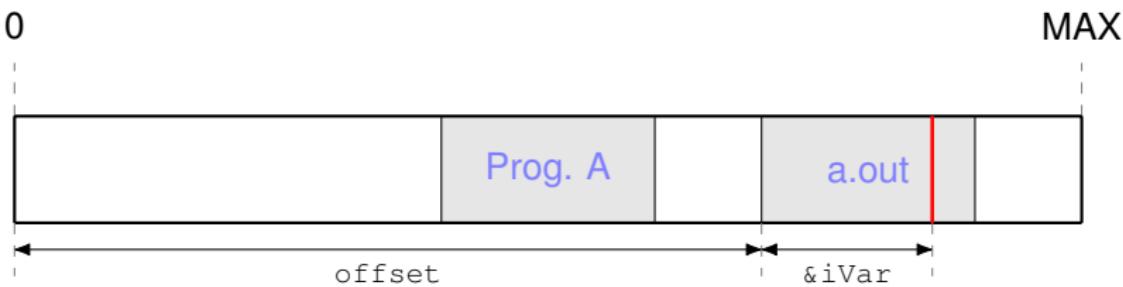
- There are two different address spaces:
 - the **logical address space** seen by the program and used by the compiler
 - the **physical address space** seen by the hardware/OS
- When compiling code, **memory addresses must be assigned to variables** and instructions, the compiler does not know what **memory addresses will be available** in physical memory
- It will just assume that the code will **start running at address 0** when generating the machine code



Computer Hardware

Memory Management Unit

- There are two different address spaces:
 - the **logical address space** seen by the program and used by the compiler
 - the **physical address space** seen by the hardware/OS
- When compiling code, **memory addresses must be assigned to variables** and instructions, the compiler does not know what **memory addresses will be available** in physical memory
- It will just assume that the code will **start running at address 0** when generating the machine code



Computer Hardware

Memory Management Unit

- On some rare occasions, the program **may run at physical address 0**
 - physical address = logical address + 0
- On other occasions, it will be running at a **completely different location** in physical memory and an offset is added
 - physical address = logical address + offset
- The **memory management unit (MMU)** is **responsible for address translation** (“adding the offset”)
 - Different program instances require **different address translation**
 - The state of the MMU is part of the state of a running program.

Computer Hardware

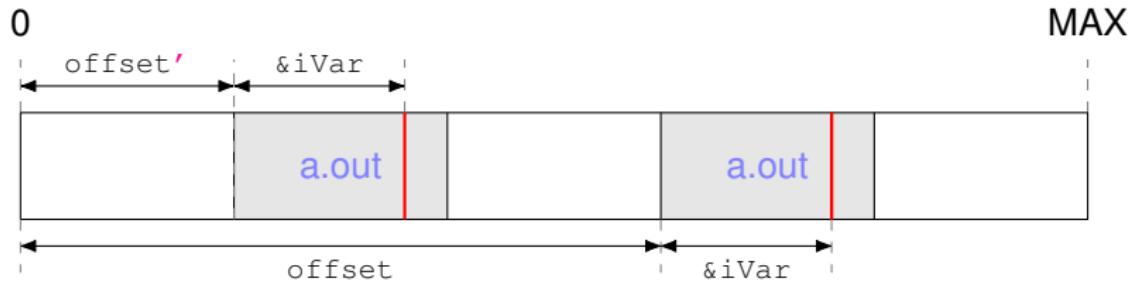
CPU Design: Memory Management Unit

- In the case of our example:

- The **address printed** on the screen is the **logical address**
- The logical address is translated into **two different physical addresses** using **different offsets**
- The code

```
printf("Addr:%p; Val:%d\n", &iVar, iVar);
```

was printing the logical address of iVar.



Computer Hardware

Memory Management Unit

- Our discussion has omitted many details, address translation may be more complex than applying a single offset uniformly. More detail in memory management lectures later.
- The key point is that the data required by the MMU to translate addresses is part of the state of a running program

Recap

Take-Home Message

- Operating Systems design is heavily coupled to hardware design and capabilities
- Registers contents and data needed by the MMU for address translation are **part of the state of a running program**
- The MMU abstracts the specifics of the actual memory hardware

Test your understanding

- ➊ Can a user program choose to write to a specific location in *physical memory*?
- ➋ Can a user program find out where a variable lives in *physical memory*?
- ➌ Are the answers to these questions a good thing or a bad thing?

Operating Systems and Concurrency

Introduction 3
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

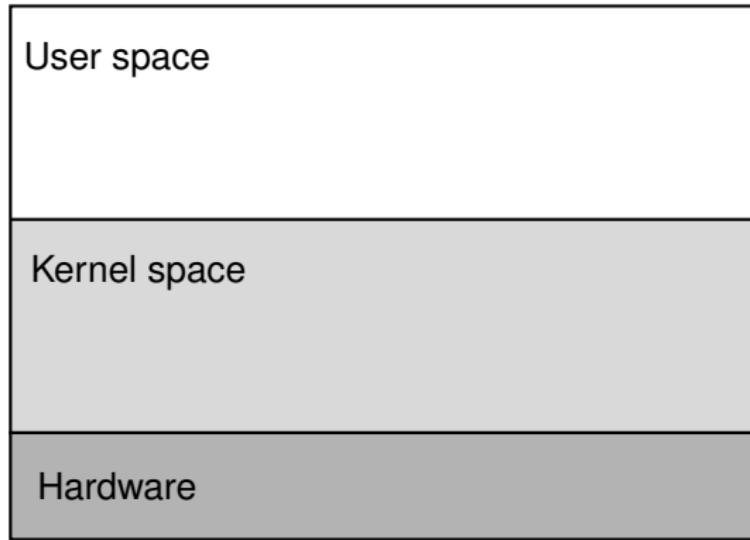
- ① Operating system **kernel** design closely linked to hardware capabilities
- ② **Registers** provide data consumed by the CPU
- ③ The **MMU** translates **logical** to **physical addresses** abstracting the details of the memory hardware

Goals for Today

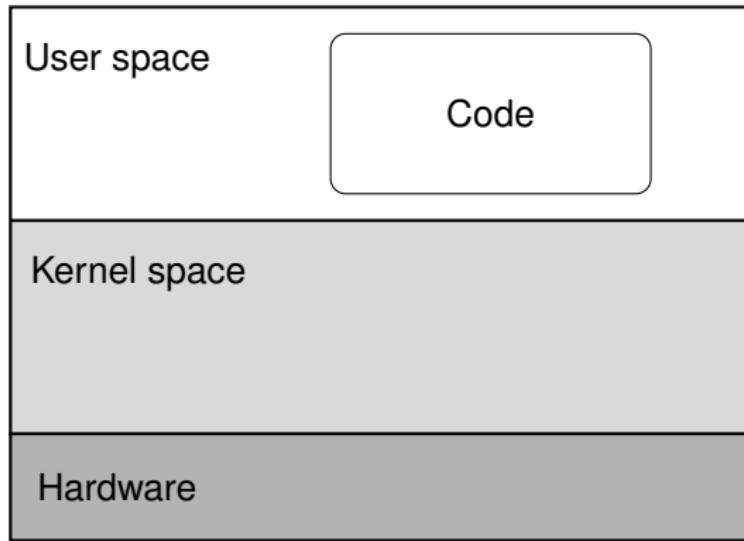
Overview

- **Kernel and user space**
- **Interrupts**
- **System calls**
- **The C programming language** - an operating systems perspective.

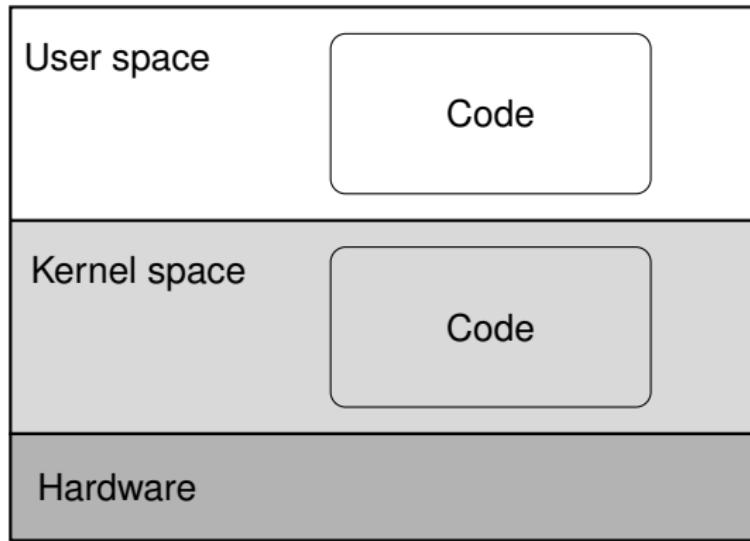
Kernel and user space



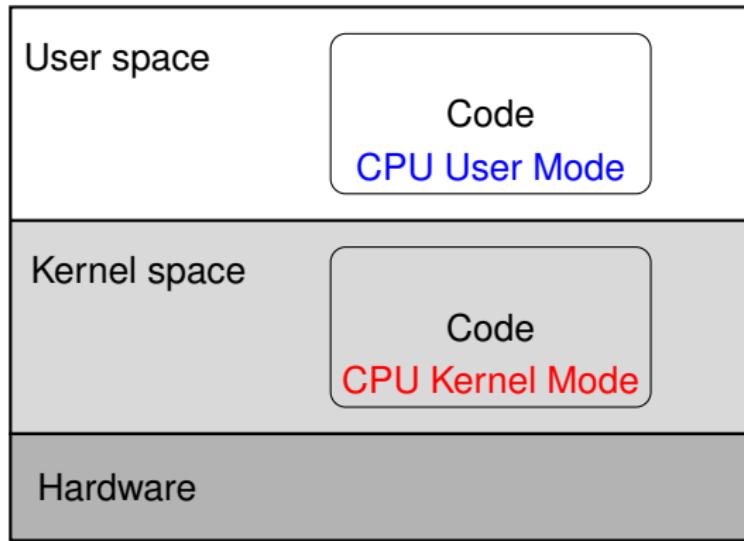
Kernel and user space



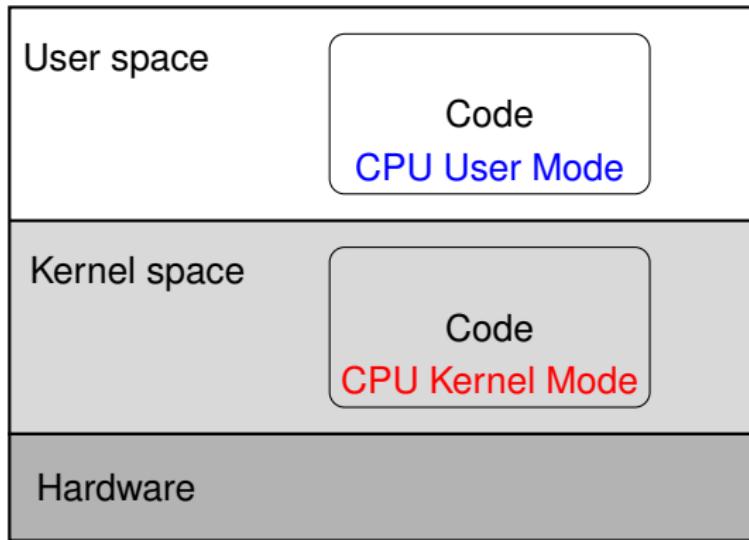
Kernel and user space



Kernel and user space



Kernel and user space



Terminology

The code running in kernel mode is often referred to as the **kernel**. Although this is the core of an operating system, typically they will also have user processes for user interfaces, scheduling daemons, ...

Interrupts

Entering the kernel to respond to events

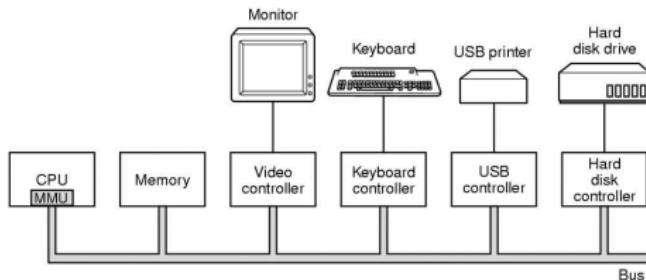


Figure: Simplified computer model (Tanenbaum, 2014)

The CPU cannot usefully live in isolation

Programs, and therefore the CPU, must be able to respond to various events.

For example:

- Connected hardware devices might need to communicate
- The passing of time
- Bad things happening - division by zero, hardware faults...

Interrupts

Entering the kernel to respond to events

Interrupts are the mechanism by which such events are handled.

- Abstractly, **an interrupt is a mechanism for changing the normal flow of execution**
- Can happen **asynchronously**, triggered by unpredictable factors external to the CPU, such as user input
 - Sometimes the term **interrupt** is reserved for this class
- Can also happen **synchronously**, triggered directly by the CPU executing an instruction
 - Sometime the term **exception** is used for this class

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

- ① The CPU is doing some work - for example running a user process doing some calculations

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

- ① The CPU is doing some work - for example running a user process doing some calculations
- ② An interrupt is signalled by a hardware device - for example to indicate some IO data is available

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

- ① The CPU is doing some work - for example running a user process doing some calculations
- ② An interrupt is signalled by a hardware device - for example to indicate some IO data is available
- ③ The CPU records aspects of its current state, **switches to kernel mode**, and runs code in a **handler** to service the interrupt

Interrupts

Entering the kernel to respond to events

Interrupt mechanism (sketch)

- ① The CPU is doing some work - for example running a user process doing some calculations
- ② An interrupt is signalled by a hardware device - for example to indicate some IO data is available
- ③ The CPU records aspects of its current state, **switches to kernel mode**, and runs code in a **handler** to service the interrupt
- ④ Once completed, the CPU is returned to processing other tasks

Interrupts

Entering the kernel to respond to events

Challenges

Handling interrupts is one of the more challenging tasks of operating system implementation.

Interrupts

Entering the kernel to respond to events

Challenges

Handling interrupts is one of the more challenging tasks of operating system implementation.

- Interrupts can come at any time
 - Ideally handling interrupts should not take long
 - Handlers may split work into a **top** component dealt with immediately, and a **bottom** component scheduled to be dealt with later

Interrupts

Entering the kernel to respond to events

Challenges

Handling interrupts is one of the more challenging tasks of operating system implementation.

- Interrupts can come at any time
 - Ideally handling interrupts should not take long
 - Handlers may split work into a **top** component dealt with immediately, and a **bottom** component scheduled to be dealt with later
- Interrupts may be interrupted by other interrupts - it must be possible to **nest** interrupt handlers

Interrupts

Entering the kernel to respond to events

Challenges

Handling interrupts is one of the more challenging tasks of operating system implementation.

- Interrupts can come at any time
 - Ideally handling interrupts should not take long
 - Handlers may split work into a **top** component dealt with immediately, and a **bottom** component scheduled to be dealt with later
- Interrupts may be interrupted by other interrupts - it must be possible to **nest** interrupt handlers
- Sometimes critical code cannot be interrupted and they must be temporarily disabled

Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

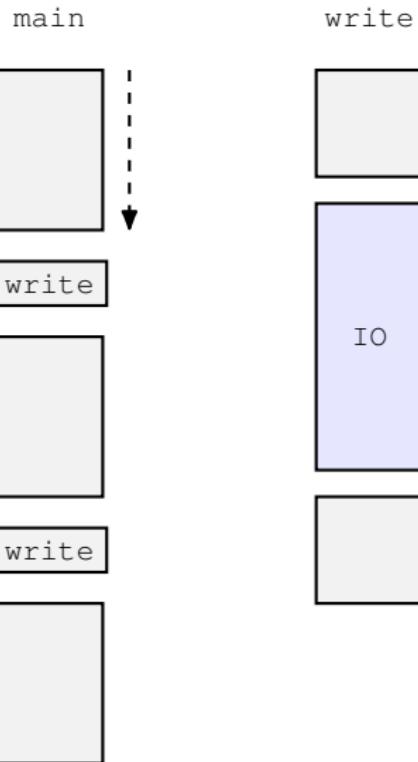


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

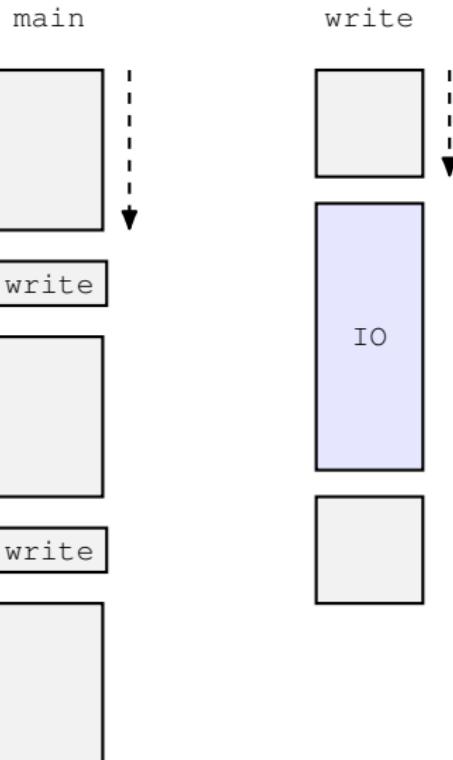


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

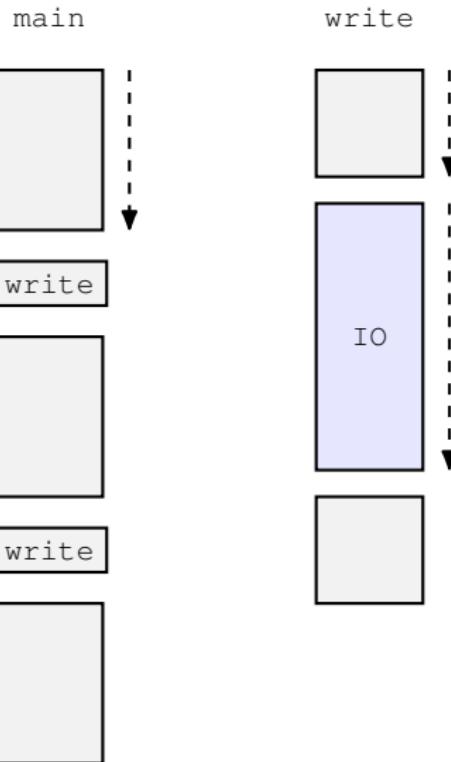


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

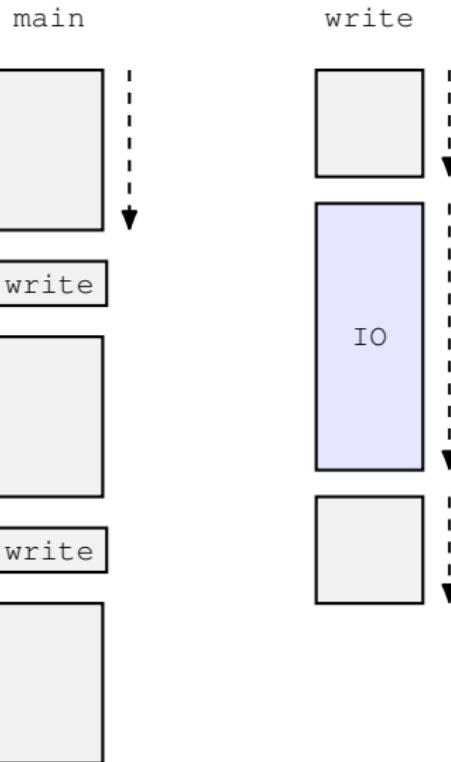


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

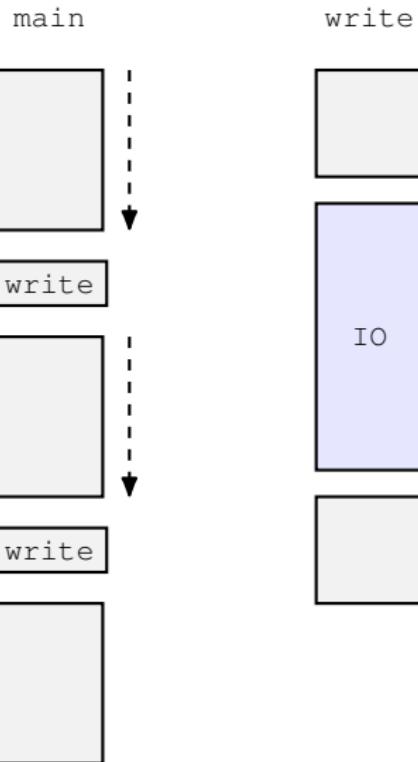


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

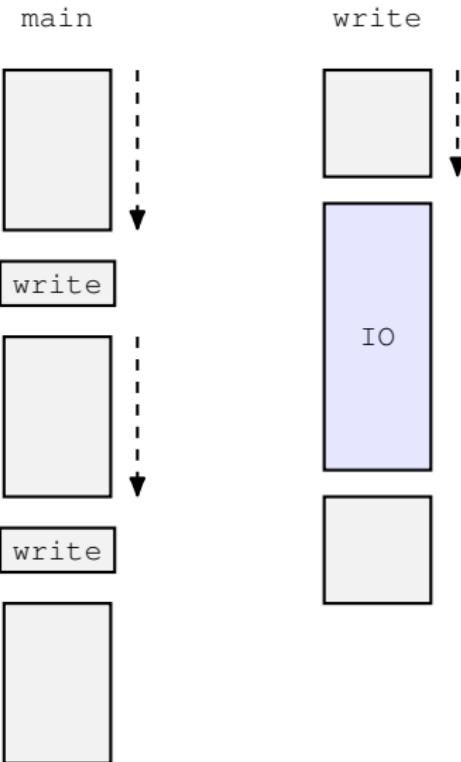


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

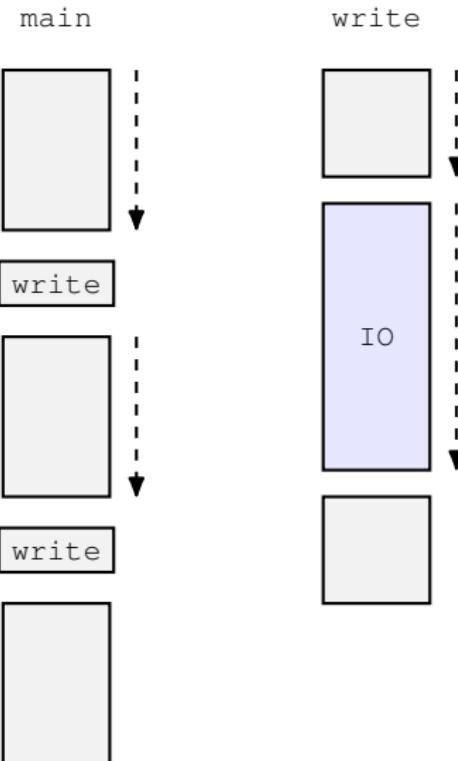


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

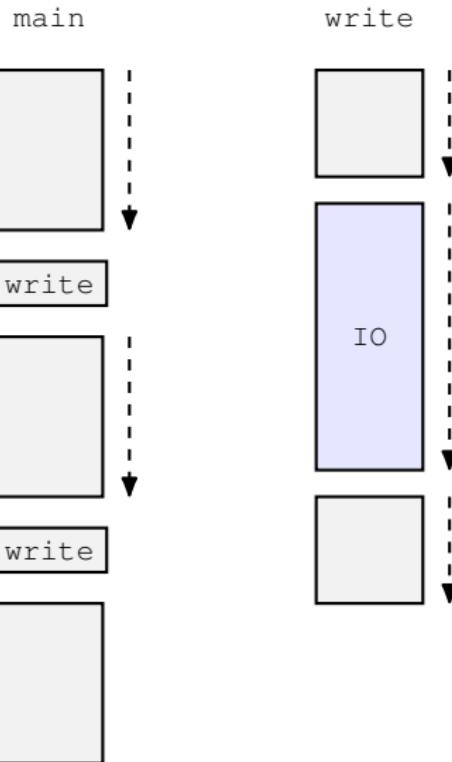


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.

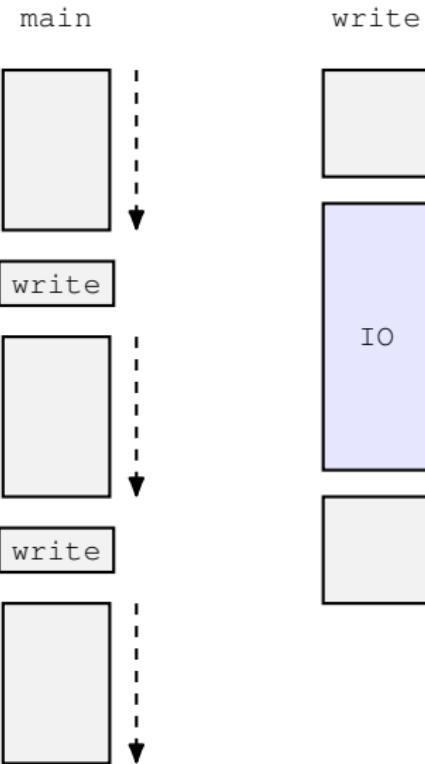


Interrupts

CPU utilisation

Blocking IO

Naive IO which wastes CPU cycles waiting for slow devices to respond.



Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



send_write



write_handler



send_write



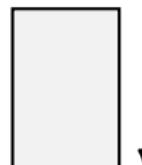
Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



send_write



write_handler



send_write



Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



send_write



write_handler



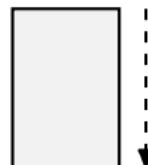
Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



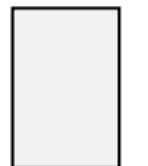
send_write



write_handler



send_write



Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



send_write



write_handler



send_write



Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



send_write



write_handler

send_write



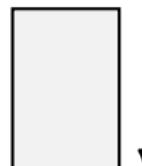
Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

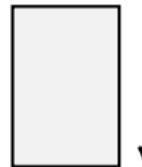
main



send_write



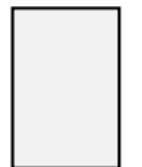
send_write



write_handler



send_write



Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



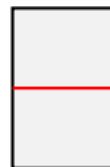
send_write



write_handler



send_write



Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



send_write



write_handler



send_write



Interrupts

CPU utilisation

Interrupts

Better CPU usage supported by interrupts “calling back” when there is work to do.

main



send_write



send_write



write_handler



send_write



System calls

Entering the kernel by software request

System calls are how programs request services from the operating system.

Example (Typical system calls)

- Requesting memory
- Accessing files
- Running programs (processes)
- Accessing concurrency features

System calls

Entering the kernel by software request

We can distinguish between an API and systems calls.

- An API is a programming interface - typically a **library of functions that run in user space**, for example pthreads
- To provide the required features **an API may need to make system calls to access the required functionality**
- The relationship is *not* necessarily one-to-one - a **single API function may invoke zero, one or many system calls**
- Often programmers use less fussy terminology, and simply refer to the API functions as system calls

System calls

Entering the kernel by software request

Question

How can a system call allow a user process to run kernel space code?

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

- 1 Some user code is running

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

- ① Some user code is running
- ② A system call is required - each has a **unique system call number**

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

- ① Some user code is running
- ② A system call is required - each has a **unique system call number**
- ③ The system call number is stored in a designated register

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

- ① Some user code is running
- ② A system call is required - each has a **unique system call number**
- ③ The system call number is stored in a designated register
- ④ The system call parameters are stored in designated registers

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

- ① Some user code is running
- ② A system call is required - each has a **unique system call number**
- ③ The system call number is stored in a designated register
- ④ The system call parameters are stored in designated registers
- ⑤ A synchronous interrupt (exception) is triggered by an instruction referred to as a **trap**

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

- ① Some user code is running
- ② A system call is required - each has a **unique system call number**
- ③ The system call number is stored in a designated register
- ④ The system call parameters are stored in designated registers
- ⑤ A synchronous interrupt (exception) is triggered by an instruction referred to as a **trap**
- ⑥ The interrupt is handled by kernel mode code, which calls a **system call service routine** which delivers the required functionality

System calls

Entering the kernel by software request

System call mechanisms (naive sketch)

- ① Some user code is running
- ② A system call is required - each has a **unique system call number**
- ③ The system call number is stored in a designated register
- ④ The system call parameters are stored in designated registers
- ⑤ A synchronous interrupt (exception) is triggered by an instruction referred to as a **trap**
- ⑥ The interrupt is handled by kernel mode code, which calls a **system call service routine** which delivers the required functionality
- ⑦ The operating system continues running the calling code

System calls

Entering the kernel by software request

System call mechanisms (slightly less naive)

- The system call service routine **may not actually service the request immediately** - for example if it must wait on some resource such as IO
- The operating system **may not continue running the original caller**
- **System calls are the kernel's big chance to get work done for the benefit of everything running on the system**

System calls

Entering the kernel by software request

System call mechanisms (fussier details)

- Details vary by operating system
- Modern approaches may avoid interrupt based mechanisms and use other CPU support for efficiency reasons

The C Programming Language

Criteria for an OS Implementation Language

Question

Why would we choose such an old language like C for OS implementation?

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello Operating Systems and Concurrency\n");
5 }
```

The C Programming Language

Criteria for an OS Implementation Language

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

The C Programming Language

Criteria for an OS Implementation Language

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

- A slow OS slows every program that it runs.

The C Programming Language

Criteria for an OS Implementation Language

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

- A slow OS slows every program that it runs.
- Mobile hardware must also consider energy efficiency.

The C Programming Language

Criteria for an OS Implementation Language

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

- A slow OS slows every program that it runs.
- Mobile hardware must also consider energy efficiency.
- Bias toward performance over simplicity, elegance and maintainability.

The C Programming Language

Criteria for an OS Implementation Language

Performance

As general purpose software, an OS must be fast enough for as many users as possible.

- A slow OS slows every program that it runs.
- Mobile hardware must also consider energy efficiency.
- Bias toward performance over simplicity, elegance and maintainability.
- Cannot sacrifice correctness.

The C Programming Language

Criteria for an OS Implementation Language

Portability

As OS development is difficult, reuse is desirable.

The C Programming Language

Criteria for an OS Implementation Language

Portability

As OS development is difficult, reuse is desirable.

- Must run directly on the hardware - no interpreted languages.

The C Programming Language

Criteria for an OS Implementation Language

Portability

As OS development is difficult, reuse is desirable.

- Must run directly on the hardware - no interpreted languages.
- Ideally must be able to compile for many different hardware platforms.

The C Programming Language

Criteria for an OS Implementation Language

Predictability

The behaviour of an OS must be predictable for user programs.

The C Programming Language

Criteria for an OS Implementation Language

Predictability

The behaviour of an OS must be predictable for user programs.

- Example - games cannot tolerate unpredictable delays slowing frame rates and responsiveness.

The C Programming Language

Criteria for an OS Implementation Language

Predictability

The behaviour of an OS must be predictable for user programs.

- Example - games cannot tolerate unpredictable delays slowing frame rates and responsiveness.
- Unpredictable behaviour such as garbage collection is inappropriate.

Recap

Take-Home Message

- Kernel mode code has more privileges than user code
- **Interrupts change the normal flow** of execution to invoke kernel code
- **System calls allow us to run kernel code** to access services of the operating system

Test your understanding

- What would be the advantages and disadvantages of a small kernel with a limited collection of system calls?
- What would be the advantages and disadvantages of a large kernel with a rich collection of system calls?
- How much harm can a bug in a user process cause?
- How much harm can a bug in the kernel cause?

Attendance Barcode

COMP2007: Operating Systems & Concurrency
Week 3 – 3:00pm Monday – 09 October 2023



Operating Systems and Concurrency

Processes 1
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

- **Kernel mode** code has more privileges than user code
- **Interrupts** change the flow of execution to invoke kernel code
- **System calls** allow us to run kernel code to access services of the operating system

Goals for Today

Overview

- Introduction to **processes** and their **implementation**
- Process **states** and state **transitions**
- **System calls** for process management

Processes

Definition

- **A process is an abstraction of a running instance of a program**
 - A program is **passive** and “sits” on a disk
 - A process has **control structures** associated with it, may be **active**, and may have **resources** assigned to it (e.g. I/O devices, memory, processor)
- All the information necessary to administer a process is stored by the kernel in a **process control block (PCB)**.
- All the process control blocks are recorded in the **process table**.

Processes

Memory Image of Processes

- A process' memory image contains:
 - The program **code** (could be shared between multiple processes running the same code)
 - A **data** segment, **stack** and **heap**
- Every process has its own **logical address space**, in which the **stack** and **heap** are placed at **opposite sides** to allow them to grow

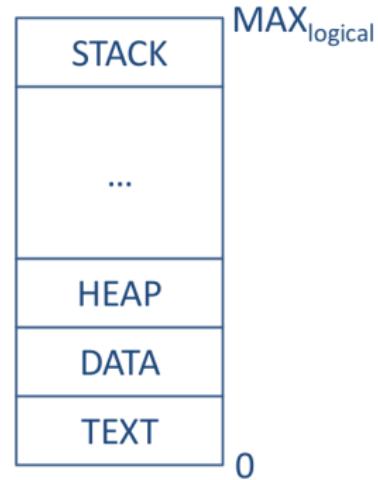
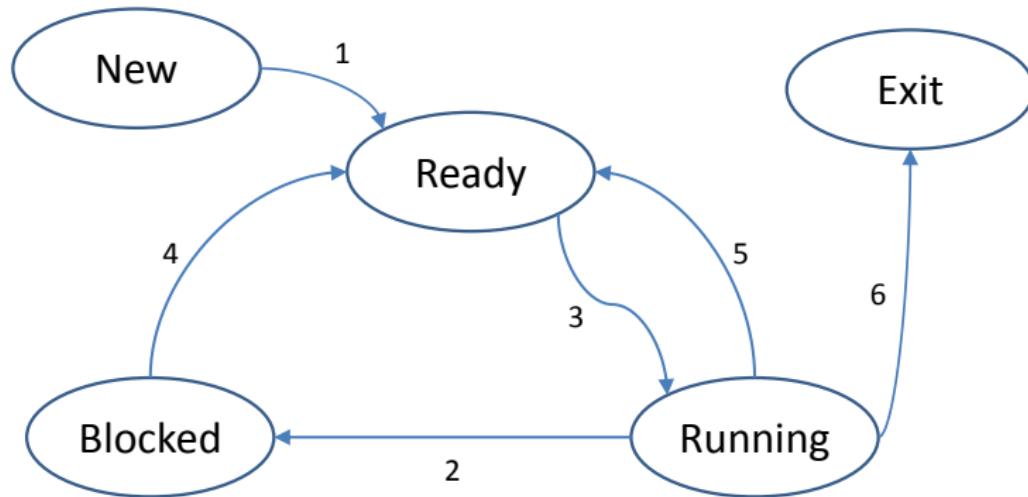


Figure: Representation of a process in memory

Process States and Transitions

Diagram



Process States and Transitions

States

- A **new** process has just been created. It has a PCB and is waiting to be admitted, although it may not yet be in memory.
- A **ready** process is waiting for CPU to become available.
- A **running** process is currently having its instructions executed by the CPU.
- A **blocked** process cannot continue, e.g. is waiting for I/O
- A **terminated** process is no longer executable. The data structures - PCB - may be temporarily preserved.

Process States and Transitions

States

- A **new** process has just been created. It has a PCB and is waiting to be admitted, although it may not yet be in memory.
- A **ready** process is waiting for CPU to become available.
- A **running** process is currently having its instructions executed by the CPU.
- A **blocked** process cannot continue, e.g. is waiting for I/O
- A **terminated** process is no longer executable. The data structures - PCB - may be temporarily preserved.
- A **suspended** process is swapped out (not discussed further)

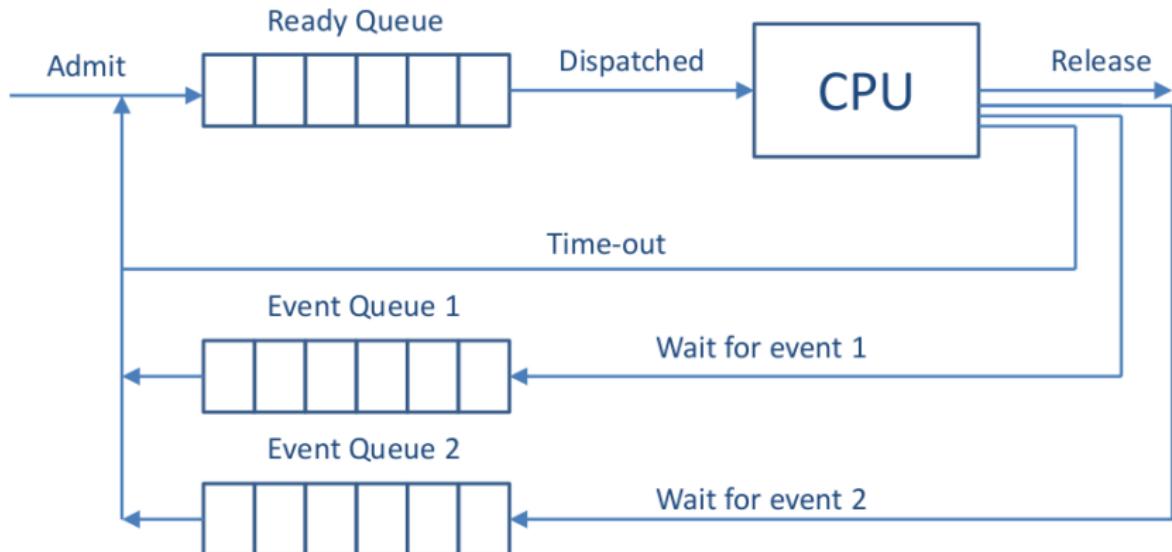
Process States and Transitions

Transitions

- State transitions include:
 - ① **New → ready**: admit the process and commit to execution
 - ② **Running → blocked**: e.g. process is waiting for input or carried out a system call
 - ③ **Ready → running**: the process is selected by the **process scheduler**
 - ④ **Blocked → ready**: event happens, e.g. I/O operation has finished
 - ⑤ **Running → ready**: the process surrenders the CPU, for example due to an interrupt or by pause
 - ⑥ **Running → exit**: process has finished, e.g. program ended or exception encountered
- **Interrupts and system calls** drive these transitions.

Process States and Transitions

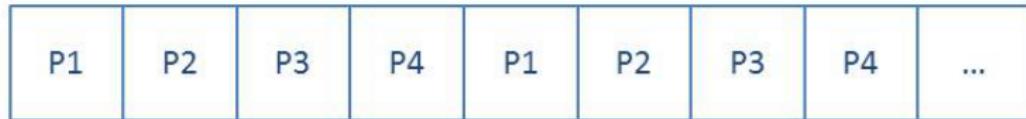
OS Queues



Context Switching

Multi-programming

- Modern computers are **multi-programming** systems
- Assuming a **single processor system**, the instructions of individual processes are executed **sequentially**
 - Multi-programming goes back to the “**MULTICS**” age
 - Multi-programming is achieved by **interleaving** the execution of processes, dividing the CPU time into **time-slices**
 - Control is exchanged between processes via a procedure known as **context switching**
 - A **trade-off** exists between the length of the **time-slice** and the **context switch time**
 - **True parallelism** requires **hardware support**



TIME

Context Switching

Multi-programming (Cont'd)

- When a **context switch** takes place, the system **saves the state** of the old process and **loads the state** of the new process (creates **overhead**)
 - Saved** ⇒ the process control block is **updated**
 - (Re-)started** ⇒ the process control block **read**

Context Switching

Multi-programming (Cont'd)

Short time slices result in good response times but low effective utilisation. For example, assume both context switches and time slices take 1ms. Then:

- It will take $99 \times (1 + 1) = 198\text{ms}$ for the last of 100 processes to start running.
- $\frac{1}{1+1} = 0.5$ of the CPU time is doing useful work.



TIME

Context Switching

Multi-programming (Cont'd)

Long time slices result in poor response times but better effective utilisation. For example, assume context switches take 1ms and time slices are 100ms. Then:

- It will take $99 \times (100 + 1) = 9999\text{ms}$ for the last of 100 processes to start running.
- $\frac{100}{1+100} = 0.99$ of the CPU time is doing useful work



TIME

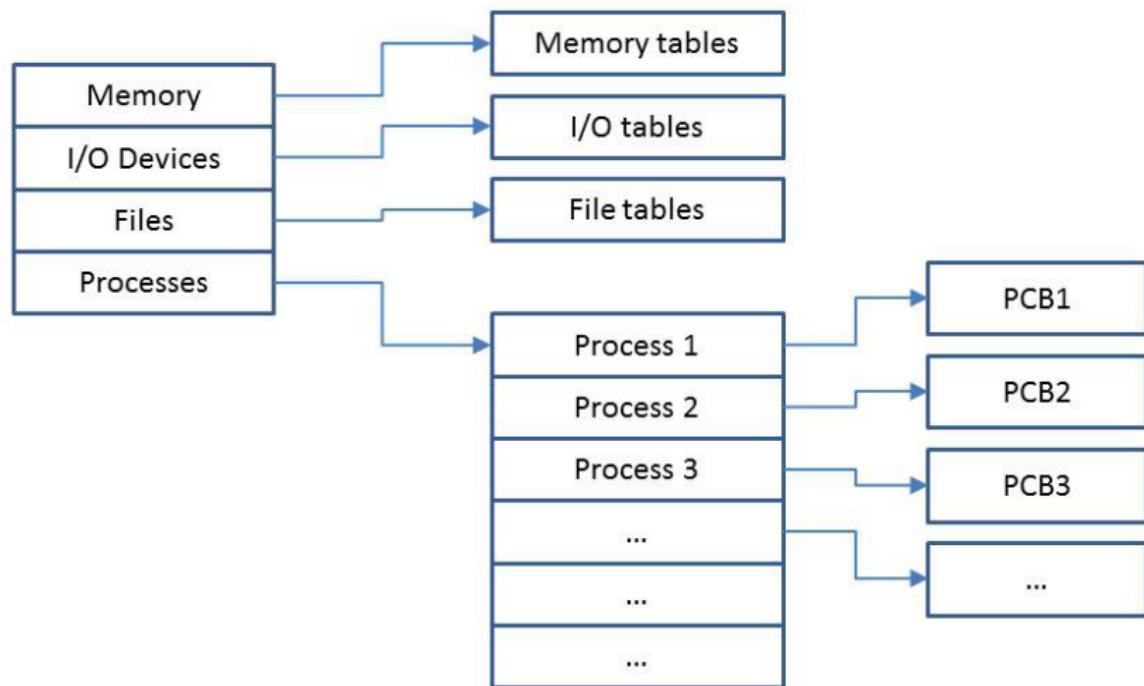
Context Switching

Multi-programming (Cont'd)

- A **process control block** contains three types of **attributes**:
 - **Process identification** (PID, UID, Parent PID)
 - **Process control information** (process state, scheduling information, etc.)
 - **Process state information** (user registers, program counter, stack pointer, program status word, memory management information, files, etc.)
- **Process control blocks** are **kernel data structures**, i.e. they are **protected** and only accessible in **kernel mode**!
 - Allowing user applications to access them directly could **compromise their integrity**
 - The **operating system manages** them on the user's behalf through **system calls** (e.g. to set **process priority**)

Process Implementation

Tables and Control Blocks



Process Implementation

Tables and Control Blocks

- An operating system **maintains information** about the status of “resources” in **tables**
 - **Process tables** (process control blocks)
 - **Memory tables** (memory allocation, memory protection, virtual memory)
 - **I/O tables** (availability, status, transfer information)
 - **File tables** (location, status)
- The **process table** holds a **process control block** for each process, allocated upon **process creation**
- Tables are maintained by the **kernel** and are usually **cross referenced**

Context Switching

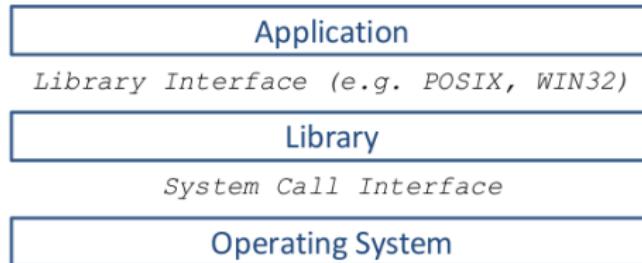
Switching Processes

1. Save process state (program counter, registers)
2. Update PCB (running → ready/blocked)
3. Move PCB to appropriate queue (ready/blocked)
4. Run scheduler, select new process
5. Update to running state in the new PCB
6. Update memory management unit (MMU)
7. Restore process

System Calls

Process Creation

- The true system calls are “**wrapped**” in the **OS libraries** (e.g. libc) following a well defined interface (e.g. POSIX, WIN32 API)
- For example, on Unix-like operating systems `fork` is called to create a copy of a process. On Linux, the underlying system call used to implement `fork` is `clone`.



System Calls

Process Termination

- System calls via `exit` and `abort` can be used to explicitly **notify the OS** that the **process has terminated**
 - Resources must be de-allocated
 - Output must be flushed
 - Process admin may have to be carried out
- A system calls to terminate other processes:
 - UNIX/Linux: `kill()`
 - Windows: `TerminateProcess()`

Process Creation in Linux

Fork

- `fork()` creates an **exact copy** of the current process
 - The first instruction carried out by the child is the first one after the `fork` call
- `fork()` returns the **process identifier** of the child process **to the parent process.**
- `fork()` **returns 0** to the **child process.**

Process Creation in Linux

The Fork and Exec Pattern

A common pattern is the following sequence:

- ① Call `fork()` to create an exact copy of the current process.
- ② In the child process call one of the “exec” functions to replace the current process with a new program.

Process Creation in Linux

The Fork and Exec Pattern

A common pattern is the following sequence:

- ① Call `fork()` to create an exact copy of the current process.
- ② In the child process call one of the “exec” functions to replace the current process with a new program. E.g. `execl("/bin/ls", "ls", "-l", 0)` replaces the current process with ls,

Process Creation in Linux

The Fork and Exec Pattern

A common pattern is the following sequence:

- ① Call `fork()` to create an exact copy of the current process.
- ② In the child process call one of the “exec” functions to replace the current process with a new program. E.g. `execl("/bin/ls", "ls", "-l", 0)` replaces the current process with ls,

This is a typical pattern of calls in a Unix shell such as bash.

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0) {
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Processes

Process Creation in Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t const pid = fork();
    if(pid < 0){
        printf("Fork failure\n");
        return -1;
    } else if(pid == 0) {
        printf("Child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    } else {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child %d returned %d\n", pid, status);
    }
}
```

Test your understanding

- Why might you run `fork` without running a subsequent `exec`?
- Do you always need to call `exit` to end a process?
- Why does a process control block contain data about register contents?
- Why might it be useful to retain a process control block for a terminated process?

Operating Systems and Concurrency

Processes 2, Scheduling
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

- Processes have “**control structures**” associated with them, the **process control blocks** and **process tables**.
- Processes can have different **states** and the kernel triggers **transitions** between these states.
- The operating system maintains multiple **process queues**.
- The operating system **manages processes** on the user’s behalf.

Goals for Today

Overview

- Introduction to **process scheduling**
- Types of **process schedulers**
- **Evaluation criteria** for scheduling algorithms
- Typical **process scheduling algorithms**

Process Scheduling

Context

- The OS is responsible for **managing** and **scheduling processes**
 - Decide when to **admit** processes to the system (new → ready)
 - Decide which process to **run** next (ready → run)
 - Decide when and which processes to **interrupt** (running → ready)
- It relies on the **scheduler (dispatcher)** to decide which process to run next, which uses a **scheduling algorithm** to do so
- The type of algorithm used by the scheduler is influenced by the **type of operating system** (e.g., real time vs. batch)

Process Schedulers

Classification by Time Horizon

- **Long term:** applies to **new processes** and controls the degree of multiprogramming by deciding which processes to admit to the system when
 - A good **mix of CPU and I/O bound processes** is favourable to keep all resources as busy as possible
 - **Usually absent** in popular modern OS
- **Medium term:** controls swapping and the degree of multi-programming
- **Short term:** decide which process to run next
 - Manages the **ready queue**
 - Invoked very **frequently**, hence must be **fast**
 - Usually called in response to **clock interrupts, I/O interrupts, or blocking system calls**

Process Schedulers

Classification by Time Horizon

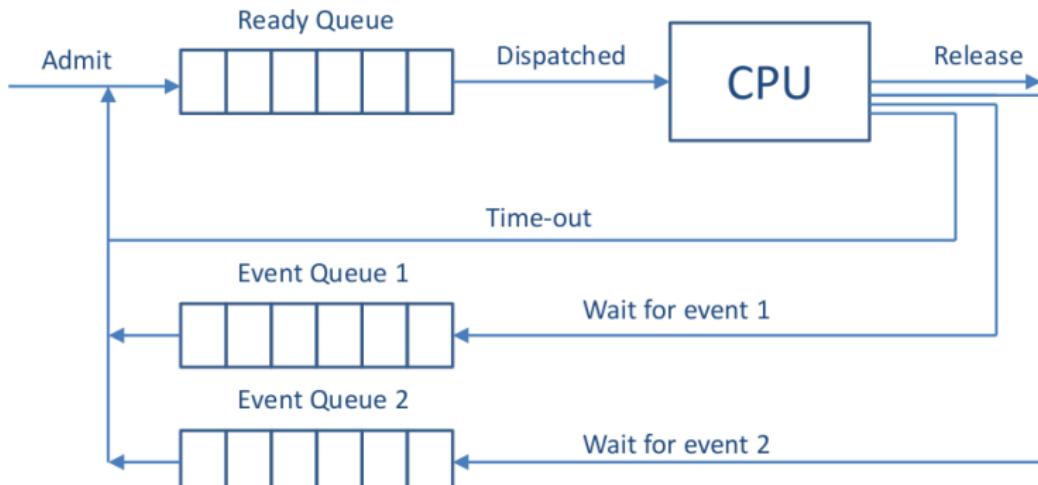


Figure: Queues in OS

Process Schedulers

Classification by Approach

- **Non-preemptive:** processes are only interrupted **voluntarily** (e.g., I/O operation or “nice” system call – `yield()`)
 - Windows 3.1 and DOS were non-preemptive
- **Preemptive:** processes can be **interrupted forcefully or voluntarily**
 - Typically **driven by interrupts from a system clock.**
 - Requires additional context switches which generate **overhead**, too many of them should be avoided (recall last lecture)
 - Prevents processes from **monopolising the CPU**
 - **Most popular** modern operating systems are preemptive

Performance Assessment

Criteria

- **User oriented criteria:**
 - **Response time:** minimise the time between creating the job and its first execution
 - **Turnaround time:** minimise the time between job creation and completion
 - **Predictability:** minimise the variance in processing times
- **System oriented criteria:**
 - **Throughput:** maximise the number of jobs processed per hour
 - **Fairness:**
 - Are processing power/waiting time equally distributed?
 - Are some processes kept waiting excessively long (**starvation**)
- Evaluation criteria can be **conflicting**, i.e., **improving the response time** may require **more context switches**, and hence **worsen the throughput** and **increase the turn around time**

Scheduling Algorithms

Overview

- **Algorithms** considered:
 - 1 First Come First Served (**FCFS**)/ First In First Out (**FIFO**)
 - 2 **Shortest job first**
 - 3 **Round robin**
 - 4 **Priority queues**
- Performance measures used:
 - **Average response time:** the average of the time taken for all the processes to start
 - **Average turnaround time:** the average time taken for all the processes to finish

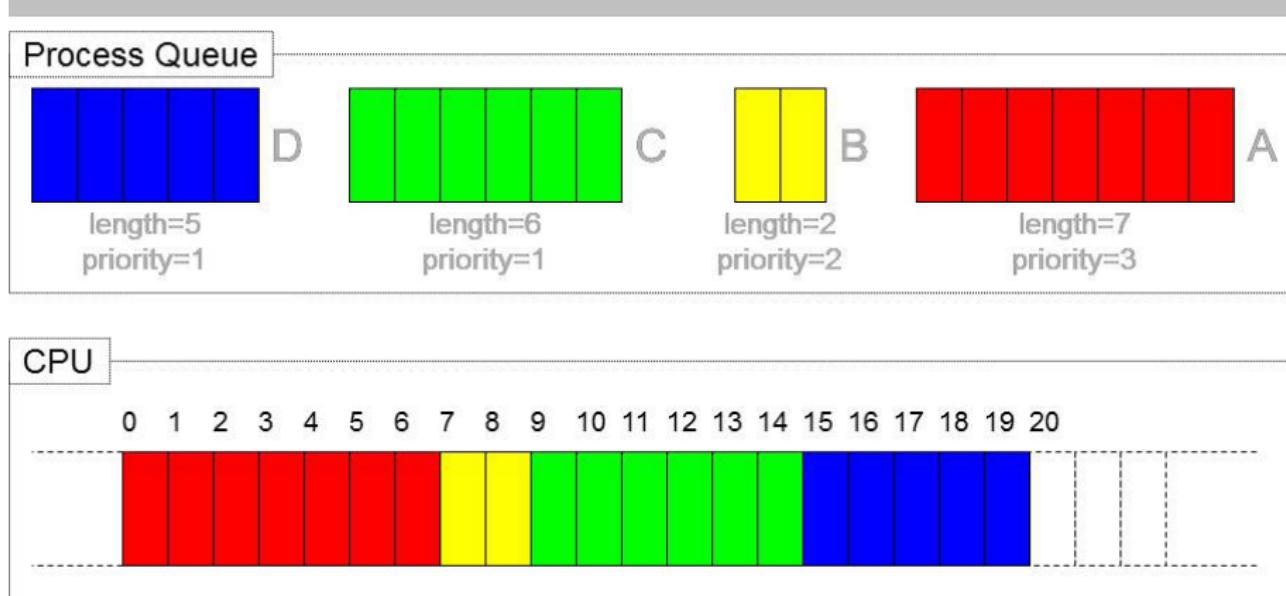
Scheduling Algorithms

First Come First Served

- Concept: a **non-preemptive algorithm** that operates as a **strict queueing mechanism** and schedules the processes in the same order that they were added to the queue
- Advantages: **positional fairness** and easy to implement
- Disadvantages:
 - **Favours long processes** over short ones (think of the supermarket checkout!)
 - Could **compromise resource utilisation**, i.e., CPU vs. I/O devices

Scheduling Algorithms

First Come First Served



- Average response time = $0 + 7 + 9 + 15 = \frac{31}{4} = 7.75$
- Average turn around time = $7 + 9 + 15 + 20 = \frac{51}{4} = 12.75$

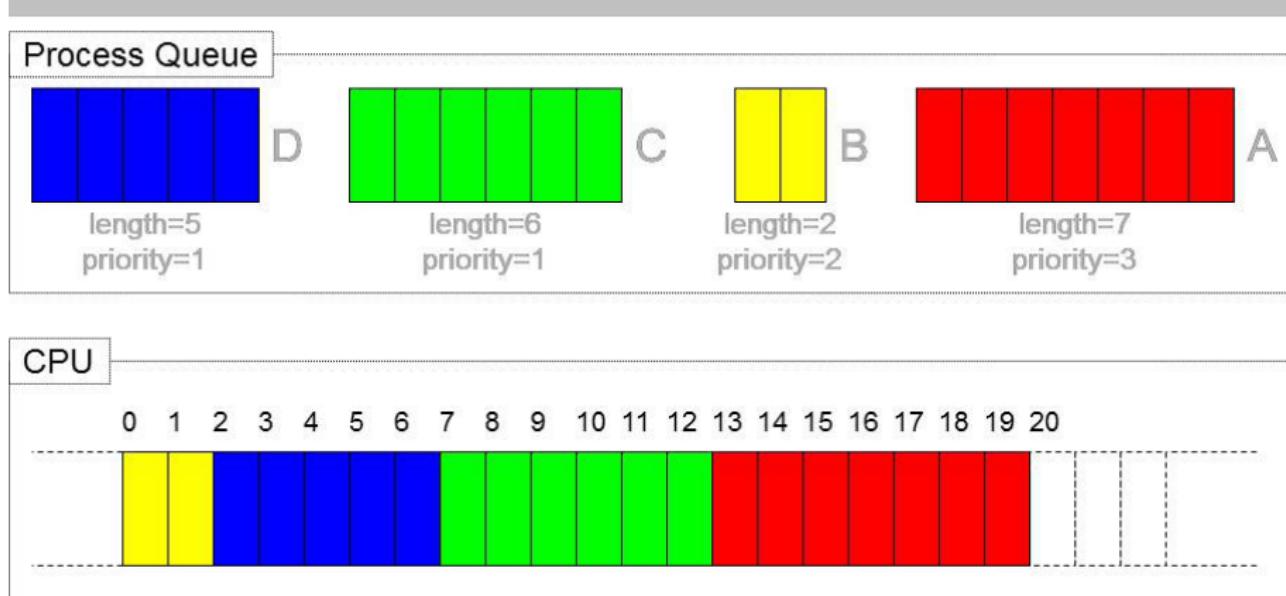
Scheduling Algorithms

Shortest Job First

- Concept: A **non-preemptive algorithm** that starts processes in order of **ascending processing time** using a provided/known estimate of the processing
- Advantages: always result in the **optimal turn around time**
- Disadvantages:
 - **Starvation** might occur
 - **Fairness and predictability** are compromised
 - **Processing times have to be known** beforehand

Scheduling Algorithms

Shortest Job First



- Average response time = $0 + 2 + 7 + 13 = \frac{22}{4} = 5.5$
- Average turn around time = $2 + 7 + 13 + 20 = \frac{42}{4} = 10.5$

Scheduling Algorithms

Round Robin

- Concept: a **preemptive version of FCFS** that forces **context switches** at **periodic intervals** or **time slices**
 - Processes **run in the order that they were added** to the queue
 - Processes are forcefully **interrupted by the timer**
- Advantages:
 - Improved **response time**
 - Effective for general purpose **interactive/time sharing systems**
- Disadvantages:
 - Increased **context switching** and thus overhead
 - **Favours CPU bound processes** (which usually run long) over I/O processes (which do not run long)
 - Can be prevented by working with multiple queues?
 - Can **reduce to FCFS**

Exam 2013-2014: Round Robin is said to favour CPU bound processes over I/O bound processes. Explain why may this be the case (if this is the case at all)?

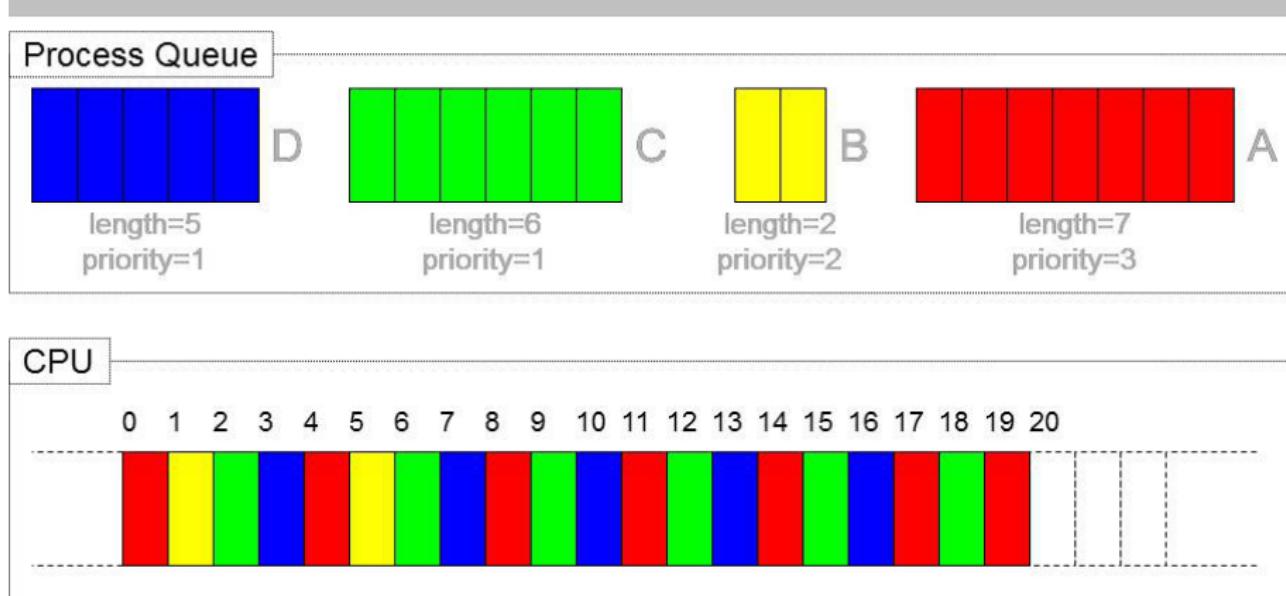
Scheduling Algorithms

Round Robin

- The **length** of the **time slice** must be carefully considered!
- For instance, assuming a **multi-programming system** with **preemptive scheduling** and a **context switch time** of 1ms:
 - E.g., a **good (low) response time** is achieved with a **small time slice** (e.g. 1ms) ⇒ low throughput
 - E.g., a **high throughput** is achieved with a **large time slice** (e.g. 1000ms) ⇒ high response time
- If a time slice is only **used partially**, the next process **starts immediately**

Scheduling Algorithms

Round Robin



- Average response time = $0 + 1 + 2 + 3 = \frac{6}{4} = 1.5$
- Average turn around time = $6 + 17 + 19 + 20 = \frac{62}{4} = 15.5$

Scheduling Algorithms

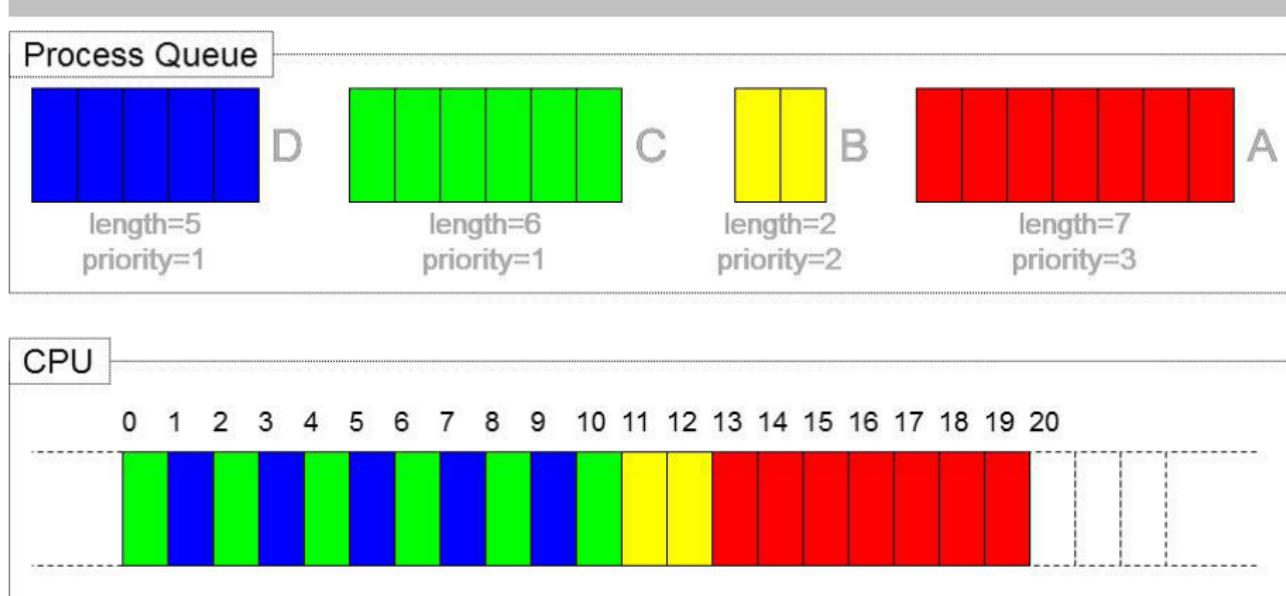
Priority Queues

- Concept: A **preemptive algorithm** that schedules processes by priority (high → low)
 - A **round robin** is used **within the same priority levels**
 - The process priority is saved in the **process control block**
- Advantages: can **prioritise I/O bound jobs**
- Disadvantages: low priority processes may suffer from **starvation** (when priorities are static)

Exam 2013-2014: Out of the following four scheduling algorithms, which one can lead to starvation: FCFS, shortest job first, round robin, highest priority first? Explain your answer.

Scheduling Algorithms

Priority Queues



- Average response time = $0 + 1 + 11 + 13 = \frac{25}{4} = 6.25$
- Average turn around time = $10 + 11 + 13 + 20 = \frac{54}{4} = 13.5$

Scheduling Algorithms

Priority Queues

- Give the **order in which the processes are scheduled** when using **priority queues**, together with the **times at which they will start, end, and are interrupted** (all processes are available at the time of scheduling)
- You can assume a time slice of 15 milliseconds
- Calculate the **average response and turn around time**

	FCFS Position	CPU burst time	Priority
Process A	1	67	1 (high)
Process B	2	37	1 (high)
Process C	3	14	2 (low)
Process D	4	16	2 (low)

Scheduling Algorithms

Priority Queues

- Solution:
 - Sequence: A(15) \Rightarrow B(15) \Rightarrow A(15) \Rightarrow B(15) \Rightarrow A(15) \Rightarrow B(7) \Rightarrow A(15) \Rightarrow A(7) \Rightarrow C(14) \Rightarrow D(15) \Rightarrow D(1)
 - Average response time = $(0 + 15 + 104 + 118) / 4$
 - Average turnaround time = $(82 + 104 + 118 + 134) / 4$
- Note: we ignore **context switch time**

Test your understanding

- On a non-preemptive operating systems, what sort of things can a process do without potentially “volunteering” to cede the CPU to another process.
- Using the non-preemptive shortest job first scheduler, does the shortest job run on the CPU until it is completed?

Operating Systems and Concurrency

Processes 3: Threads
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

- **Types of schedulers:** preemptive/non-preemptive, long/medium/short term)
- Performance **evaluation criteria**
- Scheduling **algorithms:** FCFS, SJF, Round Robin, Priority Queues

Goals for Today

Overview

- ① Threads vs. processes
- ② Different thread implementations
- ③ POSIX Threads (PThreads)

Threads

Threads from an OS Perspective

- A process consists of two **fundamental units**
 - **Resources:** all related resources are grouped together
 - A logical address space containing the process image (program, data, heap, stack)
 - Files, I/O devices, I/O channels, ...
 - **Execution trace**, i.e., an entity that gets executed
- A process can **share its resources** between **multiple execution traces**, i.e., multiple threads running in the same resource environment

Threads

Threads from an OS Perspective (Cont'ed)

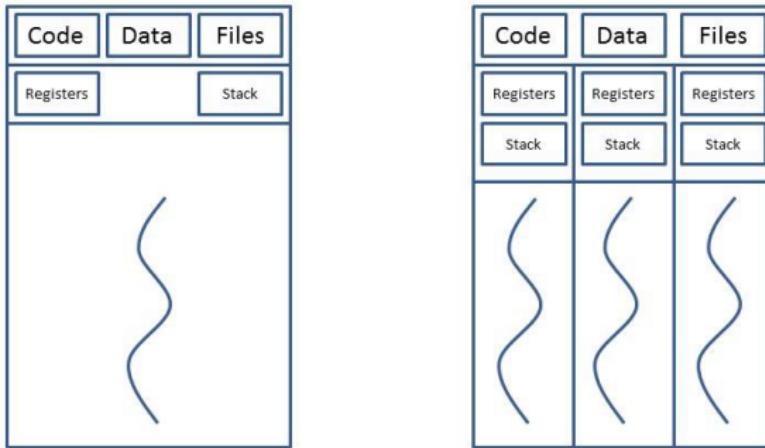


Figure: Single threaded process (left), multi-threaded process (right)

Threads

Threads from an OS Perspective (Cont'ed)

- Every thread has its own **execution context** (e.g. program counter, stack, registers)
- All threads have **access** to the process' **shared resources**
 - E.g. files, one thread opens a file, all threads of the same process can access the file
 - Global variables, memory, etc. (\Rightarrow synchronisation!)
- Similar to processes, threads have:
 - **States** and **transitions** (new, running, blocked, ready, terminated)
 - A **thread control block (TCB)**

Threads

Threads from an OS Perspective (Cont'ed)

Processes	Threads
Address space	Program Counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	Local vars
Signals and signal handlers	
Accounting information	

Table: Shared resources left, private resources right

Threads

Threads from an OS Perspective (Cont'ed)

- Threads incur **less overhead** to create/terminate/switch (address space remains the same for threads of the same process)
- Some CPUs have direct **hardware support** for **multi-threading**
 - Typically, they can offer up to 8 hardware threads per core

Threads

Threads from an OS Perspective (Cont'ed)

- **Inter-thread communication** is easier/faster than **interprocess** communication (threads share memory by default)
- **No protection boundaries** are required in the address space (threads are cooperating, belong to the same user, and have a common goal)
- **Synchronisation** has to be considered carefully!

Threads

Why Use Threads

- Multiple **related activities** apply to the **same resources**, these resources should be accessible/shared
- Processes will often contain multiple **blocking tasks**
 - I/O operations (thread blocks, **interrupt** marks completion)
 - Memory access: pages faults are result in blocking
- Such activities should be carried out in **parallel/concurrently**
- **Application examples:** webservers, make program, spreadsheets, word processors, processing large data volumes

Threads

OS Implementations of Threads

- **User** threads
- **Kernel** threads
- **Hybrid** implementations

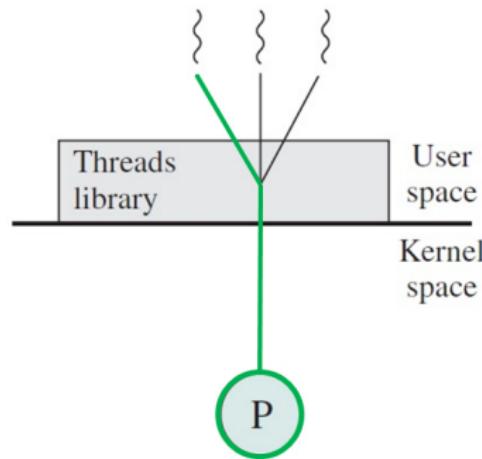
User Threads

Many-to-One

- **Thread management** (creating, destroying, scheduling, thread control block manipulation) is carried out **in user space** with the help of a **user library**
- The process maintains a **thread table** managed by the **runtime system** without the **kernel's knowledge**
 - Similar to **process table**
 - Used for **thread switching**
 - Tracks thread related information

User Threads

Many-to-One

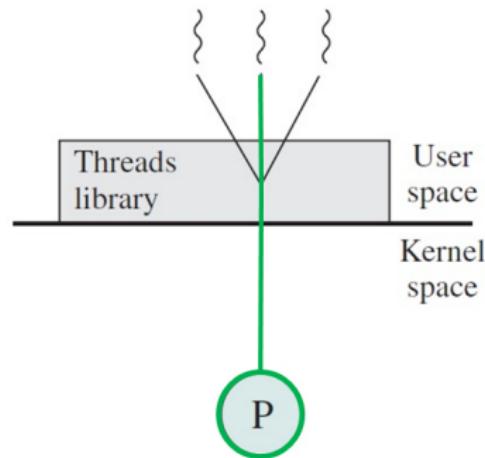


(a) Pure user-level

Figure: User threads (Stallings)

User Threads

Many-to-One

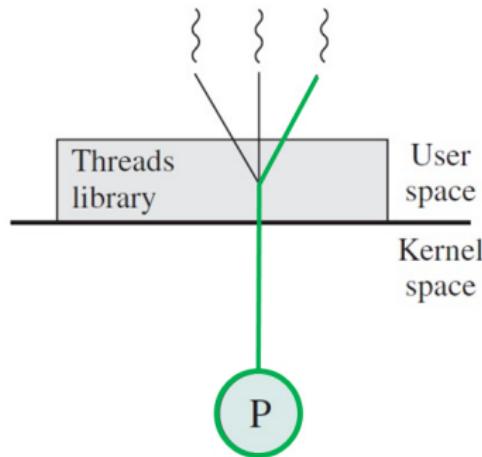


(a) Pure user-level

Figure: User threads (Stallings)

User Threads

Many-to-One

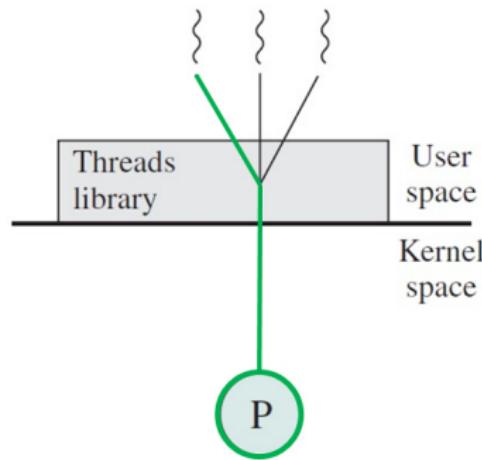


(a) Pure user-level

Figure: User threads (Stallings)

User Threads

Many-to-One

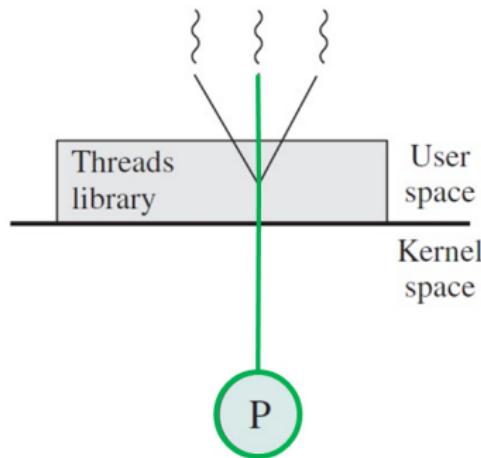


(a) Pure user-level

Figure: User threads (Stallings)

User Threads

Many-to-One

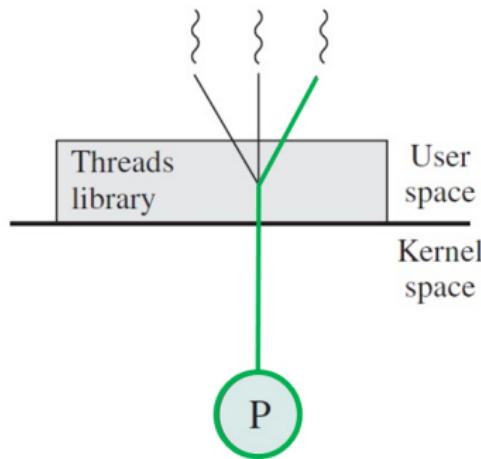


(a) Pure user-level

Figure: User threads (Stallings)

User Threads

Many-to-One

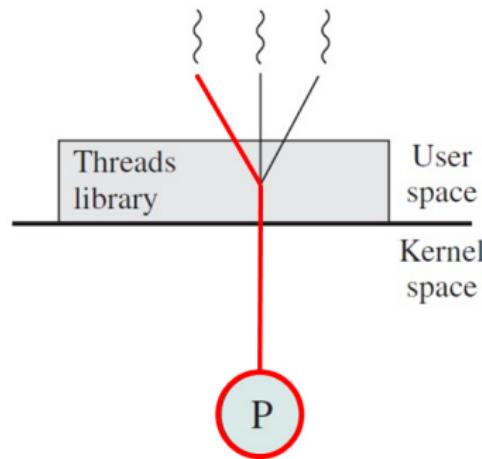


(a) Pure user-level

Figure: User threads (Stallings)

User Threads

Many-to-One



(a) Pure user-level

Figure: User threads (Stallings)

User Threads

Many-to-One

- Advantages:
 - Threads are in user space (i.e., **no mode switches** required)
 - **Full control** over the thread scheduler
 - **OS independent** (threads can run on OS that do not support them)
- Disadvantages:
 - **Blocking system calls** suspend the entire process (user threads are mapped onto a single process, managed by the kernel)
 - **No true parallelism** (a process is scheduled on a single CPU)
 - **Clock interrupts** are non-existent (i.e. user threads are non-preemptive)
 - **Page faults** result in blocking the process

User Threads

Many-to-One

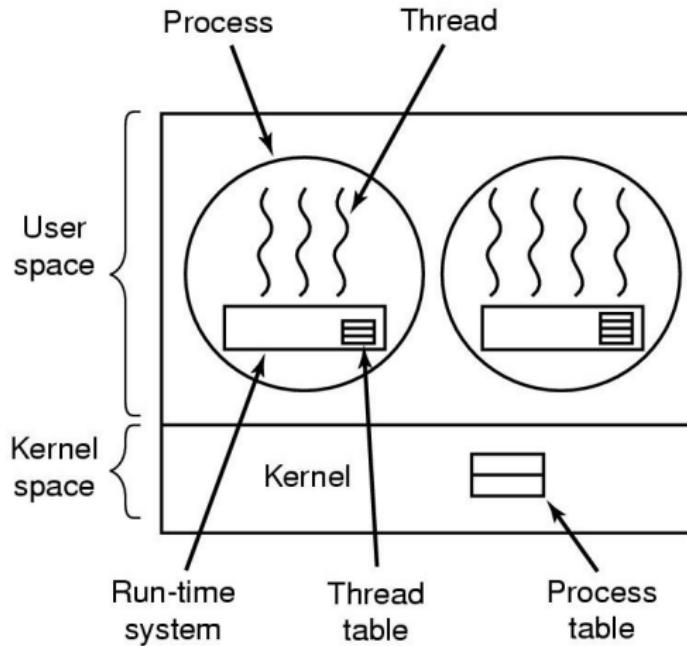


Figure: User threads (Tanenbaum 2014)

Kernel Threads

One-to-One

- The **kernel manages** the threads, user application accesses threading facilities through **API** and **system calls**
 - **Thread table** is in the kernel, containing thread control blocks (subset of process control blocks)
 - If a **thread blocks**, the kernel chooses thread from same or different process
- Advantages:
 - **True parallelism** can be achieved
 - No run-time system needed
- Frequent **mode switches** take place, resulting in lower performance
- Windows and Linux apply this approach

Kernel Threads

One-to-One

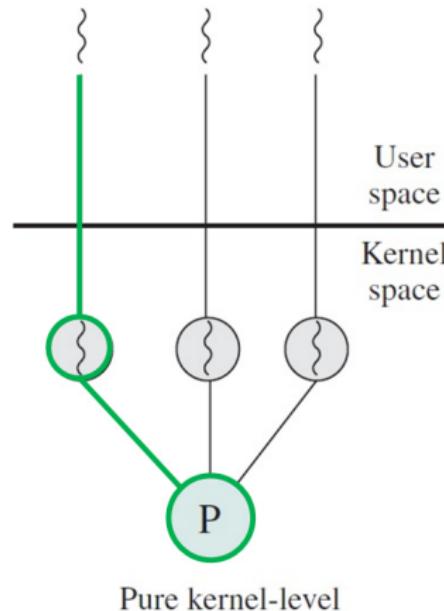


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

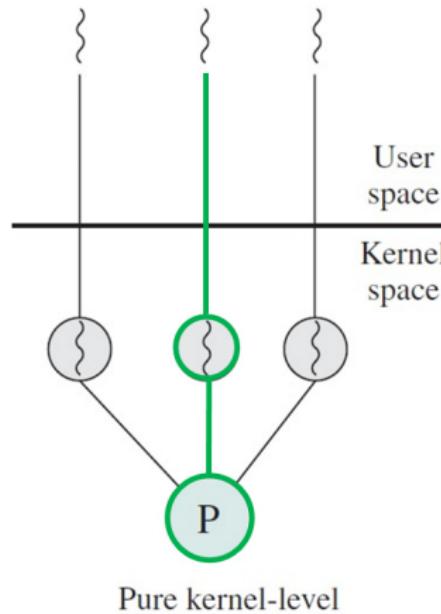


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

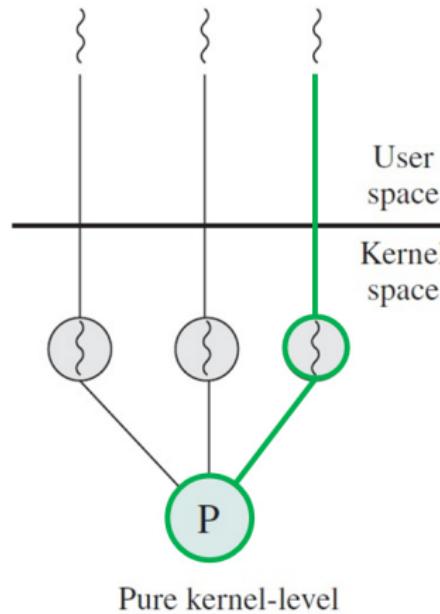


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

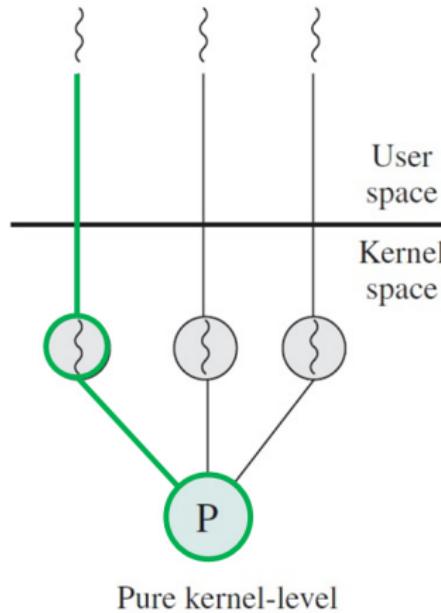


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

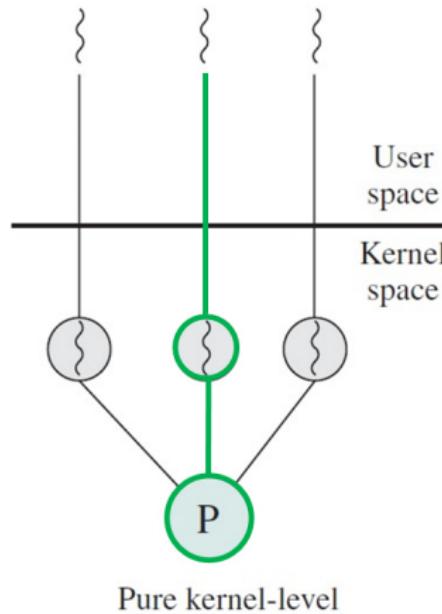


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

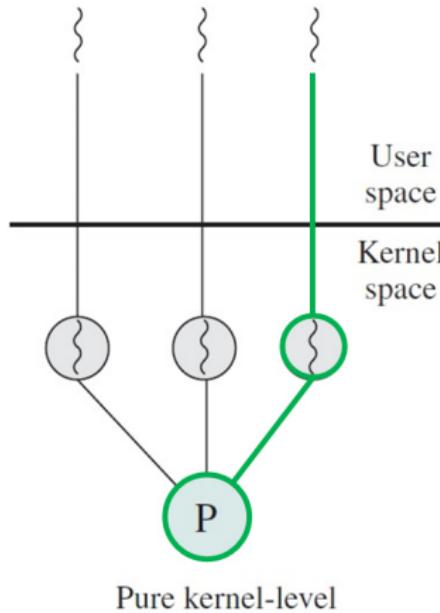


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

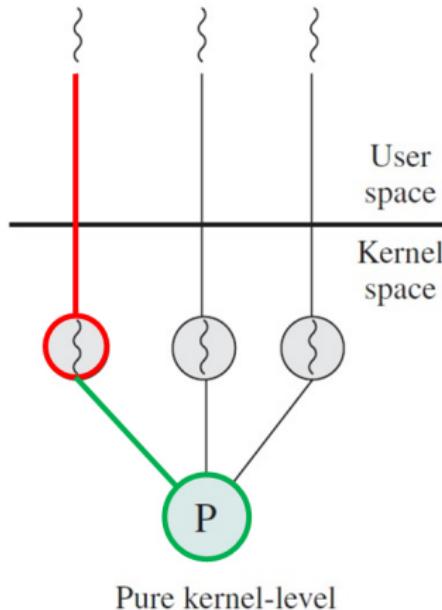


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

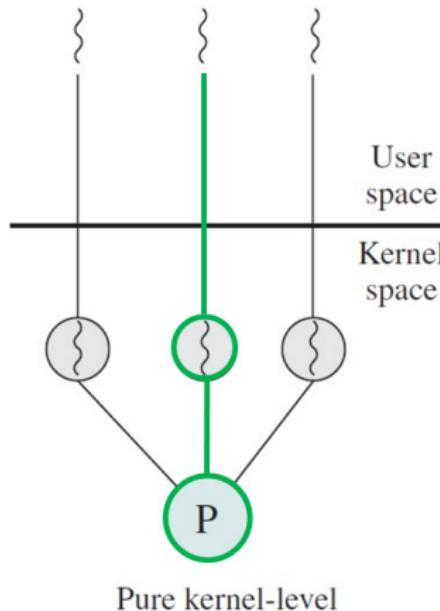


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

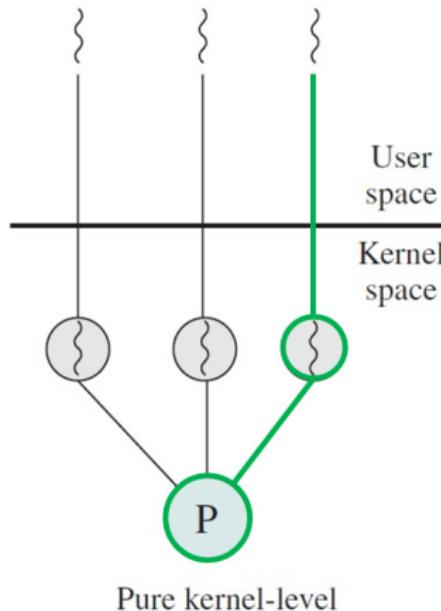


Figure: Kernel threads (Stallings 2014)

Kernel Threads

One-to-One

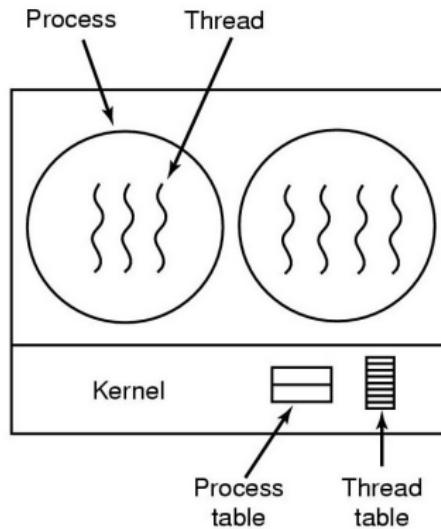


Figure: Kernel threads (Tanenbaum 2014)

Performance

User Threads vs. Kernel Threads vs. Processes

- Null fork: the overhead in creating, scheduling, running and terminating a null process/thread
- Signal wait: overhead in synchronising threads

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Figure: Comparison, in μs (Stallings)

Hybrid Implementations

Many-to-Many

- User threads are **multiplexed** onto kernel threads
- Kernel sees and schedules the kernel threads (a limited number)
- User application sees user threads and creates/schedules these (an “unrestricted” number)

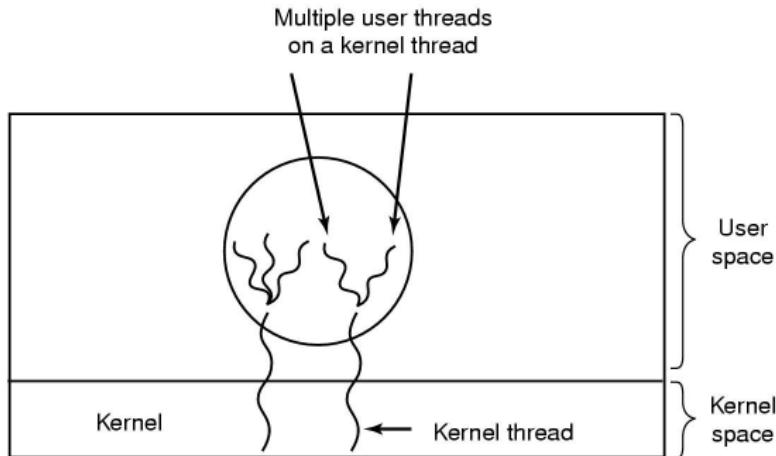


Figure: Kernel threads (Tanenbaum 2014)

Comparison

Thread Implementations

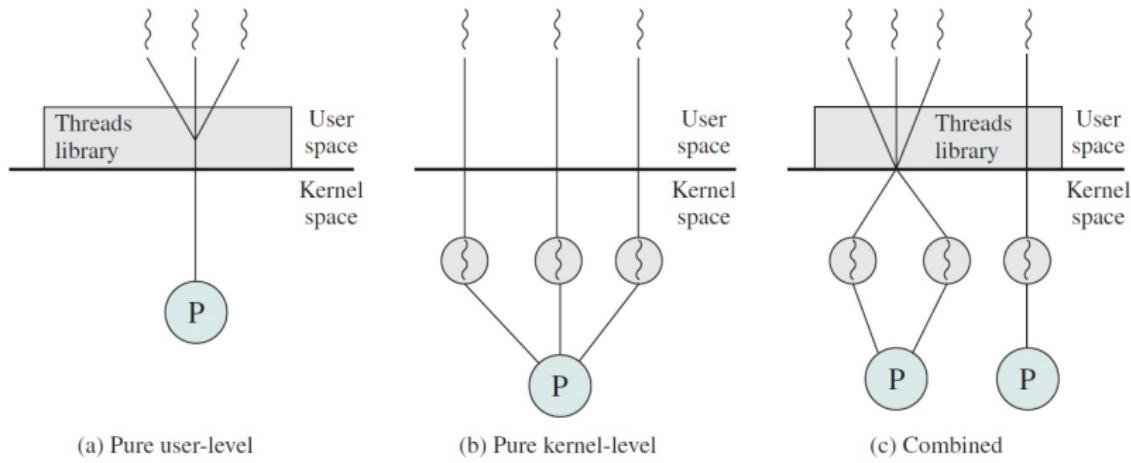


Figure: Comparison (Stallings)

Exam 2013-2014: In which situations would you favour user level threads? In which situation would you definitely favour kernel level threads?

Thread Management

Libraries

- Thread libraries provide an **API/interface for managing threads** (e.g. creating, running, destroying, synchronising, etc.)
- Thread libraries can be implemented:
 - Entirely in **user space** (i.e. user threads)
 - Based on **system calls**, i.e., rely on the kernel for thread implementations
- Examples of thread APIs include **POSIX's PThreads**, Windows Threads, and Java Threads
 - The PThread specification can be implemented as user or kernel threads

POSIX Threads

Overview

- POSIX threads are a **specification** that “**anyone**” can implement, i.e., it defines a set of APIs (function calls, over 60 of them) and what they do
- Core functions of PThreads include:

Function Call	Summary
<code>pthread_create</code>	Create new thread
<code>pthread_exit</code>	Exit existing thread
<code>pthread_join</code>	Wait for thread with ID
<code>pthread_yield</code>	Release CPU
<code>pthread_attr_init</code>	Thread Attributes (e.g. priority)
<code>pthread_attr_destroy</code>	Release Attributes

Table: PThread examples

- More detailed descriptions can be found using `man function_name` on the command line

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

POSIX Threads

Example

```
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
    printf("Hello from thread %d\n", *((int*)arg));
    return 0;
}

int main() {
    int const THREADS = 10;
    int args[THREADS] = { 0 };
    pthread_t threads[THREADS];
    for(int i = 0; i < THREADS; i++) {
        args[i] = i;
        if(pthread_create(threads + i, NULL, hello, args + i)) {
            printf("Creating thread %d failed\n", i); return -1;
        }
    }
    for(int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);
}
```

Test your understanding

- If the threads in a process share the same memory, why do they have independent stacks?
- Is it always necessary to call `pthread_exit` when ending a thread?
- What is the minimum number of threads a process can have?
- Can user threads make good use of concurrent hardware?

Operating Systems and Concurrency

Processes 4: Further Scheduling
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

- ① Threads are an abstraction of execution traces.
- ② Threads vs. processes.
- ③ Thread implementations - user, kernel and hybrid.
- ④ PThreads.

Goals for Today

Overview

- **Multi-level feedback queues.**
- Scheduling in **Windows 7**.
- Scheduling in **Linux**.
- **Load balancing.**
- Scheduling **related processes/threads**.

Priority Queues

Recall

- Jobs can have **different priority levels**.
- Jobs of the **same priority** are run in **round robin** fashion.
- Usually implemented by using **multiple queues**, one for each priority level.

Multi-level Feedback Queues

Moving Beyond Priority Queues

- Different **scheduling algorithms** can be used for the **individual queues** (e.g., round robin, SJF, FCFS)
- **Feedback queues** allow **priorities to change dynamically**. Jobs can **move between queues**:
 - Move to **lower priority queue** if too much CPU time is used (prioritise I/O and interactive processes)
 - Move to **higher priority queue** to prevent **starvation** and avoid **inversion of control**

Inversion of Control

Illustration

Process A (low)	Process B (high)	Process C (medium)
...		
request X		
receive X		
...	RUN	
...	request X	
...	blocked	RUN
...

Multi-level Feedback Queues

Moving Beyond Priority Queues

- Defining characteristics of feedback queues include:
 - The **number of queues**
 - The **scheduling algorithms** used for the individual queues
 - **Migration policy** between queues
 - Initial **access** to the queues
- Feedback queues are highly **configurable** and offer significant flexibility

Multi-level Feedback Queues

Windows 7

- An **interactive system** using a **preemptive scheduler** with **dynamic priority levels**
 - Two **priority classes** with **16 different priority levels** exist
 - “Real time” processes/threads have a **fixed priority level**
 - “Variable” processes/threads can have their priorities **boosted temporarily**
- A **round robin algorithm** is used within the queues

Multi-level Feedback Queues

Windows 7 (Cont'd)

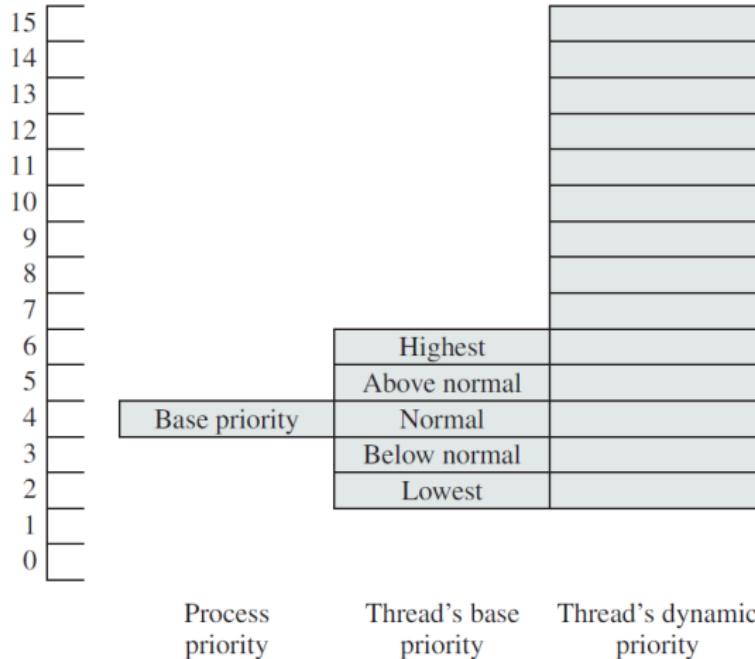


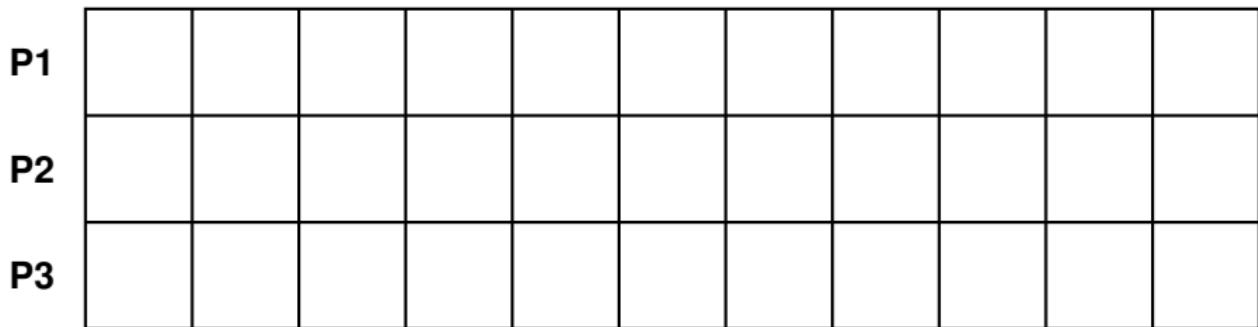
Figure: Priorities in Windows 7 (Stallings, 7th edition)

Multi-level Feedback Queues

Windows 7 (Cont'ed)

- Priorities are based on the **process base priority** (between 0-15) and **thread base priority** (± 2 relative to the process priority)
- A thread's **priority dynamically changes** during execution between its base priority and the maximum priority within its class
 - **Interactive I/O bound processes** (e.g. keyboard) receive a **larger boost**
 - Boosting priorities prevents **starvation** and **priority inversion**

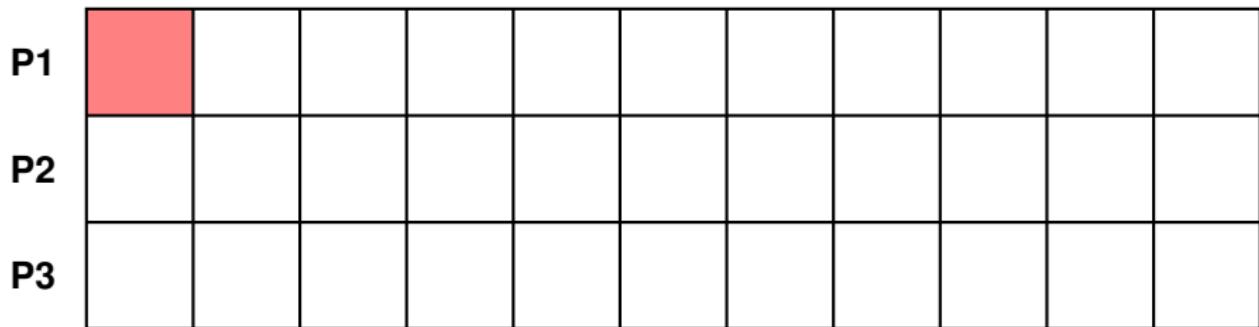
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

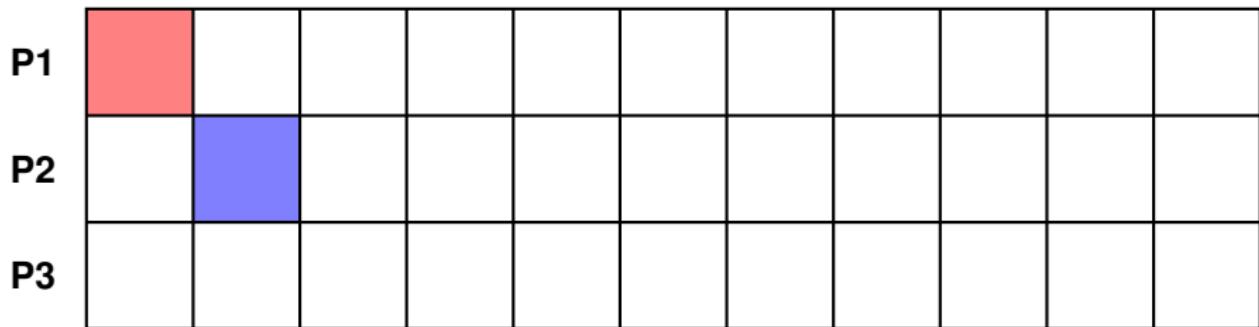
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

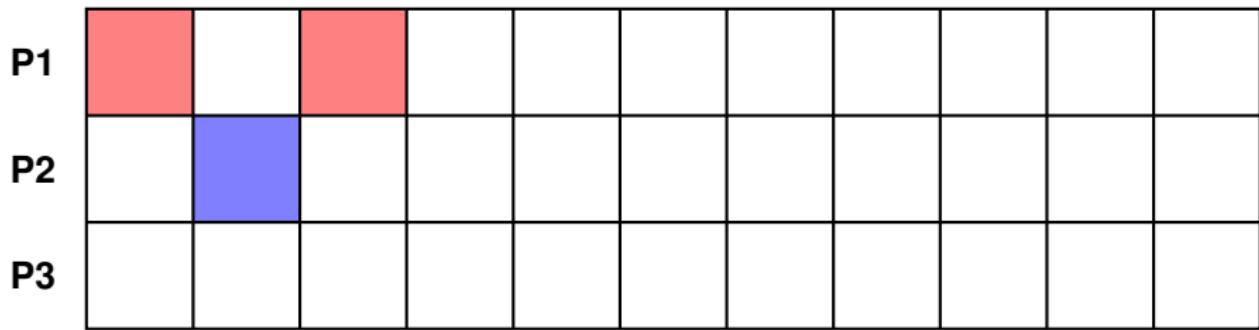
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

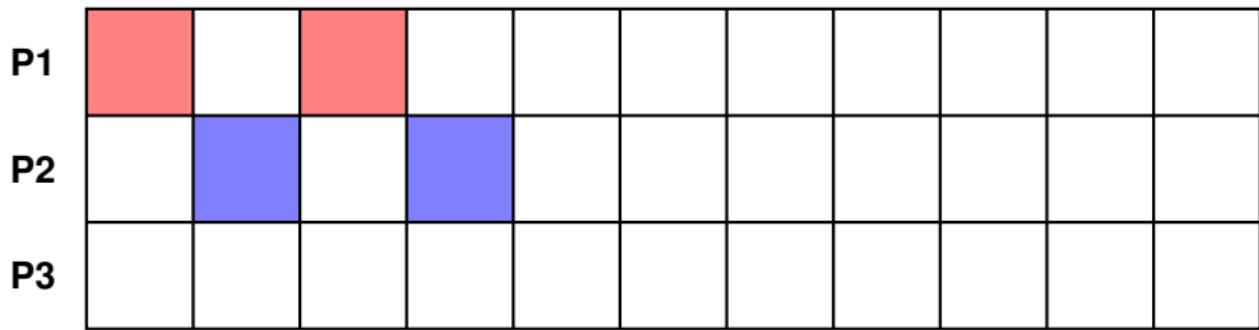
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

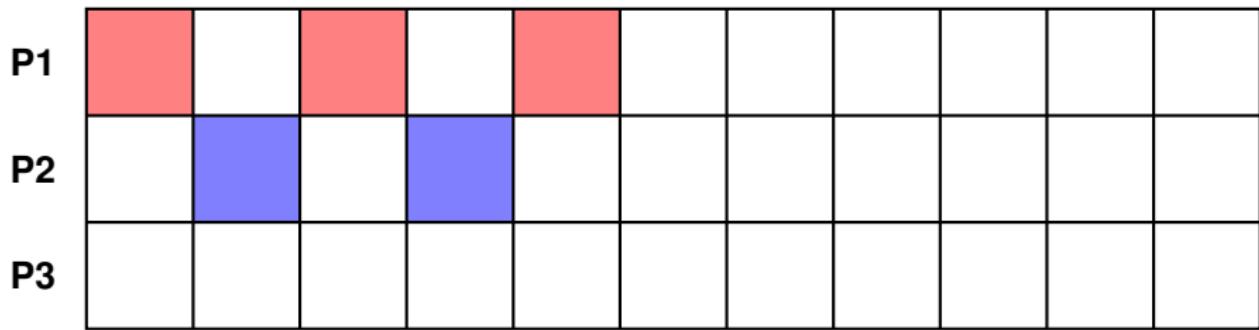
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

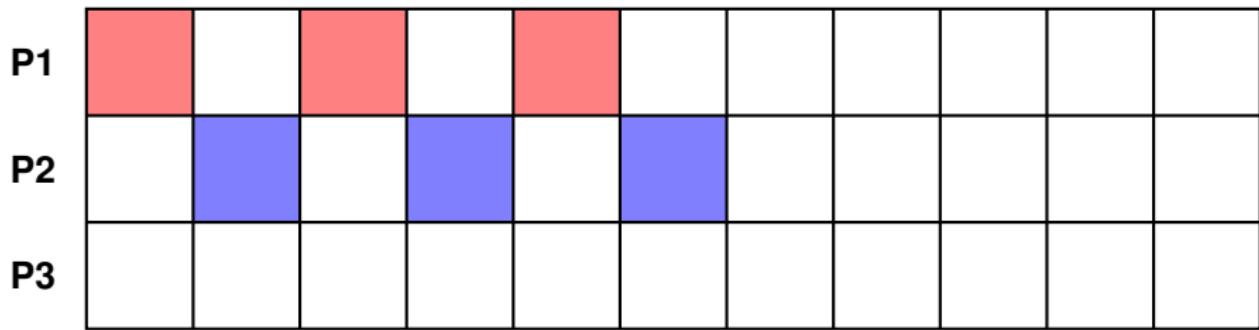
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

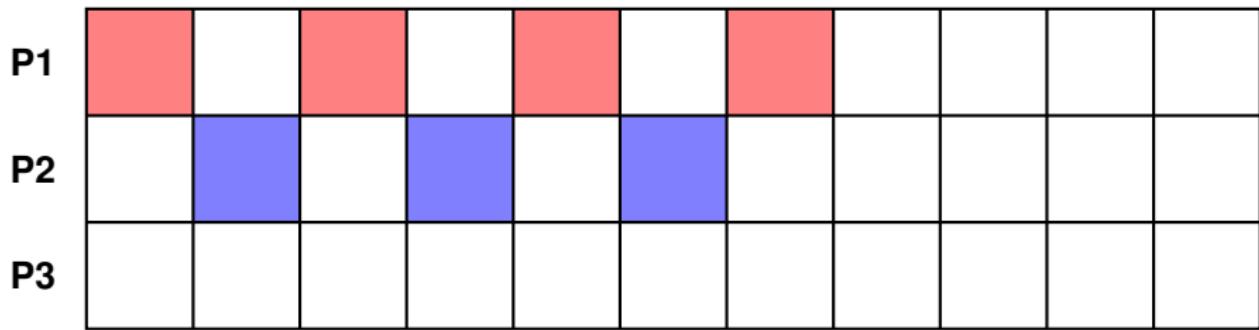
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

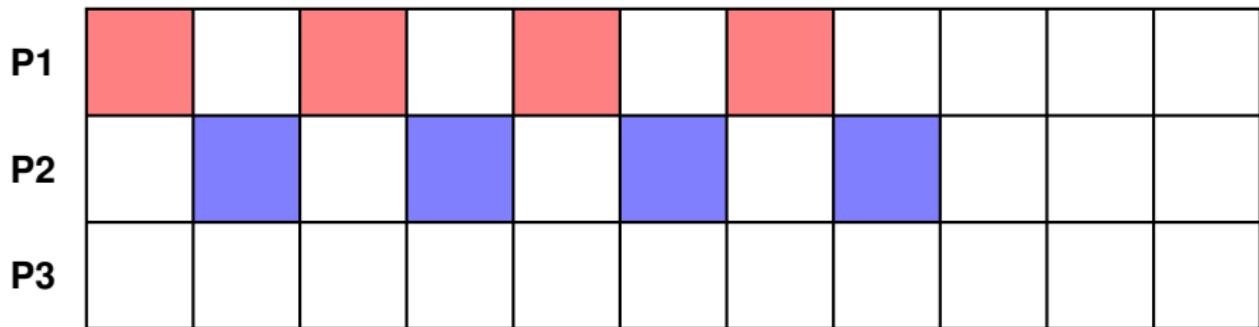
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

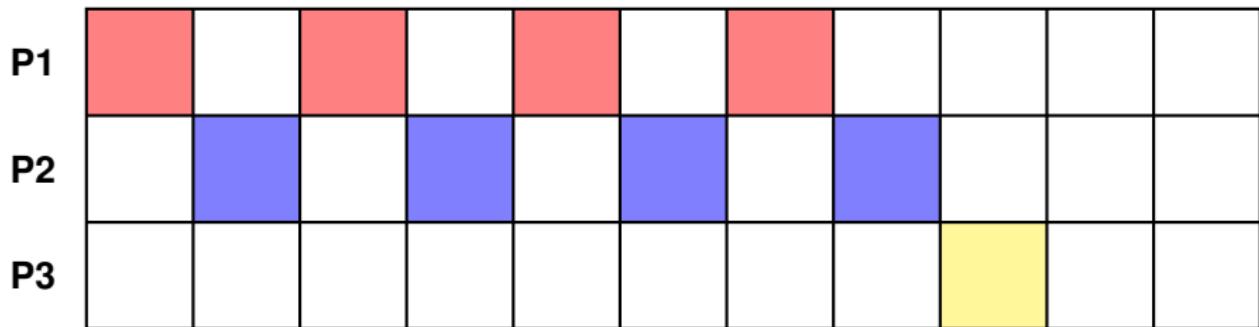
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).
- The scheduler concludes P3 is being starved of CPU time.

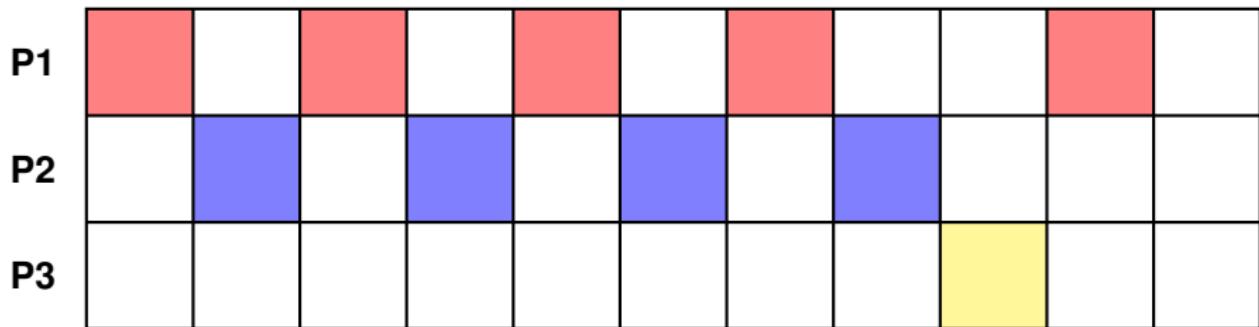
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).
- The scheduler concludes P3 is being starved of CPU time.
- Process P3 has its priority temporarily promoted to prevent starvation.

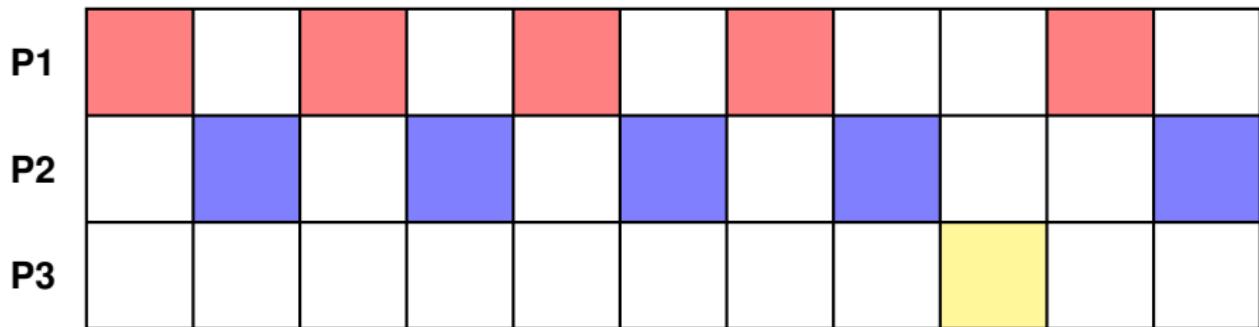
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).
- The scheduler concludes P3 is being starved of CPU time.
- Process P3 has its priority temporarily promoted to prevent starvation.
- Computation continues as before.

Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).
- The scheduler concludes P3 is being starved of CPU time.
- Process P3 has its priority temporarily promoted to prevent starvation.
- Computation continues as before.

Scheduling in Linux

The Completely Fair Scheduler

- Process scheduling has **evolved** over different versions of Linux to make efficient use of **multiple processors/cores**
- Linux distinguishes between two types of tasks for scheduling:
 - **Real time tasks** (to be POSIX compliant), divided into:
 - Real time FIFO tasks
 - Real time Round Robin tasks
 - **Time sharing tasks** using a **preemptive** approach which are similar to **variable** in Windows.
- The most recent scheduling algorithm in Linux for **time sharing tasks** is the **completely fair scheduler (CFS)**.

Scheduling in Linux

Real-Time Tasks

- Real time **FIFO** tasks have the **highest priority** and are scheduled using a **FCFS approach**, using **preemption if a higher priority job shows up**
- Real time **round robin tasks** are preemptable by **clock interrupts** and have a **time slice** associated with them
- Both approaches **cannot guarantee hard deadlines**

Scheduling on Linux

Time Sharing Tasks - The Ideal Situation

The Ideal Fair Scheduler

We imagine a hypothetical ideal scenario:

- Our CPU allows all N current tasks to be run simultaneously, with each receiving $\frac{1}{N}$ of the CPU power.
- For example, with 5 tasks wanting to run, each gets 20% of the available computational power.
- Unfortunately real CPUs cannot run an arbitrary number of tasks in parallel in this way - but can we approximate this ideal?

Scheduling on Linux

Time Sharing Tasks - The CFS

Deciding how to divide up the CPU time

- We choose a **target latency** - this is the amount of time before every task gets access to the CPU. The target latency also bounds how far we will drift from being fair.
- To hit this target latency, for N tasks, each task is allowed to run for $\frac{1}{N}$ of the target latency.
- To avoid excessive context switching when N is large, we also choose a **minimum granularity** - a minimum amount of time we will allow a task to run on the CPU before being considered for replacement.

Scheduling in Linux

Time Sharing Tasks - the CFS

Approximating Fairness

- We record a **virtual time** that each task has had on the CPU, and order tasks by their virtual CPU time.
- Tasks are ordered in **ascending order of virtual time used** - implemented using a red-black tree.
- The task with the lowest virtual time on the CPU is considered to have been **treated least fairly**, and will be the next one chosen to run on the CPU.
- After that task has had $\frac{1}{N}$ of the target latency in virtual time, we replace it with the next task with lowest virtual run time.
- Note - system calls may lead to a task using less than its full allocated time. **This will mean they will get back on the CPU more quickly.**

Scheduling in Linux

Time Sharing Tasks - the CFS

Accounting for priorities

- A **weighting scheme** is used to take different priorities into account - we will assume that the weight is literally the task priority - a simplification!
- The recorded virtual time on the CPU is the real time on the CPU scaled up by the weight. After 100ms of actual computation time:
 - A priority 1 (higher priority) process is considered to have used 100ms of virtual time.
 - A priority 2 (lower priority) process is considered to have used 200ms of virtual time.
- **Virtual time runs at different speeds for different priority processes!**
- Note that tasks will be given varying windows of time to run - unlike traditional time slicing.

Scheduling in Linux

Time Sharing Tasks - Example

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

- 1 CPU: $T1(0,0)$, queue: $T2(0,0), T3(0,0)$

Scheduling in Linux

Time Sharing Tasks - Example

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

- ① CPU: $T1(0,0)$, queue: $T2(0,0), T3(0,0)$
- ② CPU: $T2(0,0)$, queue: $T3(0,0), T1(100,100)$

Scheduling in Linux

Time Sharing Tasks - Example

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

- ① CPU: $T1(0,0)$, queue: $T2(0,0), T3(0,0)$
- ② CPU: $T2(0,0)$, queue: $T3(0,0), T1(100,100)$
- ③ CPU: $T3(0,0)$, queue: $T1(100,100), T2(100,50)$

Scheduling in Linux

Time Sharing Tasks - Example

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

- ① CPU: $T1(0,0)$, queue: $T2(0,0), T3(0,0)$
- ② CPU: $T2(0,0)$, queue: $T3(0,0), T1(100,100)$
- ③ CPU: $T3(0,0)$, queue: $T1(100,100), T2(100,50)$
- ④ CPU: $T1(100,100)$, queue $T2(100,50), T3(100,33)$

Scheduling in Linux

Time Sharing Tasks - Example

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

- ① CPU: $T1(0,0)$, queue: $T2(0,0), T3(0,0)$
- ② CPU: $T2(0,0)$, queue: $T3(0,0), T1(100,100)$
- ③ CPU: $T3(0,0)$, queue: $T1(100,100), T2(100,50)$
- ④ CPU: $T1(100,100)$, queue $T2(100,50), T3(100,33)$
- ⑤ CPU: $T2(100,50)$, queue $T3(100,33), T1(200,200)$

Scheduling in Linux

Time Sharing Tasks - Example

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

- ① CPU: $T1(0,0)$, queue: $T2(0,0), T3(0,0)$
- ② CPU: $T2(0,0)$, queue: $T3(0,0), T1(100,100)$
- ③ CPU: $T3(0,0)$, queue: $T1(100,100), T2(100,50)$
- ④ CPU: $T1(100,100)$, queue $T2(100,50), T3(100,33)$
- ⑤ CPU: $T2(100,50)$, queue $T3(100,33), T1(200,200)$
- ⑥ ...

Scheduling in Linux

Time Sharing Tasks - Example

Further enhancements

To avoid potential pathological behaviours:

- New tasks have their virtual run time set to the **current minimum virtual run time** - Think about how unfairly advantaged they might be if this was set to zero!

Scheduling in Linux

Time Sharing Tasks - Example

Further enhancements

To avoid potential pathological behaviours:

- New tasks have their virtual run time set to the **current minimum virtual run time** - Think about how unfairly advantaged they might be if this was set to zero!
- Blocked tasks have their virtual run time set to the greater of:
 - The **current minimum virtual run time, minus a small offset** - to ensure it gets to run.
 - Its **old virtual run time** - in this case it is already getting a good share of the CPU.
- Think about how long a task that was blocked for a long time might get on the CPU otherwise!

Multi-processor Scheduling

Scheduling Decisions

- **Single processor** machine: **which thread** to run next?
- Scheduling decisions on a **multi-core** machine include:
 - Which thread to run **when**?
 - Which thread to run **where**?

Multi-processor Scheduling

Shared Queues

- A single or multi-level queue **shared** between all CPUs
- Advantage: automatic **load balancing**
- Disadvantages:
 - **Contention** for the queues.
 - Does not take advantage of the current state of the CPU's
 - **Cache** becomes invalid when moving to a different CPU
 - Translation look aside buffers (**TLBs** - part of the MMU) become invalid

Multi-processor Scheduling

Private Queues

- Each CPU has a **private queue or queues**.
- Advantages:
 - Often can reuse existing CPU state such as cache and TLB
 - **Contention** for shared queue is minimised
- Disadvantages: less **load balancing**
- To mitigate the lack of load balancing **migration** between CPUs is possible

Related vs. Unrelated Threads

Thread Types

- **Related:** multiple threads that communicate with one another and ideally run together (e.g. search algorithm)
- **Unrelated:** e.g. processes threads that are independent, possibly started by different users running different programs

Scheduling Related Threads

Working Together

- E.g., threads belong to the same process and are **cooperating**, e.g. they **exchange messages** or **share information**, e.g.
 - Process A has thread A_0 and A_1 , A_0 and A_1 cooperate
 - Process B has thread B_0 and B_1 , B_0 and B_1 cooperate
 - The scheduler selects A_0 and B_1 to run first, then A_1 and B_0 , and A_0 and A_1 , and B_0 and B_1 run on different CPUs
 - They try to send messages to the other threads, which are still in the ready state

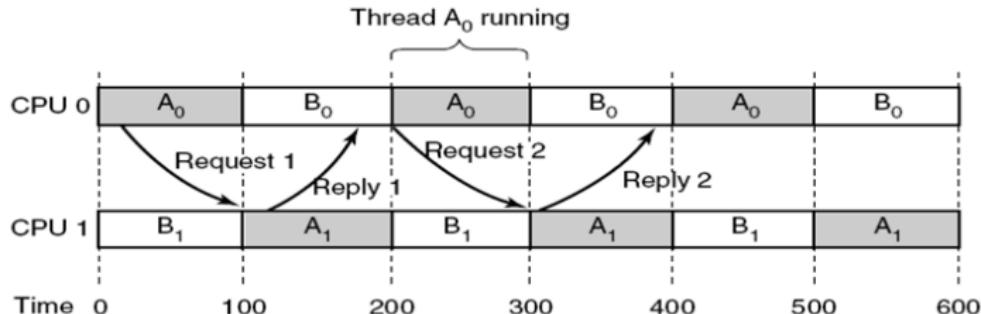


Figure: Tanenbaum, Chapter 8

Scheduling Related Threads

Working Together

- The aim is to get **collaborating threads** running, as much as possible, at the **same time** across **multiple CPUs**
- Approaches include:
 - **Space** sharing
 - **Gang** scheduling

Scheduling Related Threads

Space sharing

- Approach known as **space sharing scheduling**:
 - N related threads, typically from a single process, are allocated to N **dedicated CPUs** when enough CPUs are available.
 - M related threads, typically from another process, are **kept waiting until M CPUs are available**.
 - At any point in time the available CPUs are partitioned into blocks of related threads.
 - As thread complete, their dedicated CPUs are returned to the collection of available CPUs.
 - The CPUs are **not multiprogrammed** to keep related threads running together. This means blocking calls result in **idle CPUs**.

Scheduling Related Threads

Gang scheduling

- Space sharing scheduling shares work by space (CPU)
 - Keeps related threads running together.
 - Lack of multiprogramming avoids context switching overhead, but leads to wasted CPU cycles.
 - **Gang scheduling** is an attempt to schedule “in both time and space” to avoid this waste of CPU time.

Scheduling Related Threads

Gang scheduling

- The scheduler **groups related threads** together into **gangs** to run simultaneously on different CPUs.
- This is a **preemptive** algorithm, with time slices synchronised across all CPUs.
- Blocking threads** result in idle CPUs
 - If a thread blocks the rest of the time slice will be unused, due to the time slice synchronisation across all CPUs.

		CPU					
		0	1	2	3	4	5
Time slot		A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
		B ₀	B ₁	B ₂	C ₀	C ₁	C ₂
0	D ₀	D ₁	D ₂	D ₃	D ₄	E ₀	
1	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	
2	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅	
3	B ₀	B ₁	B ₂	C ₀	C ₁	C ₂	
4	D ₀	D ₁	D ₂	D ₃	D ₄	E ₀	
5	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆	

Test your understanding

- Why would boosting thread priorities prevent priority inversion?
- Why is it efficient to schedule threads that communicate with each other at the same time?
- How does the CFS avoid starvation?

Operating Systems and Concurrency

Concurrency 1
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Concurrency

Context

- Threads and processes **execute concurrently** or **in parallel** and can **share resources** (e.g., devices, memory – variables and data structures)
 - Multi-programming/multi-processing **improves system utilisation**
- A thread can be **interrupted at any point in time** (timer, I/O)
 - The process state is **saved** in the **process control block**
- The outcome of programs may become **unpredictable**:
 - Sharing data can lead to **inconsistencies** - we can be interrupted part way through doing something.
 - The **outcome of execution** may **depend on the order** in which code gets to run on the CPU.

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

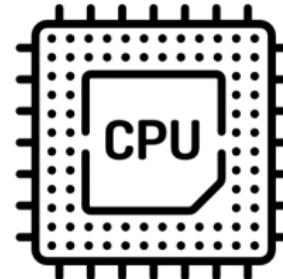
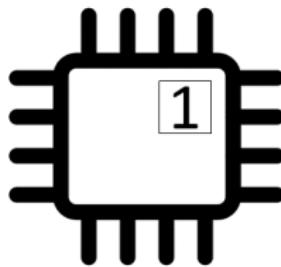
int counter = 0;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++)
        counter++;
    return 0;
}

int main() {
    pthread_t tid1 = 0,tid2 = 0;
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

- `counter++` consists of three separate actions:
 - ➊ read the value of counter from memory and **store it in a register**
 - ➋ add one to the value in the register
 - ➌ store the value of the register **in counter** in memory
- The above actions are **NOT** “atomic”¹

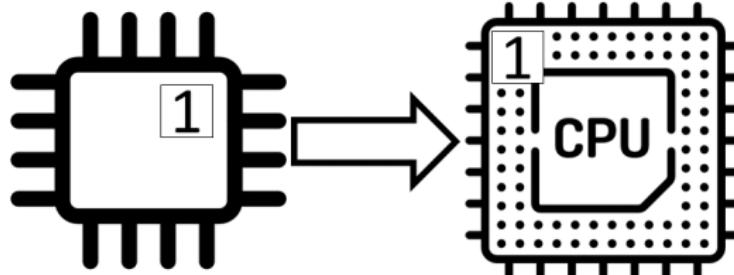


¹Icons from <https://www.flaticon.com/>
©University of Nottingham

Example

Incrementing a counter

- `counter++` consists of three separate actions:
 - ➊ read the value of counter from memory and **store it in a register**
 - ➋ add one to the value in the register
 - ➌ store the value of the register **in counter** in memory
- The above actions are **NOT** “atomic”¹

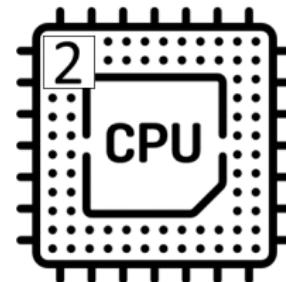
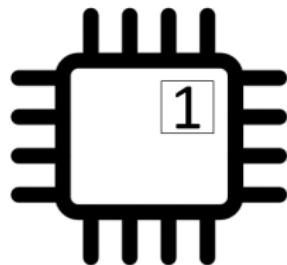


¹Icons from <https://www.flaticon.com/>
©University of Nottingham

Example

Incrementing a counter

- `counter++` consists of three separate actions:
 - ➊ read the value of counter from memory and **store it in a register**
 - ➋ add one to the value in the register
 - ➌ store the value of the register **in counter** in memory
- The above actions are **NOT** “atomic”¹



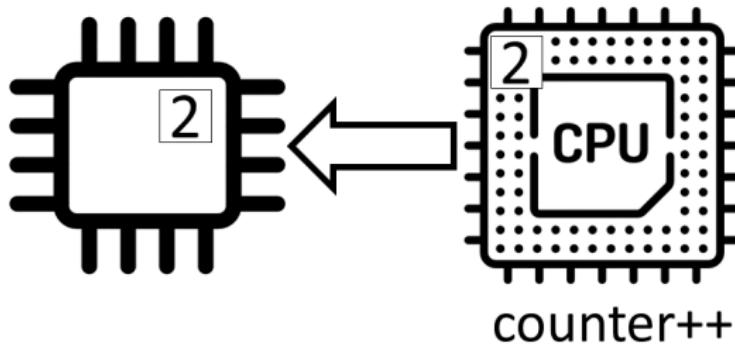
counter++

¹Icons from <https://www.flaticon.com/>

Example

Incrementing a counter

- `counter++` consists of three separate actions:
 - ➊ read the value of counter from memory and **store it in a register**
 - ➋ add one to the value in the register
 - ➌ store the value of the register **in counter** in memory
- The above actions are **NOT** “atomic”¹



¹Icons from <https://www.flaticon.com/>

Example

Incrementing a counter

Thread 1:

...

Read counter -> register (= 1)

Add 1 to register value (= 2)

Store register in counter (= 2)

...

...

...

Thread 2:

...

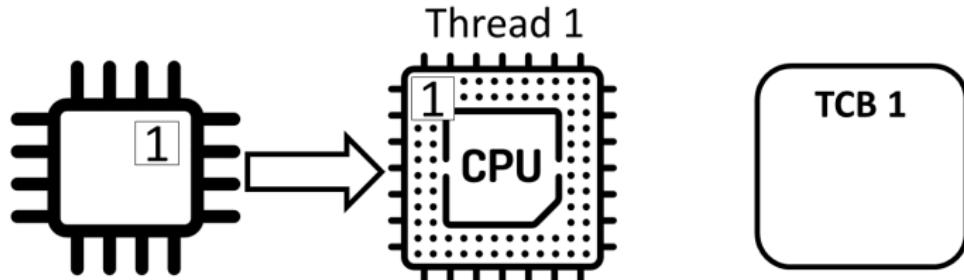
...

...

Read counter -> register (= 2)

Add 1 to register value (= 3)

Store register in counter (= 3)



Example

Incrementing a counter

Thread 1:

...
Read counter -> register (= 1)

Add 1 to register value (= 2)

Store register in counter (= 2)

...

...

...

Thread 2:

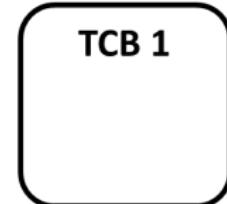
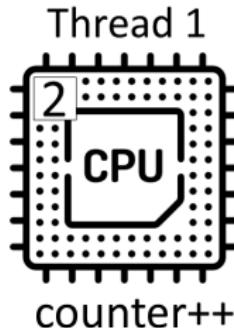
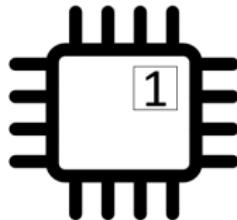
...
...

...

Read counter -> register (= 2)

Add 1 to register value (= 3)

Store register in counter (= 3)



Example

Incrementing a counter

Thread 1:

...
Read counter -> register (= 1)

Add 1 to register value (= 2)

Store register in counter (= 2)

...

...

...

Thread 2:

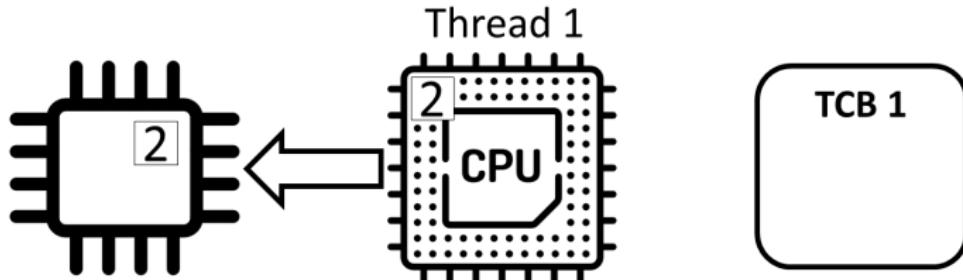
...
...

...

Read counter -> register (= 2)

Add 1 to register value (= 3)

Store register in counter (= 3)



Example

Incrementing a counter

Thread 1:

...
Read counter -> register (= 1)

Add 1 to register value (= 2)

Store register in counter (= 2)

...

...

...

Thread 2:

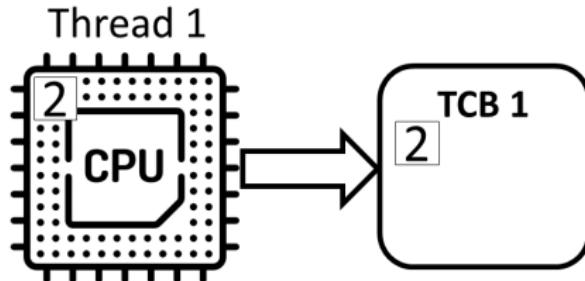
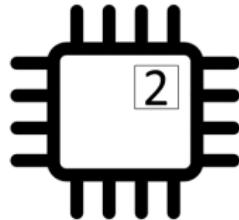
...
...

...

Read counter -> register (= 2)

Add 1 to register value (= 3)

Store register in counter (= 3)



Example

Incrementing a counter

Thread 1:

...
Read counter -> register (= 1)

Add 1 to register value (= 2)

Store register in counter (= 2)

...

...

...

Thread 2:

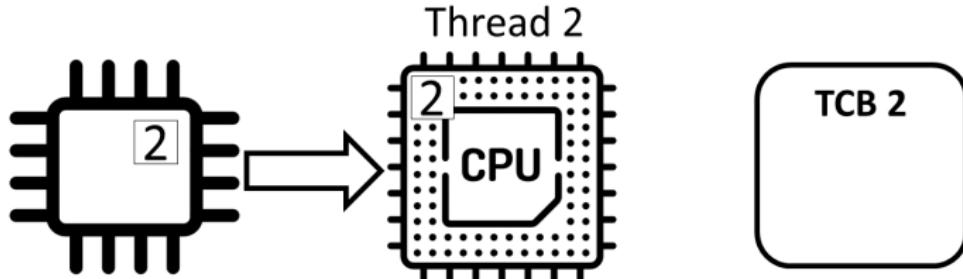
...
...

...

Read counter -> register (= 2)

Add 1 to register value (= 3)

Store register in counter (= 3)



Example

Incrementing a counter

Thread 1:

...
Read counter -> register (= 1)

Add 1 to register value (= 2)

Store register in counter (= 2)

...

...

...

Thread 2:

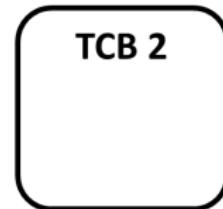
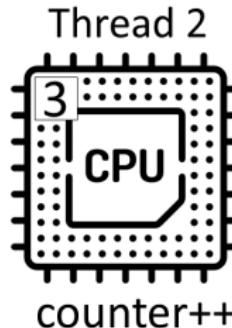
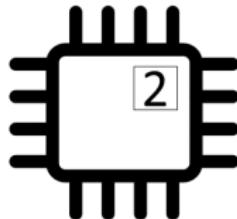
...
...

...

Read counter -> register (= 2)

Add 1 to register value (= 3)

Store register in counter (= 3)



Example

Incrementing a counter

Thread 1:

...
Read counter -> register (= 1)

Add 1 to register value (= 2)

Store register in counter (= 2)

...

...

...

Thread 2:

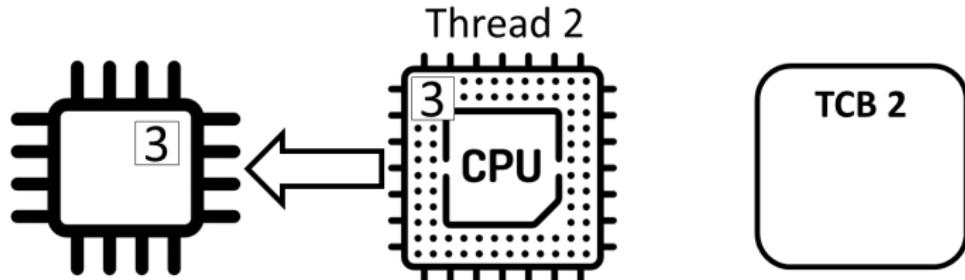
...
...

...

Read counter -> register (= 2)

Add 1 to register value (= 3)

Store register in counter (= 3)



Example

Incrementing a counter

Thread 1:

...
Read counter -> register (= 1)

Add 1 to register value (= 2)

Store register in counter (= 2)

...

...

...

Thread 2:

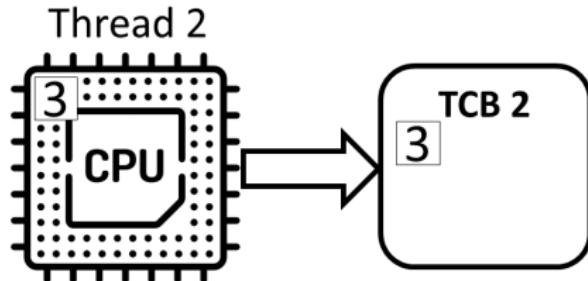
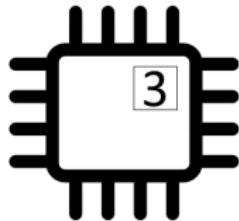
...
...

...

Read counter -> register (= 2)

Add 1 to register value (= 3)

Store register in counter (= 3)



Example

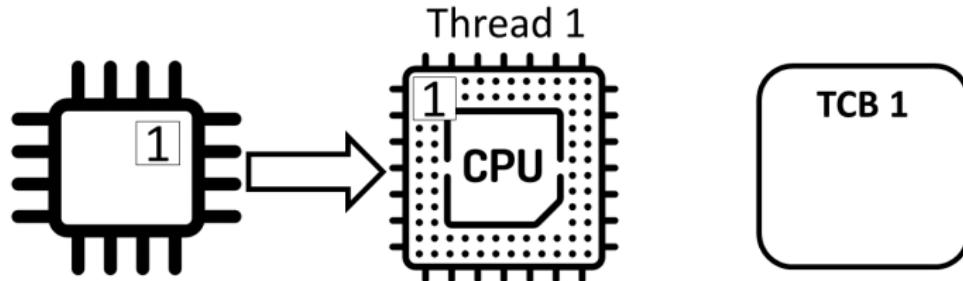
Incrementing a counter

Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

```
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

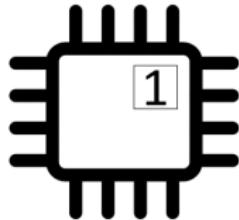


Example

Incrementing a counter

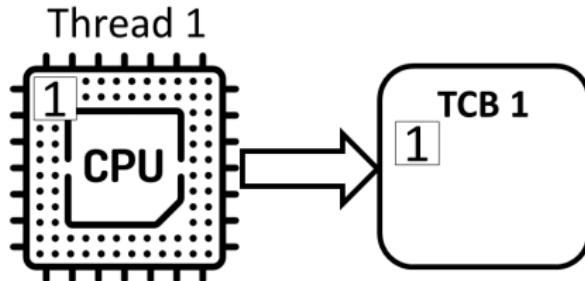
Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```



Thread 2:

```
...
...
Read counter -> register (= 1)
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```



Example

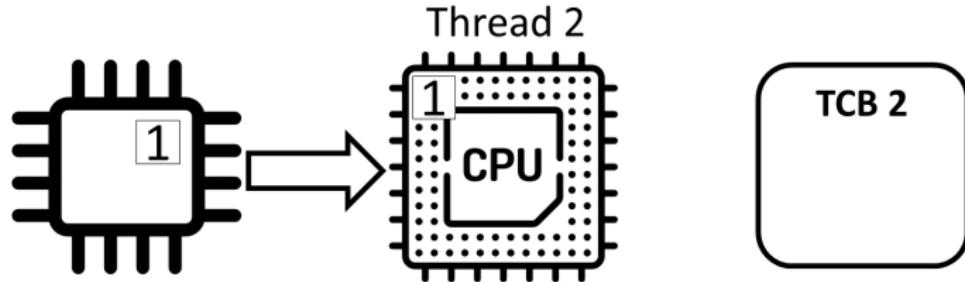
Incrementing a counter

Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

```
...
...
Read counter -> register (= 1)
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```



Example

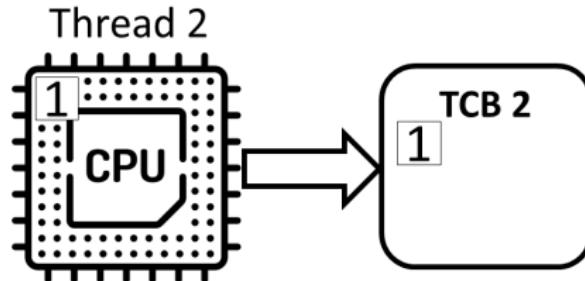
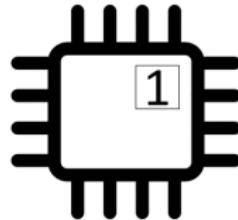
Incrementing a counter

Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

```
...
...
Read counter -> register (= 1)
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```



Example

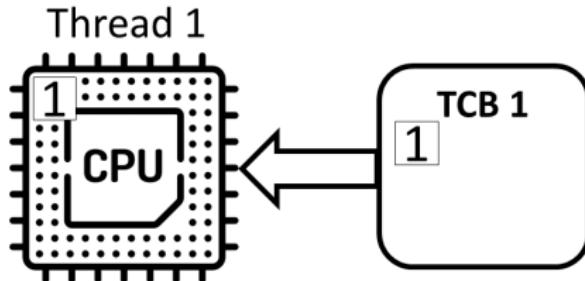
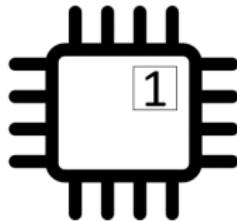
Incrementing a counter

Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

```
...
...
Read counter -> register (= 1)
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```



Example

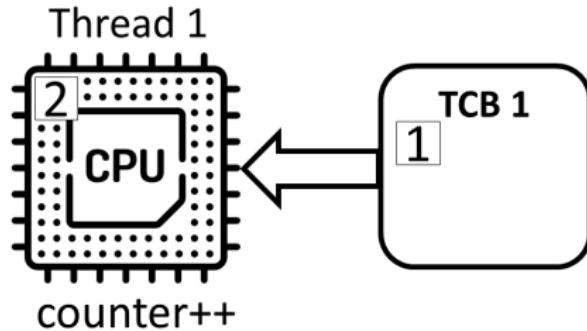
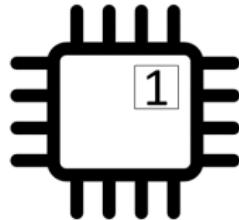
Incrementing a counter

Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

```
...
...
Read counter -> register (= 1)
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```



Example

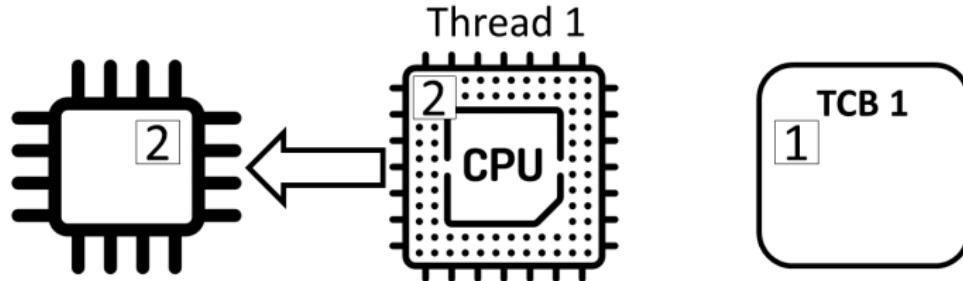
Incrementing a counter

Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

```
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```



Example

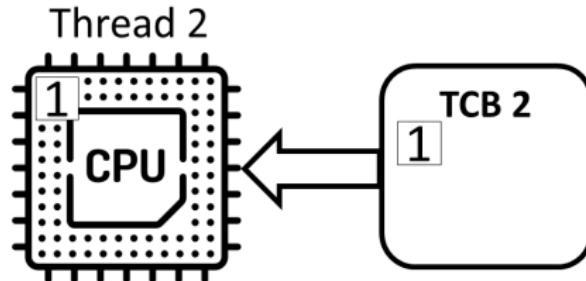
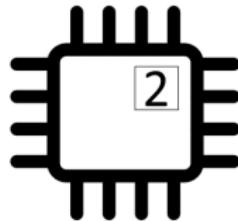
Incrementing a counter

Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

```
...
...
Read counter -> register (= 1)
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

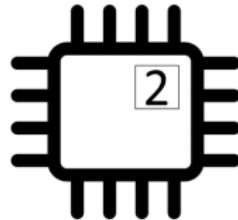


Example

Incrementing a counter

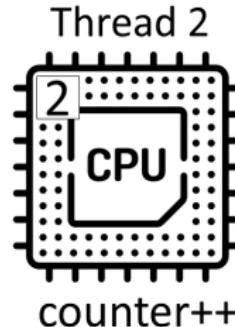
Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```



Thread 2:

```
...
Read counter -> register (= 1)
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```



Example

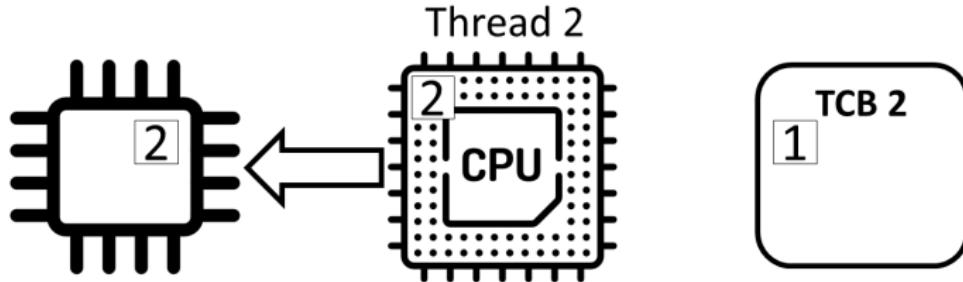
Incrementing a counter

Thread 1:

```
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

```
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```



Example 2

Shared procedures

- Consider the following **code shared** between threads/processes
- `chin` and `chout` **shared global variables**

```
void print()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

Example 2

Shared procedures

- Consider **two processes/threads** and the following **interleaved sequence of instructions** (they do **NOT** interact):

Thread 1:

```
...
chin = getchar(); ...
chout = chin;
putchar(chout);
...
...
...
```

Thread 2:

```
...
...
chin = getchar();
chout = chin;
putchar(chout);
```

Example 2

Shared procedures

- Consider **two processes/threads** and the following **interleaved sequence of instructions** (they **DO** interact):

Thread 1:

```
...
chin = getchar(); ...
...
chout = chin;
putchar(chout);
...
...
```

Thread 2:

```
...
chin = getchar();
...
...
chout = chin;
putchar(chout);
```

Example 3

Bounded Buffers

- Consider a **bounded buffer** in which N **items** can be stored
- A **counter** is maintained to count the number of items currently in the buffer
 - **Incremented** when an item is **added**
 - **Decrement** when an item is **removed**
- Similar **concurrency problems** as with the calculation of sums happen when multiple threads read and write to the bounded buffer.

Example 3

Bounded Buffers – Producer/Consumer

```
// producer
while (true) {
    while (counter == BUFFER_SIZE);
    buffer[in] = new_item;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

// consumer
while (true) {
    while (counter == 0);
    consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Race Conditions

Definition

- Code has a **race condition** if its behaviour is dependent on the timing of when computation is performed.
- A race condition typically occurs when multiple threads **access shared data** and the result is dependent on **the order in which the instructions are interleaved**.
- We will be interested in **mechanisms** to provide **synchronised** access to data and **avoid race conditions**.

Concurrency within the OS

Data Structures

- Kernels are preemptive these days
 - Multiple threads are running in the kernel.
 - Kernel code can be interrupted at any point.
- The kernel maintains **data structures** such as process tables and open file lists:
 - These data structures may be accessed **concurrently**.
 - These can be subject to **concurrency issues**.
- The OS must make sure that interactions within the OS **do not result in race conditions**.

Critical Sections and Mutual Exclusion

- As race conditions can lead to wild, unpredictable and plain wrong behaviour, we want software abstractions help to prevent them.
- A **critical section** is a section of code that can only be run by one thread at a time. This property is referred to as **mutual exclusion**.
- The question then becomes how to enforce mutual exclusion? Potentially:
 - The O/S and compiler could provide **direct support for critical sections** as a programming primitive.
 - The O/S and compiler provide **locks** which can be held by at most one thread at a time.

Mutual Exclusion

Definition

```
do {  
    ...  
    // ENTRY to critical section  
  
    critical section, e.g. counter++;  
  
    // EXIT critical section  
  
    // remaining code  
    ...  
} while (...);
```

Critical Sections, Mutual Exclusion

Definition

- A **solution to the critical section problem** should satisfy the following requirements:
 - **Mutual exclusion:** only one process can be in its critical section at any one point in time
 - **Progress:** any process must be able to enter its critical section at some point in time
 - Processes/threads in the “**remaining code**” do not influence access to critical sections
 - **Fairness/bounded waiting:** fairly distributed waiting times/processes cannot be made to wait indefinitely
- These requirements have to be satisfied, **independent of the order** in which computations are executed

Enforcing Mutual Exclusion

Approaches

- A standard approach to enforcing mutual exclusion is via locks known as **mutexes**. These can be implemented in various ways:
 - **Software based:** Peterson's solution
 - **Hardware based:** `test_and_set()`, `swap_and_compare()`
 - **O/S based** - the operating system blocks processes waiting for the lock.
- Unfortunately, **mutexes** and other concurrency primitives such as **semaphore** introduce a new problem - **deadlocks**.

Deadlocks

Example

- Assume that X and Y are **mutexes**.
- Thread A and B need to **acquire both mutexes**, and request them in **opposite orders**.
- The following **sequence of events** could occur in a **multi-programmed system**:

THREAD A:

```
request mutex X  
acquire mutex X  
...  
...  
request mutex Y  
...
```

THREAD B:

```
...  
...  
request mutex Y  
acquire mutex Y  
...  
request mutex X  
...
```

Deadlocks

Definition

Tanenbaum

*"A set of threads is deadlocked if **each thread** in the set is waiting for **an event** that only the **other thread** in the set can cause"*

- Each **deadlocked thread** is **waiting for** a resource held by **an other deadlocked thread** (which cannot run and hence cannot release the resources)
- This can happen between **any number of threads** and for **any number of resources**



Figure: Deadlocks

Deadlocks

Minimum Conditions

- **Four conditions** must hold for deadlocks to occur (Coffman et al. (1971)):
 - **Mutual exclusion**: a resource can be assigned to at most one process at a time
 - **Hold and wait condition**: a resource can be held whilst requesting new resources
 - **No preemption**: resources cannot be forcefully taken away from a process
 - **Circular wait**: there is a circular chain of two or more processes, waiting for a resource held by the other processes
- **No deadlocks** can occur if one of the conditions is **not satisfied**

Deadlocks

Minimum Conditions

- **Four conditions** must hold for deadlocks to occur (Coffman et al. (1971)):
 - **Mutual exclusion**: a resource can be assigned to at most one process at a time
 - **Hold and wait condition**: a resource can be held whilst requesting new resources
 - **No preemption**: resources cannot be forcefully taken away from a process
 - **Circular wait**: there is a circular chain of two or more processes, waiting for a resource held by the other processes
- **No deadlocks** can occur if one of the conditions is **not satisfied**
- If your **coursework solution deadlocks**, check for the **order in which resources are requested**

Test your understanding

- The code `x != y` doesn't modify anything. Is it certain to occur atomically?
- Can race conditions or deadlocks occur in practice on a machine with a single hardware thread?
- Can two threads running the same function deadlock against each other?

Operating Systems and Concurrency

Concurrency 2
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Coursework

- The coursework and supporting source code is now available on Moodle.
- The recommended submission deadline is **Monday 11th December, 2023**.
- The latest submission deadline is **Thursday 4th January, 2024**.
- No late submissions after that deadline!
- There will be a short introduction **next** weeks lab on **Friday 27th October, 2023**.

A question from last time

Question (paraphrased)

The slides last time defined deadlocks and critical sections in terms of processes - was this a typo?

A question from last time

Question (paraphrased)

The slides last time defined deadlocks and critical sections in terms of processes - was this a typo?

Answer

It doesn't really make very much difference. For the definitions in question, you need two key things:

- **Multiple sequences of instructions being executed concurrently** - both threads and processes satisfy this property.
- **Shared resources** - sharing is easier for threads. There are mechanisms to share resources between processes, including resources that **satisfy mutual exclusion**. A standard example would be **named semaphores**.

Recap

Last Lecture

- Examples of **concurrency issues** (e.g. `counter++`)
- Root causes of **concurrency issues** - Unpredictable order of execution.
- Potential solutions - **Critical sections** and enforcing **mutual exclusion**
- Further problems - **deadlocks**.

Mutual Exclusion

Approaches for Mutual Exclusion

- **Software based:** Peterson's solution
- **Hardware based:** `test_and_set()`, `swap_and_compare()`
- **OS based:** Blocking in the kernel.

Peterson's Solution

Software Solution

- Peterson's solution is a **software based solution** which worked well on **older machines**
- Two **shared variables** are used:
 - turn: indicates **which process is next** to enter its critical section
 - bool flag[2]: indicates that a **process is ready** to enter its critical section
- Can be **generalised to multiple processes**
- Peterson's solution for two processes **satisfies all “critical section requirements”** (mutual exclusion, progress, fairness)

Peterson's Solution

Software Solution

```
do {  
    flag[i] = true; // i wants to enter critical section  
    turn = j;        // allow j to access first  
    while (flag[j] && turn == j);  
    // whilst j wants to access critical section  
    // and its j's turn, apply busy waiting  
  
    // CRITICAL SECTION, e.g. counter++  
  
    flag[i] = false;  
  
    // remainder section  
} while (...);
```

Figure: Peterson's solution for process *i*

Peterson's Solution

Software Solution

```
do {  
    flag[j] = true; // j wants to enter critical section  
    turn = i;        // allow i to access first  
    while (flag[i] && turn == i);  
    // whilst i wants to access critical section  
    // and its i's turn, apply busy waiting  
  
    // CRITICAL SECTION, e.g. counter++  
  
    flag[j] = false;  
  
    // remainder section  
} while (...);
```

Figure: Peterson's solution for process *j*

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual exclusion requirement

- **Mutual exclusion requirement:** the variable turn can have **at most one value at a time**
 - Both `flag[i]` and `flag[j]` are true when they want to enter their critical section
 - Turn is a **singular variable** that can store **only one value**
 - Hence, at most one of `while(flag[i] && turn == i)` or `while(flag[j] && turn == j)` is true and at most **one process can enter its critical section** (mutual exclusion)

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

- **Progress:** any process must be able to enter its critical section at some point in time
 - Processes/threads in the “**remaining code**” **do not influence** access to critical sections
 - If process j does **not want to enter** its critical section
 - ⇒ `flag[j] == false`
 - ⇒ `while(flag[j] && turn == j)` will terminate for process i
 - ⇒ i enters critical section

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Fairness/bounded waiting requirement

- **Fairness/bounded waiting:** fairly distributed waiting times/processes cannot be made to wait indefinitely
 - If P_i and P_j both want to enter their critical section
 - $\Rightarrow \text{flag}[i] == \text{flag}[j] = \text{true}$
 - $\Rightarrow \text{turn}$ is either i or j \Rightarrow assuming that $\text{turn} == i \Rightarrow$
 - `while(flag[j] && turn == j) terminates and i enters section`
 - $\Rightarrow i$ finishes critical section $\Rightarrow \text{flag}[i] = \text{false} \Rightarrow$
 - `while(flag[i] && turn == i) terminates and j enters critical section.`
- Even if it loops back round again, it will set $\text{turn} = j$, letting the other thread in first.

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;  
...  
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    counter++;  
    flag[i] = false;  
} while (...);
```

Process i

```
flag[j] = false;  
...  
do {  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    counter++;  
    flag[j] = false;  
} while (...);
```

Process j

Atomic Instructions

Hardware approaches

- Implement `test_and_set()` and `swap_and_compare()` instructions as a **set of atomic (= uninterruptible) instructions**
 - Reading and setting the variables appears as **a single instruction**
 - If `test_and_set()` / `compare_and_swap()` are **called simultaneously**, they will be **executed sequentially**
- They can be used in combination with **lock variables**, assumed to be `true` (or 1) if the lock is **in use**

Atomic Instructions

Hardware approaches

```
// Test and set method
bool test_and_set(bool* bIsLocked) {
    bool rv = *bIsLocked;
    *bIsLocked = true;
    return rv;
}

// Example of using test and set method
do {
    // WHILE the lock is in use, apply busy waiting
    while (test_and_set(&bIsLocked));
    // Lock was false, now true

    // CRITICAL SECTION
    ...
    bIsLocked = false;
    ...
    // remainder section
} while (...)
```

Atomic Instructions

Hardware approaches: test_and_set() and compare_and_swap()

- Test and set must be **atomic/UN-interruptable**

THREAD 1

```
...
bool rv = *bIsLocked;
...
*bIsLocked = true;
return rv;
...
...
```

THREAD 2

```
...
bool rv = *bIsLocked;
...
*bIsLocked = true;
return rv;
```

while (test_and_set(&bIsLocked)); while (test_and_set(&bIsLocked));

Atomic Instructions

Hardware approaches: `test_and_set()` and `compare_and_swap()`

- Test and set must be **atomic/UN-interruptable**

```
THREAD 1                                THREAD 2
...
bool rv = *bIsLocked;
*bIsLocked = true;
return rv;
...
...
...
---                                         ---
while (test_and_set(&bIsLocked));    while (test_and_set(&bIsLocked));
```

Atomic Instructions

Hardware approaches: compare_and_swap()

```
// Compare and swap method
int compare_and_swap(
    int* iIsLocked, int expected, int new_value) {
    int const old_value = *iIsLocked;
    if(old_value == expected)
        *iIsLocked = new_value;
    return old_value;
}

do {
    // While the lock is in use (i.e. == 1), apply busy waiting
    while (compare_and_swap(&iIsLocked, 0, 1));
    // Lock was false, now true

    // CRITICAL SECTION
    ...
    iIsLocked = 0;
    ...
    // remainder section
} while (...);
```

Mutual Exclusion

Hardware approaches

- `test_and_set` and `compare_and_swap` are rather low level, and require busy waiting.
- The **OS may use these hardware instructions** to implement **higher level mechanisms** for mutual exclusion, i.e. **mutexes** and **semaphores**

Mutual Exclusion

Hardware approaches

- `test_and_set` and `compare_and_swap` are rather low level, and require busy waiting.
- The **OS may use these hardware instructions** to implement **higher level mechanisms** for mutual exclusion, i.e. **mutexes** and **semaphores**

Questions

We have two operations, `test_and_set` and `compare_and_swap` that have use as concurrency primitives. This raises some questions:

- What sort of primitive operations are useful?
- Are some better than others?
- Are there instructions that are best for concurrency support?

Mutexes

OS approaches

- **Mutexes** are an abstraction for providing **mutual exclusion**.
- A mutex provides an interface with two functions:
 - `acquire (&mutex)` : called **before entering** a critical section, returns when nobody else is in the critical section.
 - `release (&mutex)` : called **after exiting** the critical section, allows other threads to acquire the mutex. **Should only be called after a matching acquire.**
- Details of this interface may vary - names, or using methods in an object-oriented language.
- How exactly this interface is provided is an **implementation detail**.
 - Under naive assumptions, we could use Peterson's algorithm.
 - We could use atomic hardware operations and busy waiting.
 - The operating system could block threads that are trying to acquire a mutex that is not available.
 - Hybrid solutions combining several strategies might be chosen to optimise performance, depending on design assumptions.

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations; i++) {
        pthread_mutex_lock(&lock); // acquire
        counter++;
        pthread_mutex_unlock(&lock); // release
    }
    return 0;
}
int main() {
    pthread_t tid1 = 0, tid2 = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("The value of counter is: %d\n", counter);
}
```

Test your understanding

- Can you implement mutual exclusion on concurrent hardware without either operating system or hardware support?
- Would a mutex implemented using Peterson's algorithm work on modern hardware? (Worth trying as an exercise)
- Would you need to use mutexes / a critical region to protect code that is only reading from variables?

Operating Systems and Concurrency

Concurrency 3
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture - Approaches to Mutual Exclusion

- **Software approaches:** Peterson's solution
- **Hardware approaches:**
 - `test_and_set()`
 - `compare_and_swap()`
- **Mutexes** as an abstraction providing binary locks

Recap

Concurrency Primitives

- Recall **Mutexes** are a locking abstraction for providing mutual exclusion.
- Often provided by the operating system, via an API such as pthreads.
- They are **binary** - either a thread has currently acquired the mutex or it hasn't!

```
pthread_mutex_t lock;           // declaration  
  
pthread_mutex_lock(&lock);     // acquire  
counter++;  
pthread_mutex_unlock(&lock);    // release
```

Semaphores

OS approaches

- **Semaphores** are another abstraction for **mutual exclusion and process synchronisation**, often provided by **the operating system**
 - They have a **capacity**, either a positive number or infinity.
 - We distinguish between **binary** (2 valued) and **counting semaphores** (N-valued or unbounded).
- Two **functions** are used to **manipulate semaphores** (think of the counter++ example)
 - `wait()` is called when a resource is **acquired**, the capacity is **decremented**
 - `signal()` or `/post()` is called when a resource is **released**, the capacity is **incremented**.
- The semaphore **can only be acquired when its currently capacity is strictly positive**.
- A thread calling `post` **does not** have to have previously called `wait`.

Semaphores

OS approaches

```
typedef struct {
    int value;
    struct process * list;
} semaphore;
```

Figure: Conceptual definition of a semaphore

```
void wait(semaphore* S) {
    S->count--;
    if(S->count < 0) {
        //add process to S->list
        block(); // system call
    }
}
```

Figure: Conceptual implementation of a wait()

Semaphores

OS approaches

```
void post(semaphore* S) {  
    S->count++;  
    if(S->count <= 0) {  
        // remove process P from S->list  
        wakeup(P);  
    }  
}
```

Figure: Conceptual implementation of post()

Semaphores

Implementation

Thread 1

...
wait(&s) 1 => 0

...

...

...

post(&s)

...

...

...

Thread 2

...
...

wait(&s)
(wakeup)

...

...

post(&s)

...

...

...

Thread 3

...
...
...
...
wait(&s)

...

...

(wakeup)

...

post(&s)

...

Figure: Semaphore example

Semaphores

Implementation

Thread 1	Thread 2	Thread 3
...
wait(&s)
...
...	wait(&s) 0 => -1	...
...	(wakeup)	wait(&s)
...
...
post(&s)	post(&s)	(wakeup)
...
...	...	post(&s)
...

Figure: Semaphore example

Semaphores

Implementation

Thread 1

...
wait(&s)

...
...

post(&s)
...

...

...

...

...

Thread 2

...
...
...
wait(&s)
...
(wakeup)

...
...
post(&s)

...
...
...

Thread 3

...
...
...
...
wait(&s) -1 => -2
...

...
...
...
(wakeup)

...
post(&s)
...

Figure: Semaphore example

Semaphores

Implementation

Thread 1	Thread 2	Thread 3
...
wait(&s)
...
...	wait(&s)	...
...	...	wait(&s)
post(&s) -2 => -1	(wakeup)	...
...
...
...	post(&s)	(wakeup)
...
...	...	post(&s)
...

Figure: Semaphore example

Semaphores

Implementation

Thread 1	Thread 2	Thread 3
...
wait(&s)
...
...	wait(&s)	...
...	...	wait(&s)
post(&s)	(wakeup)	...
...
...
...	post(&s) -1 => 0	(wakeup)
...
...	...	post(&s)
...

Figure: Semaphore example

Semaphores

Implementation

Thread 1	Thread 2	Thread 3
...
wait (&s)
...
...	wait (&s)	...
...	...	wait (&s)
post (&s)	(wakeup)	...
...
...
...	post (&s)	(wakeup)
...
...	...	post (&s) 0 => 1
...

Figure: Semaphore example

Semaphores

OS approaches

- Calling `wait()` will **block** the process when the internal **counter is not positive**
 - ① The process **joins the a queue blocking on the semaphore**
 - ② The **process state** is changed from **running** to **blocked**
 - ③ Control is transferred to the **process scheduler**
- Calling `post()` **removes a process** from the **blocked queue** if available:
 - ① The process state is changed from **blocked** to **ready**
 - ② Different queueing strategies can be employed to **remove processes** - so avoid unjustified assumptions in your code.

Semaphores

OS approaches

- The queue length is the number of processes waiting on the semaphore.
- `block()` and `wakeup()` are **system calls** provided by the operating system.
- `post()` and `wait()` must be **atomic**.

Semaphores

OS approaches

```
void post(semaphore* S) {  
    lock(&mutex);  
    S->count++;  
    if(S->count <= 0) {  
        // remove process P from queue  
        wakeup(P);  
    }  
    unlock(&mutex);  
}
```

Posix Semaphores

Counter++ revisited

- Semaphores within the **same process** can be declared as variables of the type `sem_t`
 - `sem_init()` initialises the value of the semaphore
 - `sem_wait()` decrements the value of the semaphore
 - `sem_post()` increments the values of the semaphore
- An **explanation** of any of these functions can be found in the **man pages**, e.g. by typing `man sem_init` on the Linux command line

Posix Semaphores

Example

```
sem_t s;
int sum = 0;
void* calc(void* arg) {
    int const iterations = 50000000;
    for(int i = 0; i < iterations;i++) {
        sem_wait(&s);
        sum++;
        sem_post(&s);
    }
    return 0;
}
int main() {
    pthread_t tid1,tid2;
    sem_init(&s,0,1);
    pthread_create(&tid1, NULL, calc, 0);
    pthread_create(&tid2, NULL, calc, 0);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("The value of sum is: %d\n", sum);
}
```

Real-world issues

Standards support

Question

Does the previous code give the right answer on my Mac?

Real-world issues

Standards support

Question

Does the previous code give the right answer on my Mac?

Answer

Unfortunately, running the code on my Mac gives an answer slightly below 100000000! Details:

- Compiles with compiler warnings that `sem_init` is deprecated.
- `sem_init` is always failing, returning `-1`!
- Using named semaphores will work - see the lab for these.
- Even then, code using named semaphores must run as root on a Mac to call `sem_unlink` successfully.

Real-world issues

Standards support

Lessons

- **Never ignore compiler warnings.**
- **Always check return values** - slide examples don't for space reasons.
- **Test code thoroughly** - implicit assumption Mac would work like Linux was wrong!
- **Be aware of platform specific issues** such as `sem_unlink` behaviour on Mac.
- **Use the appropriate concurrency primitives** - the example really needed a mutex.

Efficiency

How/when to synchronise

- Synchronising code does result in a **performance penalty**
 - Synchronise **only when necessary**.
 - Synchronise **as few instructions** as possible.
- **Carefully consider how** to synchronise!

Using Semaphores

Counter++ revisited

```
void* calc(void* increments) {
    int number_of_iterations = 50000000;
    int total = 0;
    for(int i = 0; i < number_of_iterations; i++) {
        total++; // Pretend this is non-trivial to work out
    }
    sem_wait(&s);
    sum+=total;
    sem_post(&s);
    return 0;
}
```

Figure: Fast synchronised sums

Caveats

Potential Difficulties

- **Starvation:** poorly designed **queueing approaches** (e.g. LIFO) may result in fairness violations
- **Deadlocks:** two or more processes are **waiting indefinitely** for an event that can be **caused only by one of the waiting processes**
 - I.e., every process in a set is **waiting for an event** that can only be **caused by another process in the same set**
 - E.g., consider the following sequence of **instructions on semaphores**

P0	P1
wait(S);	...
...	wait(Q);
wait(Q);	...
...	wait(S);
...	...

The Producer/Consumer Problem

Problem Description

- **Producer(s)** and **consumer(s)** share a **buffer** of values - this could for example be a printer queue.
 - The buffer can be of **bounded** (maximum size N) or **unbounded size**.
 - There can any number of **producers** or **consumers**.
- A **producer** attempts to add items and **blocks** if the buffer is **full**.
- A **consumer** attempts to remove items and **blocks** if the buffer is **empty**.

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer

- The simplest version of the problem has **one producer, one consumer**, and a buffer of **unbounded size**
- A **counter (index)** variable keeps track of the number of **items in the buffer**
- It uses **two binary semaphores**:
 - `sync` **synchronises** access to the **buffer (counter)**, initialised to **1**
 - `delay_consumer` ensures that the **consumer blocks** when there are no items available, initialised to **0**

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); 0 => -1
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); (wakeup)
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); -1 => 0
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
        printf("%d\n", items);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer); 0 => -1
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer); (wakeup)
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); -1 => 0
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: First Attempt

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer with unbounded buffer

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer

- It is obvious that any **manipulations of items** will have to be **synchronised**
- **Race conditions** still exist:
 - When the consumer has **exhausted the buffer**, should have blocked, but the **producer increments items before the consumer checks it**

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); 0 => -1
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); (wakeup)
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); -1 => 0
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items >= 0)
            printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); 0 => 1
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items >= 0)
            printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer); 1 => 0
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items > -1)
            printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

Test your understanding

- Is a binary semaphore the same thing as a mutex?
- When should you prefer a mutex rather than a binary semaphore?
- Is there a straightforward way to check concurrent code is correct?

Operating Systems and Concurrency

Concurrency 4
COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Module Feedback

Pre-lecture Request

Please log in to Moodle and complete the module feedback form near the top of the page.

Recap

Last Lecture

- Concurrency synchronisation using **semaphores**.
- Producer / consumer problems and difficulties of concurrent programming.

Goals For Today

Classic Synchronisation Problems

- Continuing with the **bounded buffer** problem
- The **dining philosophers** problem

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); 0 => -1
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); (wakeup)
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); -1 => 0
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items >= 0)
            printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
            sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); 0 => 1
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items >= 0)
            printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
            sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer); 1 => 0
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items == -1)
            printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
            sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition
(non-existing element => items = -1)

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer

- Although we synchronise access to `items`, this did **not** prevent a race condition involving this variable!
- We did not preserve the relationship between `items` and the semaphore `delay_consumer`.

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); 0 => -1
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); (wakeup)
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); -1 => 0
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer); 0=>-1 (sleep)
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 0 => -1 (sleep)
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

- Use a **temporary variable**:
 - Copies the value of items inside the critical section
 - Decrement the `delay_consumer` semaphore to make it consistent

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); 0 => -1
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); (wakeup)
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); -1 => 0
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items >= 0)
            temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}
```



```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}
```



```
void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++; 0 => 1
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer); 0 => 1
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync); 0 => 1
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer); 1 => 0
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync); 1 => 0
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        if(items >= 0)
            temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer); 0 => -1
    }
}

void * producer(void * p)
{
    while(1)
    {
        sem_wait(&sync);
        items++;
        printf("%d\n", items);
        if(items == 1)
            sem_post(&delay_consumer);
        sem_post(&sync);
    }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

The Producer/Consumer Problem

Multiple Producers, Multiple Consumers, Bounded Buffer

- The previous code (one consumer, one producer) is made to work by **storing the value of items**
- A different variant of the problem has n **consumers**, m **producers**, and a **fixed buffer size N** . The solution is based on **3 semaphores**:
 - sync: used to **enforce mutual exclusion** for the buffer
 - empty: keeps track of the number of **empty buffers**, initialised to N
 - full: keeps track of the number of **full buffers**, initialised to 0
- The empty and full are **counting semaphores** and represent **resources**

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty); 3 => 2
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync); 1 => 0
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync); 0 => 1
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full); 0 => 1
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty); 2 => 1
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync); 1 => 0
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync); 0 => 1
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full); 1 => 2
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty); 1 => 0
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync); 1 => 0
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync); 0 => 1
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full); 2 => 3
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty); 0 => -1 (sleep)
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full); 3 => 2
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync); 1 => 0
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync); 0 => 1
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty); (wakeup)
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty); -1 => 0
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync); 1 => 0
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full); 2 => 1
        sem_wait(&sync);
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        printf("Producer: %d\n", items);
        sem_post(&sync);
        sem_post(&full);
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync); 0 => -1 (sleep)
        items--;
        printf("Consumer: %d\n", items);
        sem_post(&sync);
        sem_post(&empty);
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Producer/Consumer Problem

Multiple Producers & Consumers

```
void * producer(void * a)
{
    while(1)
    {
        sem_wait(&empty);
        sem_wait(&sync);
        items++;
        ...
        etc.
        ...
    }
}

void * consumer(void * a)
{
    while(1)
    {
        sem_wait(&full);
        sem_wait(&sync); 0 => -1 (sleep)
        ...
        etc.
        ...
    }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

The Dining Philosophers Problem

Description

- The problem is defined as:
 - **Five philosophers** are sitting on a round table
 - Each one has one has **a plate** of spaghetti
 - The spaghetti is too slippery, and each philosopher **needs 2 forks** to be able to eat
 - When hungry (in between thinking), the philosopher tries to **acquire the forks on their left and right**
- Note that this reflects the general problem of **sharing a limited set of resources** (forks) between **a number of processes** (philosophers)

The Dining Philosophers Problem

Description

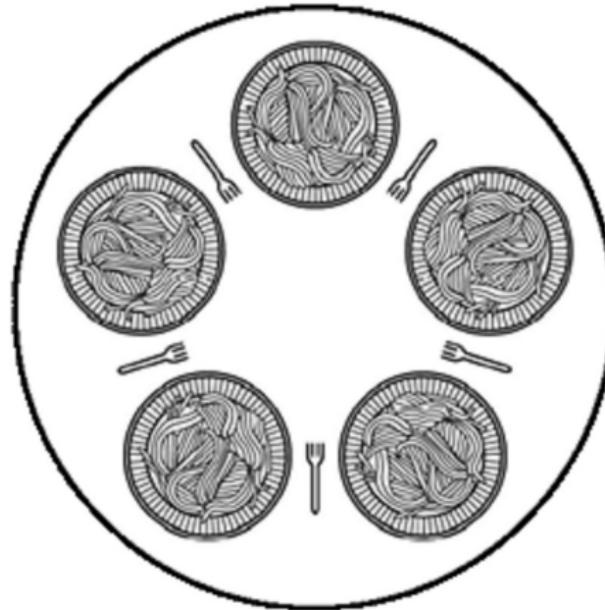


Figure: Tanenbaum, 4th edition

The Dining Philosophers Problem

Solution 1

- **Forks** are represented by **semaphores** (initialised to 1)
 - 1 if the **fork is available**: the philosopher can **continue**
 - 0 if the **fork is not available**: the philosopher goes to **sleep** if trying to acquire it
- First approach: Every philosopher **picks up one fork** and waits for the **second one to become available** (without putting the first one down)

The Dining Philosophers Problem

Solution 1: Naive will Deadlock

```
#define N 5
sem_t forks[N];

void * philosopher(void * id) {
    int i = *((int *) id);
    int left = (i + N - 1) % N;
    int right = i % N;
    while(1) {
        printf("%d is thinking\n", i);
        printf("%d is hungry\n", i);
        sem_wait(&forks[left]);
        sem_wait(&forks[right]);
        printf("%d is eating\n", i);
        sem_post(&forks[left]);
        sem_post(&forks[right]);
    }
}
```

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<code>wait(&f[4])</code> <code>1 => 0</code> <code>wait(&f[0])</code> <code>...</code> <code>// eating</code> <code>...</code> <code>post(&f[4])</code> <code>post(&f[0])</code>	<code>wait(&f[0])</code> <code>wait(&f[1])</code> <code>...</code> <code>// eating</code> <code>...</code> <code>post(&f[0])</code> <code>post(&f[1])</code>	<code>wait(&f[1])</code> <code>wait(&f[2])</code> <code>...</code> <code>// eating</code> <code>...</code> <code>post(&f[1])</code> <code>post(&f[2])</code>	<code>wait(&f[2])</code> <code>wait(&f[3])</code> <code>...</code> <code>// eating</code> <code>...</code> <code>post(&f[2])</code> <code>post(&f[3])</code>	<code>wait(&f[3])</code> <code>wait(&f[4])</code> <code>...</code> <code>// eating</code> <code>...</code> <code>post(&f[3])</code> <code>post(&f[4])</code>

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0]) 1 => 0	wait(&f[1])	wait(&f[2])	wait(&f[3])
wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0])	wait(&f[1]) 1 => 0	wait(&f[2])	wait(&f[3])
wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0])	wait(&f[1])	wait(&f[2]) 1 => 0	wait(&f[3])
wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3]) 1 => 0
wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])
...
// eating				
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])
wait(&f[0]) 0 => -1	wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])
wait(&f[0])	wait(&f[1]) 0 => -1	wait(&f[2])	wait(&f[3])	wait(&f[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])
wait(&f[0])	wait(&f[1])	wait(&f[2]) 0 => -1	wait(&f[3])	wait(&f[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])
wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3]) 0 => -1	wait(&f[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4])	wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])
wait(&f[0])	wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4]) 0 => -1
...
// eating	// eating	// eating	// eating	// eating
...
post(&f[4])	post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])
post(&f[0])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])

The Dining Philosophers Problem

Solution 1: Deadlock

- The naive solution can **deadlock**
- Deadlocks can be **prevented by**:
 - Putting the forks down and **waiting a random time** - potential for other bugs.
 - Putting **one additional fork** on the table - answering an easier problem!
 - One **global mutex** set by a philosopher when they want to eat - only one can eat at a time.

The Dining Philosophers Problem

Solution 1: Deadlock

- The naive solution can **deadlock**
- Deadlocks can be **prevented by**:
 - Putting the forks down and **waiting a random time** - potential for other bugs.
 - Putting **one additional fork** on the table - answering an easier problem!
 - One **global mutex** set by a philosopher when they want to eat - only one can eat at a time.
 - Solution does **not result in maximum parallelism** as only one eats at a time.

The Dining Philosophers Problem

Solutions 2: Global Mutex/Semaphore

```
sem_t eating;

void * philosopher(void * id)
{
    int i = (int) id;
    int left = (i + N - 1) % N;
    int right = i % N;
    while(1)
    {
        printf("%d is thinking\n", i);
        printf("%d is hungry\n", i);
        sem_wait(&eating);      /**** mutex/semaphore ****/
        sem_wait(&forks[left]);
        sem_wait(&forks[right]);
        printf("%d is eating\n", i);
        sem_post(&forks[left]);
        sem_post(&forks[right]);
        sem_post(&eating);      /**** mutex/semaphore ****/
    }
}
```

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating) 1=>0	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[4]) 1=>0	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0]) 1=>0 wait(&forks[1])		wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating) 0=>-1	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating) -1=>-2	wait(&eating)	wait(&eating)
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating) -2=>-3	wait(&eating)
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating) -3=>-4
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating				
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4]) 0=>1 post(&forks[0])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0]) 0=>1 post(&forks[1])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating) (wake)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[4])	wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[4])	post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&eating) -4=>-3	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Question in a Previous Year:

“Can I initialise the value of the eating semaphore to 2 to create more parallelism”

Test your understanding

- Can you find a better solution for the single producer consumer unbounded buffer problem?
- With $2 \times N$ philosophers, what is the maximum that can eat at once?
- What about $2 \times N + 1$ philosophers in a circle, what is the maximum number that can eat at once?

Operating Systems and Concurrency

Lecture 11: Concurrency
COMP2007 (G52OSC)

Geert De Maere
(Alexander Turner)
{Geert.DeMaere, Alexander.Turner}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2022

Goals

Today

- Parallel dining philosophers
- Readers/writers problem

The Dining Philosophers Problem

Solutions 2: Global Mutex/Semaphore

```
1 sem_t eating;
2
3 void * philosopher(void * id) {
4     int i = (int) id;
5     int left = (i + N - 1) % N;
6     int right = i % N;
7     while(1) {
8         printf("%d is thinking\n", i);
9         printf("%d is hungry\n", i);
10        sem_wait(&eating);           /**** semaphore ****/
11        sem_wait(&forks[left]);
12        sem_wait(&forks[right]);
13        printf("%d is eating\n", i);
14        sem_post(&forks[left]);
15        sem_post(&forks[right]);
16        sem_post(&eating);         /**** semaphore ****/
17    }
18 }
```

The Dining Philosophers Problem

Solutions 2: Global Mutex/Semaphore

Question in a Previous Year:

“Can I initialise the value of the eating semaphore to 2 to create maximum parallelism”

- Would it deadlock?
- Do we get maximum parallelism?

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating				
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating) 2=>1	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[0])	wait(&forks[1]) 1=>0	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2]) 1=>0	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating) 1=>0	wait(&eating)	wait(&eating)
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[0])	wait(&forks[1])	wait(&forks[2]) 0=>-1	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating) 0=>-1	wait(&eating)
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)-1=>-2
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating				
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating) -2=>-3	wait(&eating)	wait(&eating)	wait(&eating)	wait(&eating)
wait(&forks[0])	wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])
wait(&forks[1])	wait(&forks[2])	wait(&forks[3])	wait(&forks[4])	wait(&forks[0])
...
// eating	// eating	// eating	// eating	// eating
...
post(&forks[0])	post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])
post(&forks[1])	post(&forks[2])	post(&forks[3])	post(&forks[4])	post(&forks[0])
post(&eating)	post(&eating)	post(&eating)	post(&eating)	post(&eating)

The Dining Philosophers Problem

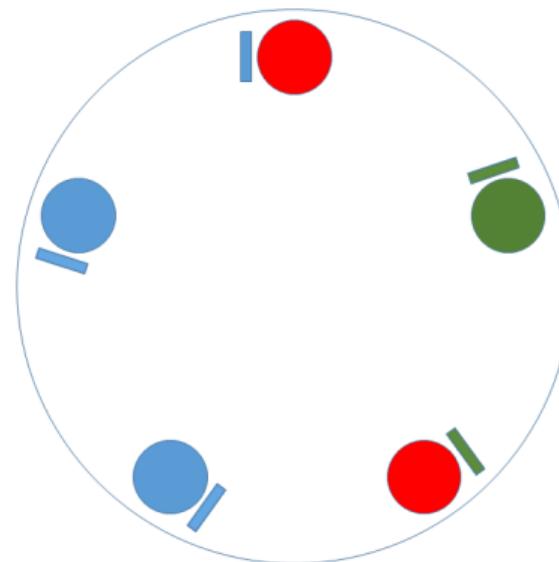
Solution 3: Maximum Parallelism

- A **more sophisticated solution** is necessary to allow **maximum parallelism**
- The solution uses:
 - state [N] : one **state variable** for every philosopher (THINKING, HUNGRY, EATING)
 - phil [N] : one **semaphore per *philosopher*** (i.e., **not forks, initialised to 0**)
 - The philosopher **goes to sleep** if one of his/her neighbours are eating
 - The **neighbours wake up the philosopher** if they have finished eating
 - sync: one **semaphore/mutex** to enforce **mutual exclusion** of the critical section (while updating the **states**)

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

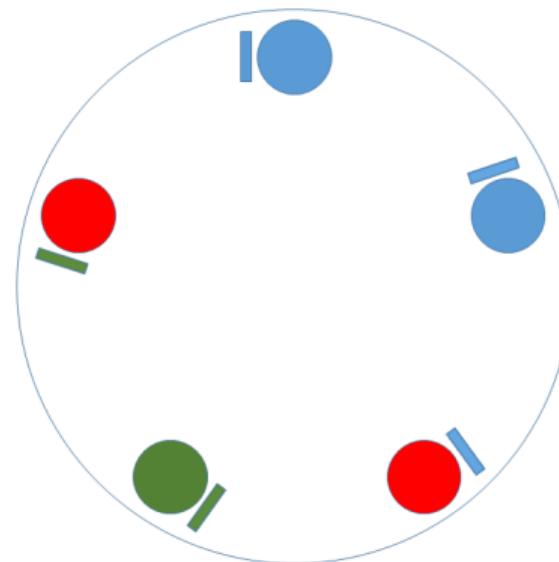
- A philosopher can only **start eating** if his/her **neighbours are not eating**



The Dining Philosophers Problem

Solution 3: Maximum Parallelism

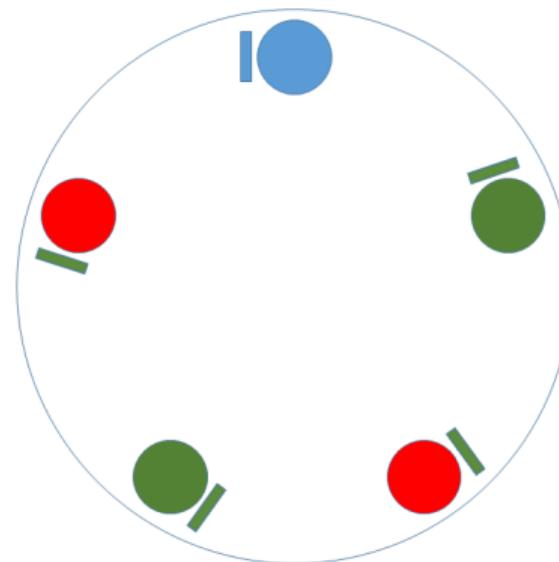
- A philosopher can only **start eating** if his/her **neighbours are not eating**



The Dining Philosophers Problem

Solution 3: Maximum Parallelism

- A philosopher can only **start eating** if his/her **neighbours are not eating**



The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
1 #define N 5
2 #define THINKING 1
3 #define HUNGRY 2
4 #define EATING 3
5
6 int state[N] = {THINKING, THINKING, THINKING, THINKING, THINKING};
7 sem_t phil[N]; // sends philosopher to sleep
8 sem_t sync;
```

```
1 void * philosopher(void * id) {
2     int i = *((int *) id);
3     while(1) {
4         printf("%d is thinking\n", i);
5         take_forks(i);
6         printf("%d is eating\n", i);
7         put_forks(i);
8     }
9 }
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
1 void take_forks(int i) {  
2     sem_wait(&sync);  
3     state[i] = HUNGRY;  
4     test(i);  
5     sem_post(&sync);  
6     sem_wait(&phil[i]);  
7 }
```

```
1 void test(int i) {  
2     int left = (i + N - 1) % N;  
3     int right = (i + 1) % N;  
4     if(state[i] == HUNGRY && state[left] != EATING && state[right] != EATING) {  
5         state[i] = EATING;  
6         sem_post(&phil[i]);  
7     }  
8 }
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
1 void put_forks(int i) {  
2     int left = (i + N - 1) % N;  
3     int right = (i + 1) % N;  
4     sem_wait(&sync);  
5     state[i] = THINKING;  
6     test(left);  
7     test(right);  
8     sem_post(&sync);  
9 }
```

```
1 void test(int i) {  
2     int left = (i + N - 1) % N;  
3     int right = (i + 1) % N;  
4     if(state[i] == HUNGRY && state[left] != EATING && state[right] != EATING) {  
5         state[i] = EATING;  
6         sem_post(&phil[i]);  
7     }  
8 }
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync) // 1 => 0
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3]) // 0 => 1
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync) // 0 => 1
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3]) // 0 => 1
}
post(&sync)
wait(&phil[3]) // 1 => 0

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync) // 1 => 0
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])//assume == -1 (wakeup) wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2]) // -1 => 0
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4]) // -1 => 0
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4]) // assume -1 (wakeup)
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync) // 0 => 1
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync) // 1 => 0
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync) // 0 => 1
wait(&phil[3])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3]) // 0 => -1 (sleeping)
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4]) // 0 => -1 (sleeping)
```

// EAT EAT EAT EAT EAT

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
```

```
wait(&sync) // 1 => 0
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

Philosopher 2
(left = 1, right = 3)

```
wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT) {
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]==EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3]) // -1 => 0
}
post(&sync)
```

Philosopher 3
(left = 2, right = 4)

```
wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3]) // wakeup
```

```
wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT) {
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

Philosopher 4
(left = 3, right = 5)

```
wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT) {
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
```

```
wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT) {
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT) {
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Readers – Writers Problem

Description

- Concurrent database processes are readers and/or writers, files, I/O devices, etc.
- **Reading** a record (variable) can happen **in parallel** without problems, **writing needs synchronisation** (i.e. exclusive access)
- **Different solutions** exist to the readers/writers problem
 - Solution 1: naive implementation with **limited parallelism**
 - Solution 2: **readers receive priority: no reader is kept waiting** (unless a writer already has access, **writers may starve**)
 - Solution 3: **writing is performed as soon as possible (readers may starve)**

The Readers – Writers Problem

Solution 1: No Parallelism

```
1 void * reader(void * arg) {
2     while(1) {
3         pthread_mutex_lock(&sync);
4         printf("reading record\n");
5         pthread_mutex_unlock(&sync);
6     }
7 }
```

```
1 void * writer(void * writer) {
2     while(1) {
3         pthread_mutex_lock(&sync);
4         printf("writing\n");
5         pthread_mutex_unlock(&sync);
6     }
7 }
```

The Readers – Writers Problem

Solution 2: Readers First

- Solution 1: prevents **parallel reading**
- Solution 2: **allows parallel reading**
- A correct implementation of solution 2 requires:
 - `iReadCount`: an integer tracking the number of readers
 - If `iReadCount > 0`: writers are blocked (`sem_wait (rwSync)`)
 - If `iReadCount == 0`: writers are released (`sem_post (rwSync)`)
 - If already writing, readers must wait
 - `sync`: a mutex for mutual exclusion of `iReadCount`
 - `rwSync`: a semaphore that **synchronises the readers and writers**, set by the **first/last reader**

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}
```

```
void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync); // iReadCount >= 0
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++; // 0=>1
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync); // 1=>0
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync); // 0=>1

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync); // 0=>-1
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync); // 1=>0
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--; // 1=>0
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync); // -1=>0
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync); // wakeup
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync); // 0=>1
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync); // 0=>1
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync); // 1=>0
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync); // iReadCount >= 0
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++; // 0=>1
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync); // 0=>-1 (sleep)
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);    (wakeup)
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync); // -1=>0
    }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sync);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&rwSync);
        sem_post(&sync);

        printf("reading record\n");

        sem_wait(&sync);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&rwSync);
        sem_post(&sync);
    }
}

void * writer(void * writer)
{
    while(1)
    {
        sem_wait(&rwSync);
        printf("writing\n");
        sem_post(&rwSync);
    }
}
```

Recap

Take-Home Message

- Dining philosophers with improved parallelism and maximum parallelism
- Readers/writers problem
 - Solution with limited/no parallelism
 - Solution with priority for the readers

Operating Systems and Concurrency

Concurrency 6 + Revision
COMP2007

Dan Marsden
(Geert De Maere)

{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Goals

Today

- Writers before readers.
- Beyond locks - alternative approaches to concurrency.

The Readers – Writers Problem (Last Time)

Solution 2: Readers First

- We implemented as solution to readers-writers that allowed concurrent reading when safe to do so.
- Recall the **room with a light switch analogy** - writers only enter a darkened room, readers happy to mix together.
- Unfortunately readers can overwhelm writers, and the **writers can starve**.
- Today - try to reverse the situation and **favour the writers**.

The Readers – Writers Problem

Solution 3: Writers First

Solution 3 gives priority to **writers** and uses:

- Integers `iReadCount` and `iWriteCount` to keep track of the number of readers/writers.
- Mutexes `sRead` and `sWrite` to synchronise the **reader's/writer's critical section**.
- Semaphore `sReadTry` to stop readers when there is **a writer waiting**.
- Semaphore `sResource` to **synchronise** the resource for **reading/writing**.

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry); // 1=>0
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}

void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead); // 1=>0
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++; // 0=>1
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource); // 1=>0
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead); // 0=>1
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry); // 0=>1

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}

void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite); // 1=>0
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++; // 0=>1
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry); // 1=>0
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}

void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite); // 0=>1

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource); // 1=>0
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource); // 0=>-1 (sleep)
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead); // 1=>0
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--; // 1=>0
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource); // -1=>0
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource); // wakeup
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead); // 0=>1
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry); // 0=>-1
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry); // 1=>0
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}

void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource); // (woken up)
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}

void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource); // 0=>1

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}

void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite); // 1=>0
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--; // 1=>0
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry); // wakeup
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}
```

```
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry); // -1=>0
        sem_post(&sWrite);
    }
}
```

The Readers – Writers Problem

Solution 3: Writers First

```
void * reader(void * arg)
{
    while(1)
    {
        sem_wait(&sReadTry);
        sem_wait(&sRead);
        iReadCount++;
        if(iReadCount == 1)
            sem_wait(&sResource);
        sem_post(&sRead);
        sem_post(&sReadTry);

        printf("reading\n");

        sem_wait(&sRead);
        iReadCount--;
        if(iReadCount == 0)
            sem_post(&sResource);
        sem_post(&sRead);
    }
}

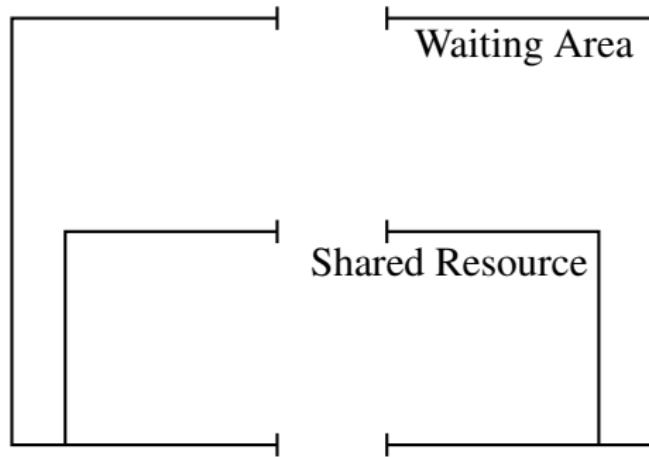
void * writer(void * arg)
{
    while(1)
    {
        sem_wait(&sWrite);
        iWriteCount++;
        if(iWriteCount == 1)
            sem_wait(&sReadTry);
        sem_post(&sWrite);

        sem_wait(&sResource);
        printf("writing\n");
        sem_post(&sResource);

        sem_wait(&sWrite);
        iWriteCount--;
        if(iWriteCount == 0)
            sem_post(&sReadTry);
        sem_post(&sWrite); // 0=>1
    }
}
```

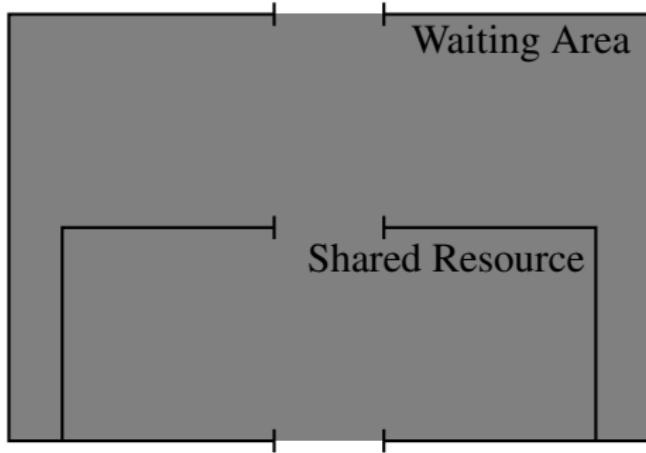
The Readers – Writers Problem

Solution 3: Writers First - Intuitions



The Readers – Writers Problem

Solution 3: Writers First - Intuitions

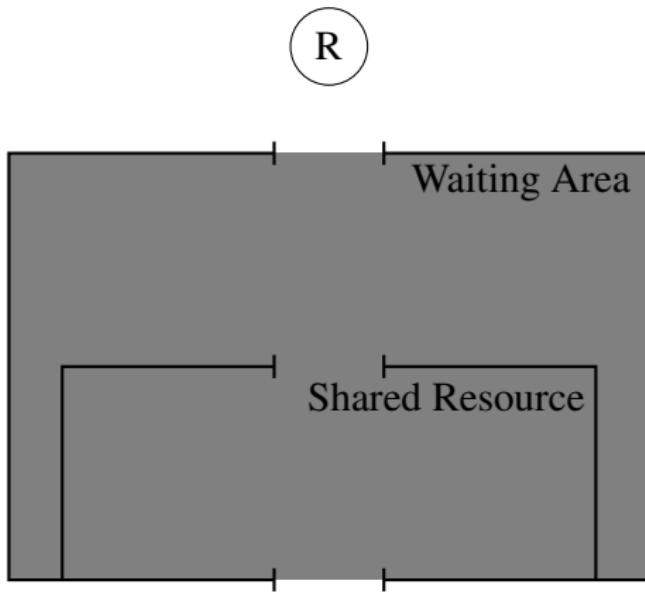


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

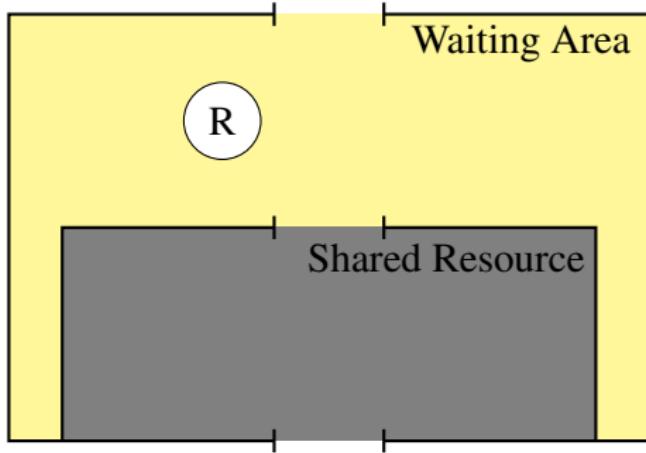


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

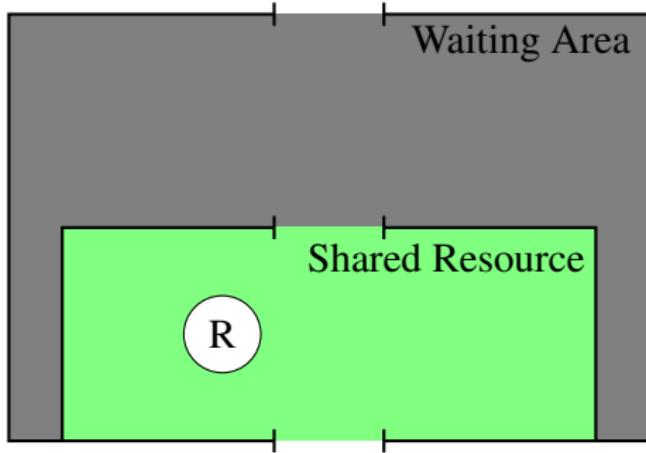


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

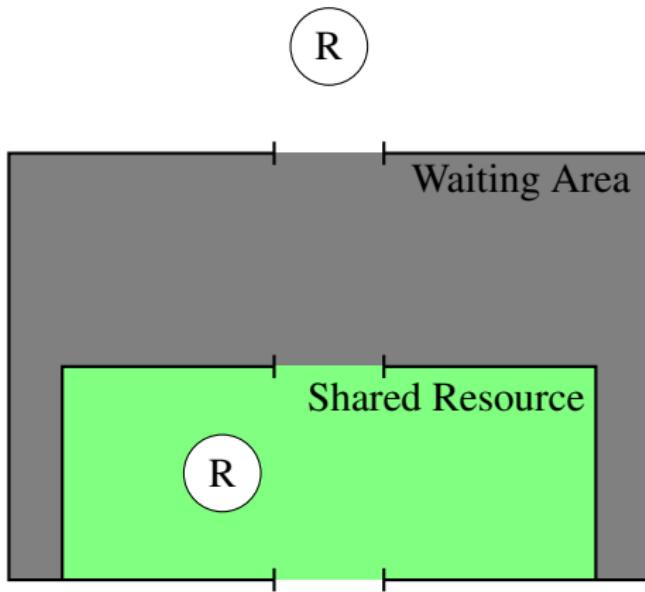


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

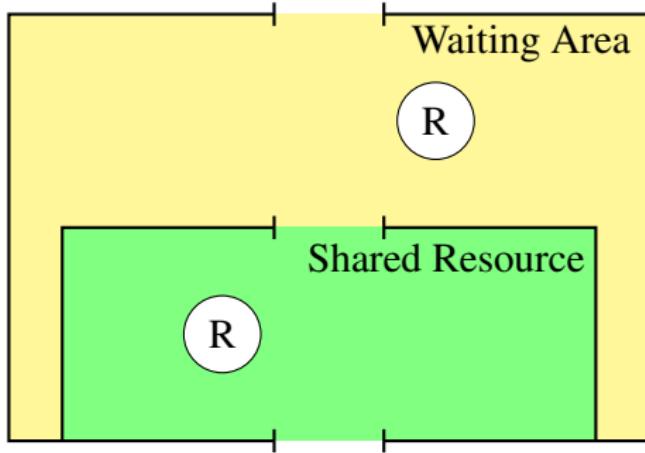


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

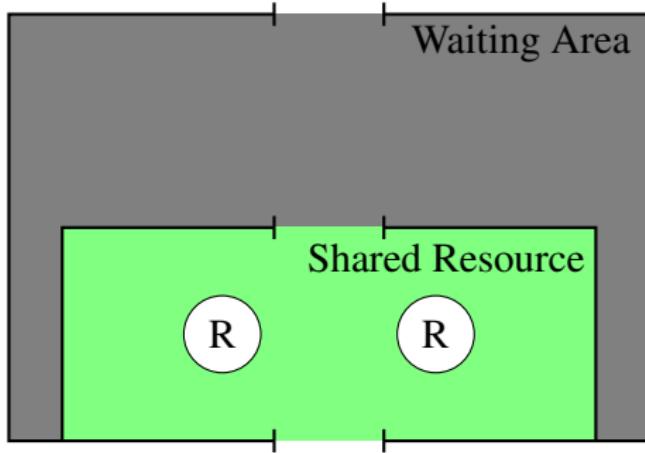


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

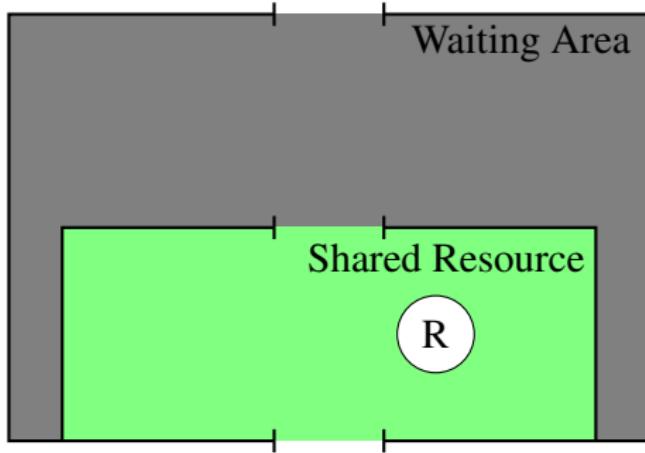


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

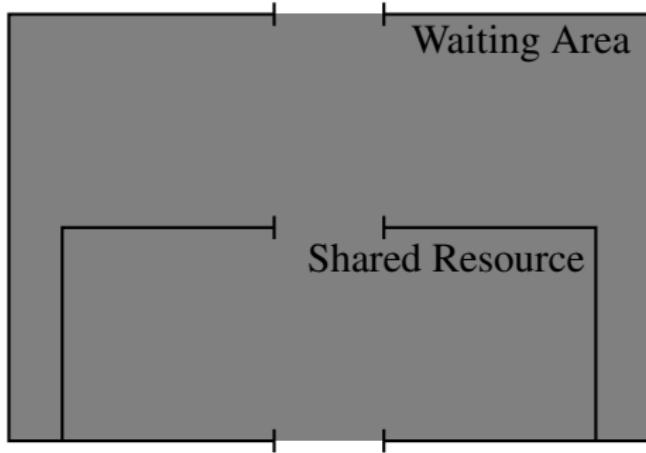


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

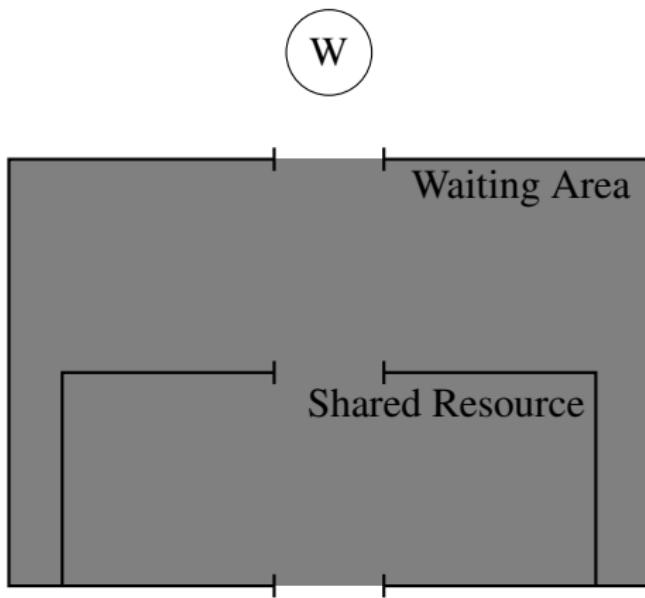


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

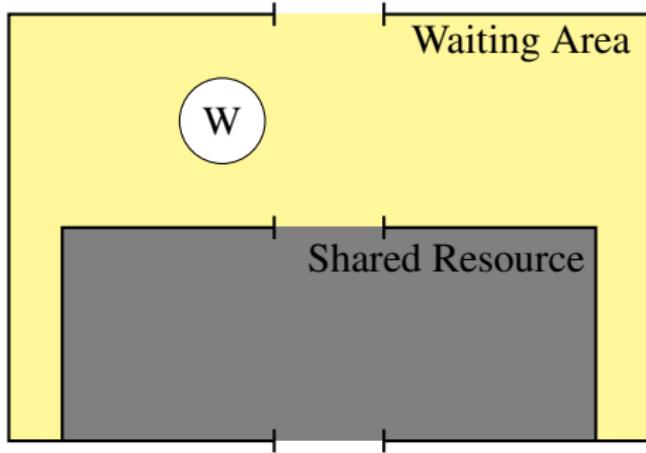


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

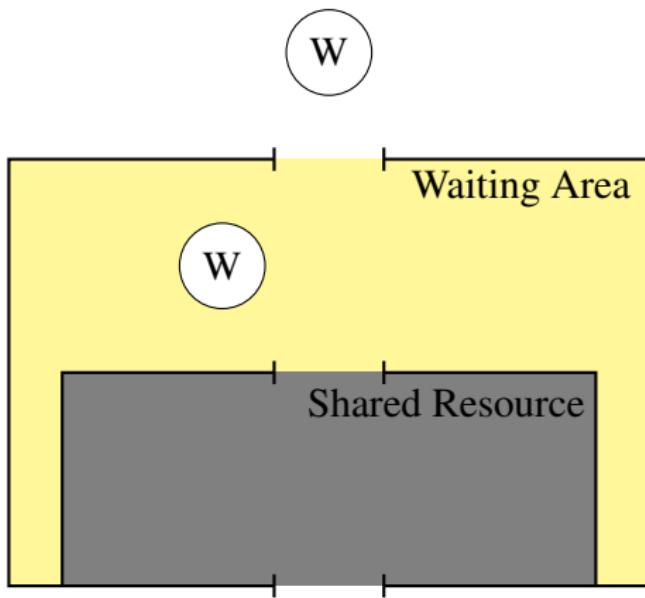


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

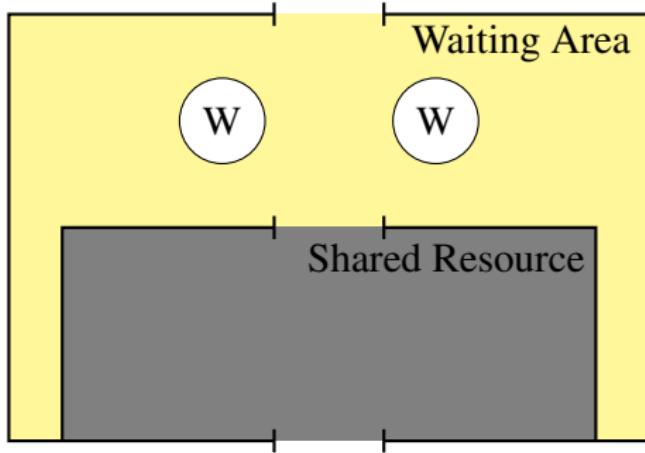


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

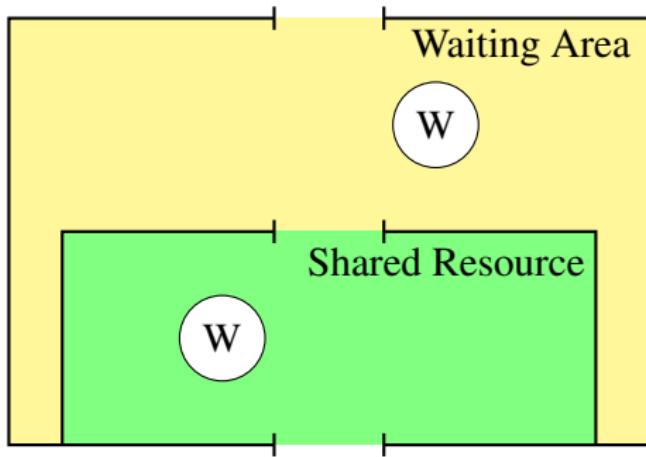


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

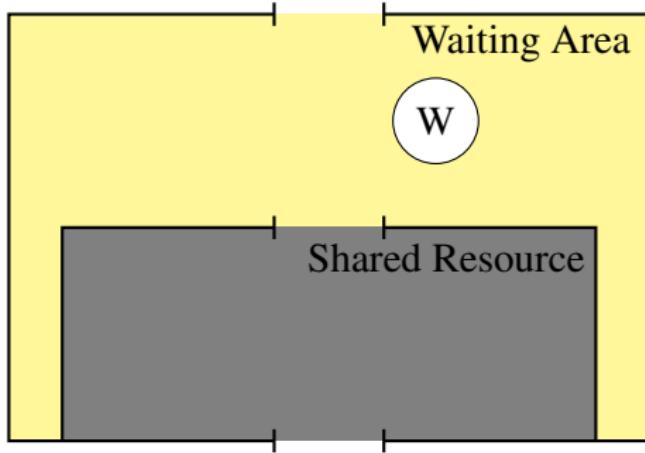


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

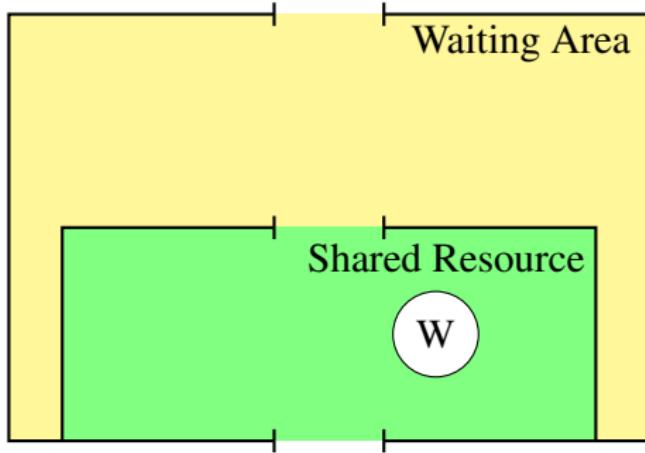


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

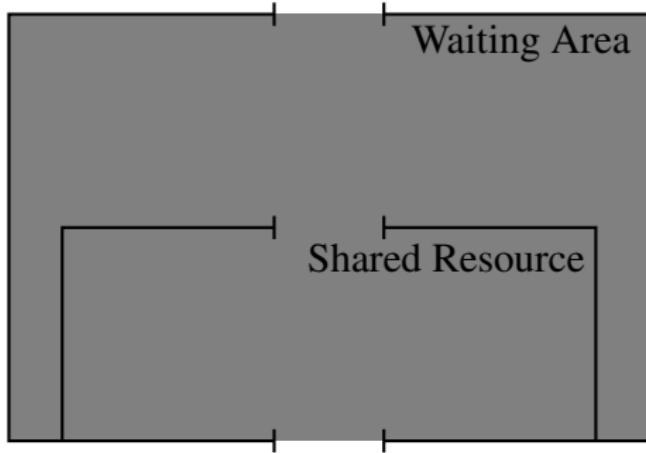


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

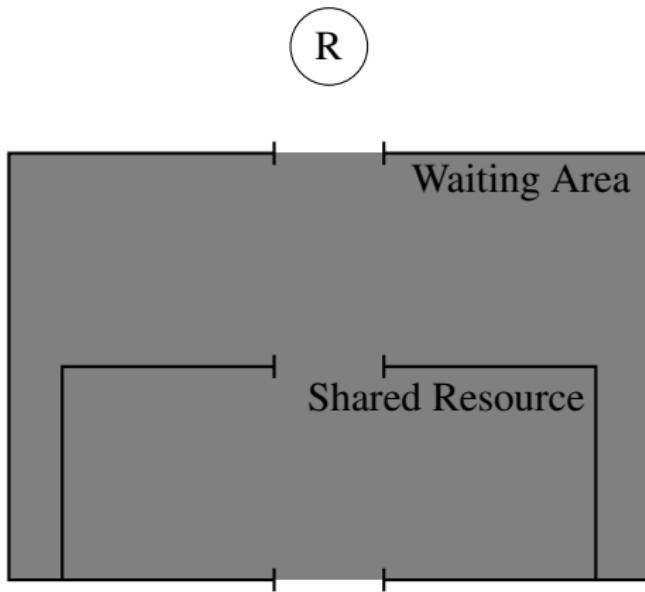


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

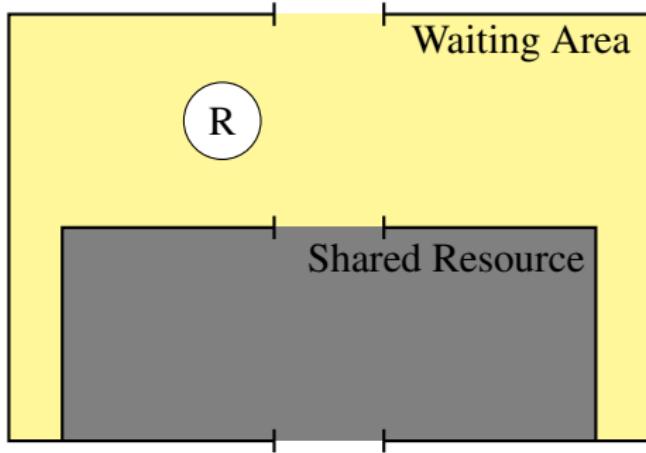


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

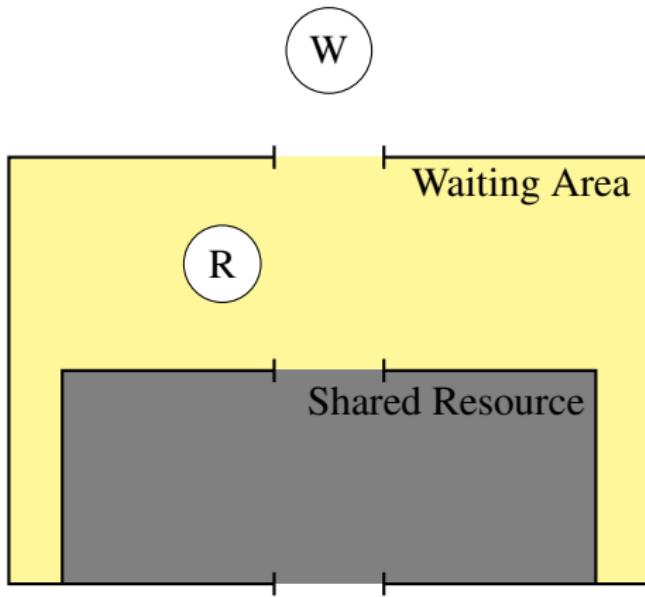


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

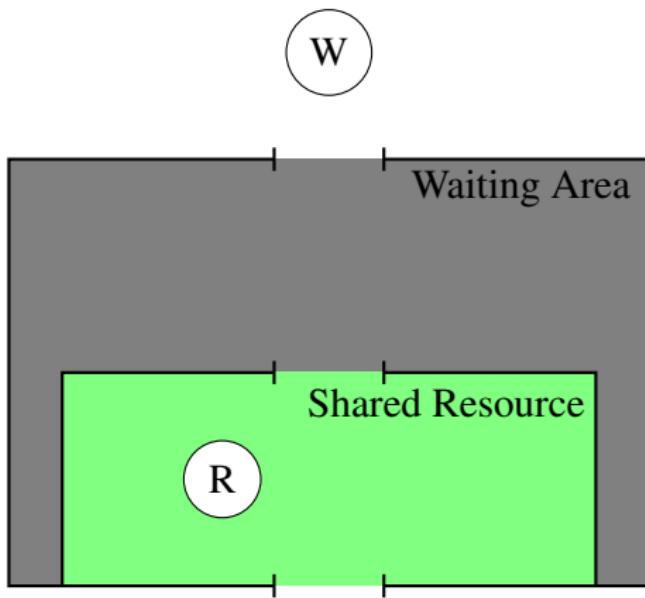


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

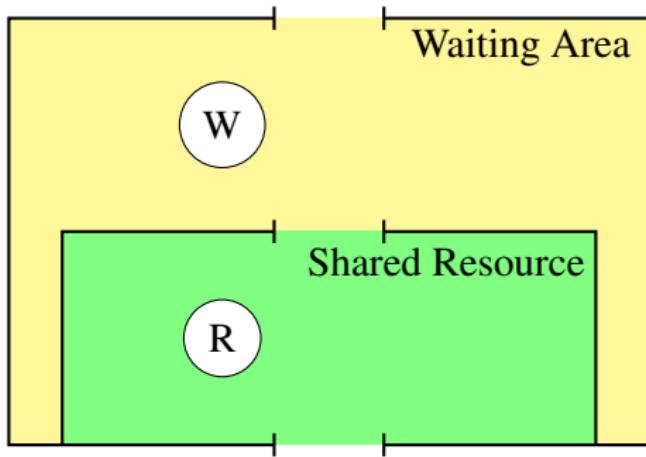


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

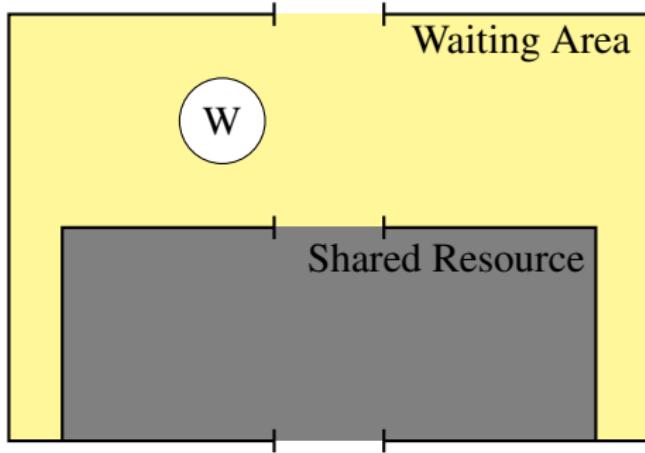


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

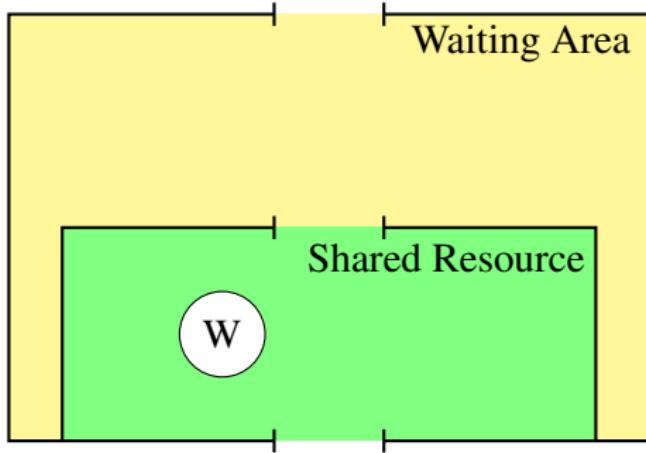


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

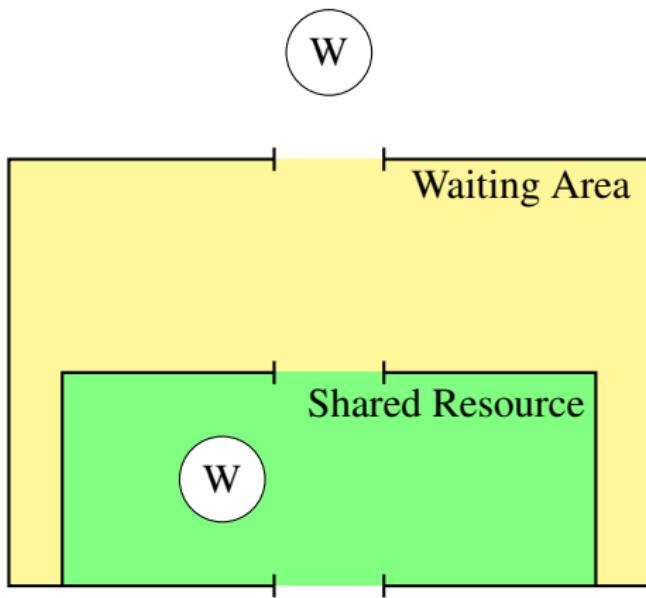


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

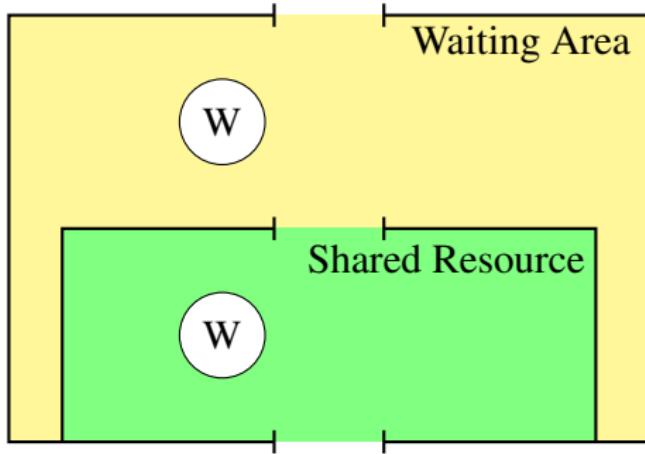


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

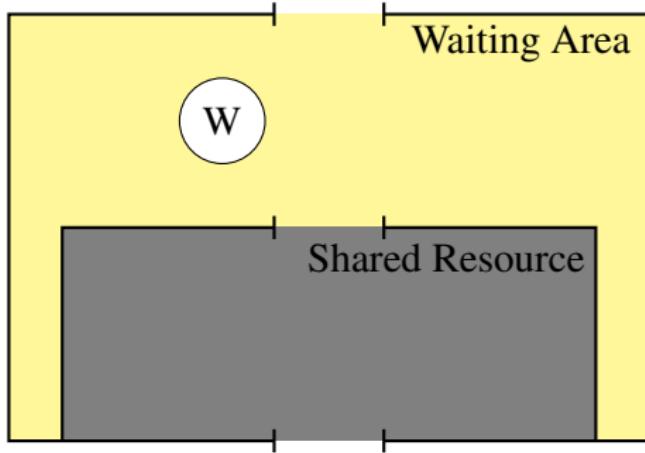


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions

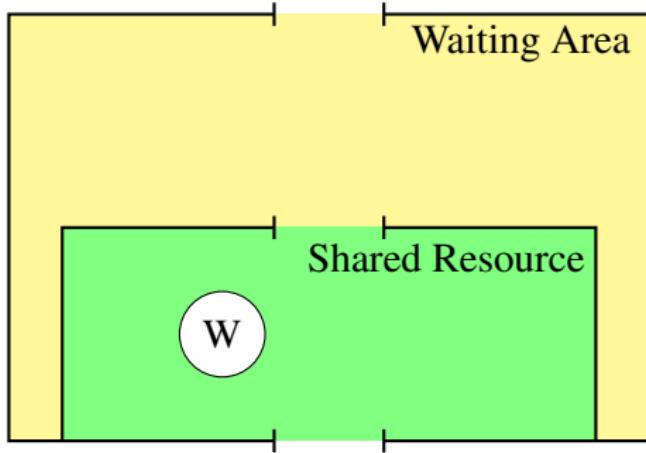


Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The Readers – Writers Problem

Solution 3: Writers First - Intuitions



Constraints:

- Everyone happy to go into a room with the light off.
- Readers and writers don't like each other.
- Readers happy to enter a room with a green light on, but not a yellow one.
- Writers happy to enter a room with a yellow light on, but not a green one.
- Only readers are allowed to use the bottom exit.

The trouble with locks

Locking style coordination of threads, such as with mutexes and semaphores has multiple problems. For example:

- **Deadlocks.**
- **Starvation.**
- **Priority inversion.**

The trouble with locks

Locking style coordination of threads, such as with mutexes and semaphores has multiple problems. For example:

- **Deadlocks.**
- **Starvation.**
- **Priority inversion.**
- Lack of **compositionality** - you cannot compose small correct concurrent programs to form large correct concurrent programs.

The trouble with locks

Locking style coordination of threads, such as with mutexes and semaphores has multiple problems. For example:

- **Deadlocks.**
- **Starvation.**
- **Priority inversion.**
- Lack of **compositionality** - you cannot compose small correct concurrent programs to form large correct concurrent programs.
- We could try to grab locks and back-off if we cannot acquire them all - we risk another problem - **livelock**.

The trouble with locks

Locking style coordination of threads, such as with mutexes and semaphores has multiple problems. For example:

- **Deadlocks.**
- **Starvation.**
- **Priority inversion.**
- Lack of **compositionality** - you cannot compose small correct concurrent programs to form large correct concurrent programs.
- We could try to grab locks and back-off if we cannot acquire them all - we risk another problem - **livelock**.
- **Access to shared state is implicit** - it can be hard to tell who's modifying what, and if suitable locks are held at the time.
- **Lock convoying** - threads end up queuing up behind a thread holding a lock - linearizes behaviour and can take a long time to clear.

Alternative Concurrency Approaches

We should not just be satisfied with such a status-quo. Therefore, we shall look at two alternatives:

Alternative Concurrency Approaches

We should not just be satisfied with such a status-quo. Therefore, we shall look at two alternatives:

- ① Message passing - **synchronous** or **asynchronous**.

Alternative Concurrency Approaches

We should not just be satisfied with such a status-quo. Therefore, we shall look at two alternatives:

- ① Message passing - **synchronous** or **asynchronous**.
- ② **Transactional memory**.

Message Passing

Synchronous Message Passing

- We avoid sharing memory, and instead have channels c, d, \dots
- There are two operations on channel c :
 - ① $c(x)$ sends message x over the channel.
 - ② $\bar{c}(x)$ receives a value over the channel into x .
- Communication is **synchronous** - a send must wait for a corresponding receive and vice-versa. **Think phone calls.**

Message Passing

Synchronous Message Passing

- We avoid sharing memory, and instead have channels c, d, \dots
- There are two operations on channel c :
 - ➊ $c(x)$ sends message x over the channel.
 - ➋ $\bar{c}(x)$ receives a value over the channel into x .
- Communication is **synchronous** - a send must wait for a corresponding receive and vice-versa. **Think phone calls.**

Thread 1

$c(data)$

Thread 2

...

Message Passing

Synchronous Message Passing

- We avoid sharing memory, and instead have channels c, d, \dots
- There are two operations on channel c :
 - ➊ $c(x)$ sends message x over the channel.
 - ➋ $\bar{c}(x)$ receives a value over the channel into x .
- Communication is **synchronous** - a send must wait for a corresponding receive and vice-versa. **Think phone calls.**

Thread 1

$c(data)$

blocked

Thread 2

...

...

Message Passing

Synchronous Message Passing

- We avoid sharing memory, and instead have channels c, d, \dots
- There are two operations on channel c :
 - ➊ $c(x)$ sends message x over the channel.
 - ➋ $\bar{c}(x)$ receives a value over the channel into x .
- Communication is **synchronous** - a send must wait for a corresponding receive and vice-versa. **Think phone calls.**

Thread 1

$c(data)$

blocked

blocked

Thread 2

...

...

$\bar{c}(x)$

Message Passing

Synchronous Message Passing

- We avoid sharing memory, and instead have channels c, d, \dots
- There are two operations on channel c :
 - ➊ $c(x)$ sends message x over the channel.
 - ➋ $\bar{c}(x)$ receives a value over the channel into x .
- Communication is **synchronous** - a send must wait for a corresponding receive and vice-versa. **Think phone calls.**

Thread 1

$c(data)$

blocked

blocked

...

Thread 2

...

...

$\bar{c}(x)$

... ($x = data$)

Message Passing

Synchronous Message Passing

Pros and cons of synchronous message passing:

- Flow of data is **explicit** - no exposure to race conditions and other difficult debugging.
- Synchronizing senders and receivers can slow performance.

Message Passing

Synchronous Message Passing

Pros and cons of synchronous message passing:

- Flow of data is **explicit** - no exposure to race conditions and other difficult debugging.
- Synchronizing senders and receivers can slow performance.
- Can implement mutexes and semaphores using synchronous message passing - **deadlocks, starvation etc. all still possible.**

Message Passing

Asynchronous Message Passing

In an attempt to mitigate the issues of synchronous message passing, **senders do not block** waiting for an available receiver. **Think sending an email.** Often use what is known as an **Actor** based model.

¹Example adapted from notes of Ian Stark
©University of Nottingham

Message Passing

Asynchronous Message Passing

In an attempt to mitigate the issues of synchronous message passing, **senders do not block** waiting for an available receiver. **Think sending an email.**

Often use what is known as an **Actor** based model.

For example, to specify a simple counter¹:

```
class Counter extends Actor {  
    var counter = 0  
  
    def receive = {  
        case Zero => counter = 0  
        case Inc => counter = counter + 1  
    }  
}
```

¹Example adapted from notes of Ian Stark
©University of Nottingham

Message Passing

Asynchronous Message Passing

In an attempt to mitigate the issues of synchronous message passing, **senders do not block** waiting for an available receiver. **Think sending an email.**

Often use what is known as an **Actor** based model.

For example, to specify a simple counter¹:

```
class Counter extends Actor {  
    var counter = 0  
  
    def receive = {  
        case Zero => counter = 0  
        case Inc => counter = counter + 1  
    }  
}
```

Then to increment a `Counter` object, you just send it a message:

```
mycounter ! Inc // Send Inc message - don't wait for receiver  
mycounter ! Inc // Send another - queued up until dealt with
```

¹Example adapted from notes of Ian Stark
©University of Nottingham

Message Passing

Asynchronous Message Passing

Pros and cons of synchronous message passing:

- As with the synchronous model, flow of data is **explicit** - less exposure to race conditions and other difficult debugging.
- Less **explicit** coupling between senders and receivers.
- *May* be less efficient than shared memory.

Message Passing

Asynchronous Message Passing

Pros and cons of synchronous message passing:

- As with the synchronous model, flow of data is **explicit** - less exposure to race conditions and other difficult debugging.
- Less **explicit** coupling between senders and receivers.
- *May* be less efficient than shared memory.
- Can simulate synchronous message passing.

Message Passing

Asynchronous Message Passing

Pros and cons of synchronous message passing:

- As with the synchronous model, flow of data is **explicit** - less exposure to race conditions and other difficult debugging.
- Less **explicit** coupling between senders and receivers.
- *May* be less efficient than shared memory.
- Can simulate synchronous message passing.
- Can implement mutexes using asynchronous message passing - **deadlocks, starvation etc. all still possible.**

Transactional Memory

A Database Analogy

Imagine I owe Geert £10.

- Inside the bank, our account balances might be stored in a database table:

Holder	Balance
Geert	10000000
Dan	15

- Transferring the money to Geert requires two steps:

- ➊ Deduct £10 from the Dan account.
- ➋ Add £10 to the Geert account.

Transactional Memory

A Database Analogy

Requirements for the bank transfer

- We would like the transfer to be **atomic** - In particular it either completely succeeds or totally fails.
- If other (atomic) changes are being made to the database, we would like them to be **serializable** - as if they don't overlap in time.

Transactional Memory

Database Transactions

To satisfy our requirements, we wrap our changes in a **transaction**:

```
begin transaction;  
update accounts set balance = balance + 10 where holder = 'Geert';  
update accounts set balance = balance - 10 where holder = 'Dan';  
commit;
```

Either both updates take place, or neither.

Transactional Memory

Database Transactions

To satisfy our requirements, we wrap our changes in a **transaction**:

```
begin transaction;  
update accounts set balance = balance + 10 where holder = 'Geert';  
update accounts set balance = balance - 10 where holder = 'Dan';  
commit;
```

Either both updates take place, or neither.

Rollback

Realistic transactions may be more complex, and can encounter reasons they cannot succeed. Instead of `commit` they explicitly call `rollback` to fail the whole transaction.

Transactional Memory

Transactions for thread coordination

The idea of transactional memory is to have the memory shared between threads behave transactionally, just like a database does.

```
do {  
    begin_transaction();  
    modify_shared_data();  
    commit();  
} while (!transaction_succeeds());
```

Transactional Memory

Transactions for thread coordination

The idea of transactional memory is to have the memory shared between threads behave transactionally, just like a database does.

```
do {  
    begin_transaction();  
    modify_shared_data();  
    commit();  
} while (!transaction_succeeds());
```

- No locks, and **no risk of deadlock**.

Transactional Memory

Transactions for thread coordination

The idea of transactional memory is to have the memory shared between threads behave transactionally, just like a database does.

```
do {  
    begin_transaction();  
    modify_shared_data();  
    commit();  
} while (!transaction_succeeds());
```

- No locks, and **no risk of deadlock**.
- A form of **livelock** still possible - as we back-off and try again after rollbacks.

Transactional Memory

Transactions for thread coordination

The idea of transactional memory is to have the memory shared between threads behave transactionally, just like a database does.

```
do {  
    begin_transaction();  
    modify_shared_data();  
    commit();  
} while (!transaction_succeeds());
```

- No locks, and **no risk of deadlock**.
- A form of **livelock** still possible - as we back-off and try again after rollbacks.
- **Starvation** still possible - smaller transactions may cause repeated rollbacks of longer ones.

Thanks!

Change of lecturer back to Geert and a new topic on Wednesday!

Operating Systems and Concurrency

Memory Management 1
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Remember

Subjects We Will Discuss

Subject	#Lectures	By
Introduction to operating systems/computer design	3	GDM/DM
Processes, process scheduling, threading, ...	4	DM
Concurrency (deadlocks)	6	DM
Memory management, swapping, virtual memory, ...	6	GDM
File Systems, file structures, management, ...	5	GDM
Revision	1	GDM

Table: Course structure

Goals for Today

Overview

- ① **Introduction** to memory management
- ② **Modelling** of multi-programming
- ③ Memory management based on **fixed partitioning**

Memory Management

Memory Hierarchies

- Computers typically have memory hierarchies:
 - **Registers, L1/L2/L3 cache**
 - **Main memory (RAM)**
 - **Disks**
- “**Higher memory**” is faster, more expensive and volatile, “**lower memory**” is slower, cheaper, and non-volatile
- The operating system provides a **memory abstraction**
- Memory can be seen as one **linear array** of bytes/words

Memory Management

OS Responsibilities

- **Allocate/deallocate** memory when requested by processes
- Keep track of **used/unused** memory
- **Distribute memory** between processes and simulate an “**infinitely large**” memory space
- **Control access** when multiprogramming is applied
- **Transparently** move data from **memory** to **disk** and vice versa

Memory Management

History of Memory Management

- Memory management has **evolved** over time
- **History repeats** itself:
 - Modern **consumer electronics** often require **less complex memory management** approaches
 - Many of the **early ideas underpin** more **modern memory management** approaches (e.g. relocation)

Partitioning

Approaches: Contiguous vs. Non-Contiguous



Figure: contiguous

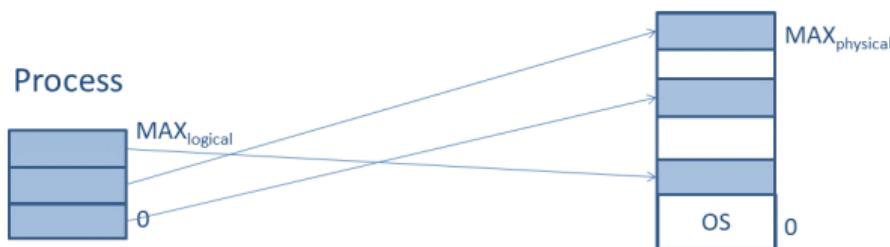


Figure: non-contiguous

Models

Approaches: Contiguous vs. Non-Contiguous

- **Contiguous memory management models** allocate memory in **one single block** without any **holes or gaps**
- **Non-contiguous memory management models:**
 - Allocate memory in **multiple blocks**, or **segments**
 - May be **placed anywhere in physical memory** (i.e., not necessarily next to each other)

Partitioning

Contiguous Approaches

- **Mono-programming:** one single partition for user processes
- **Multi-programming with fixed partitions**
 - Fixed **equal** sized partitions
 - Fixed **non-equal** sized partitions
- **Multi-programming with dynamic partitions**

Mono-Programming

No Memory Abstraction

- Only **one single user process** is in memory/executed at any point in time (no multi-programming)
- A fixed region of memory is allocated to the **OS/kernel**, the remaining memory is reserved for a **single process** (MS-DOS worked this way)
- This process has **direct access to physical memory** (i.e. no address translation takes place)

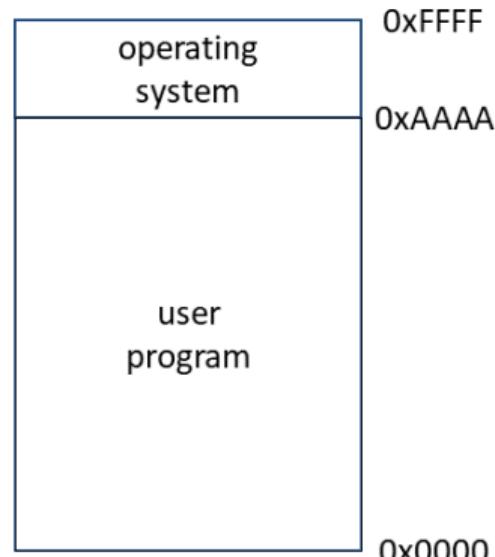


Figure: Mono-programming

Mono-Programming

No Memory Abstraction: Properties

- Every process is allocated **contiguous block of memory**, i.e. it contains no “holes” or “gaps”
- **One process** is allocated the **entire memory space**
- The process is **always located in the same address space**
- **No protection** between different user processes required (one process)
- **Overlays** enable the programmer to use more memory than available (burden on programmer)

Mono-Programming

No Memory Abstraction: Properties (Cont'ed)

- **Shortcomings** of mono-programming:
 - Since a process has **direct access to the physical memory**, it may have **access to OS** memory
 - The operating system can be seen as a process - so we have **two processes anyway**
 - **Low utilisation** of hardware resources (CPU, I/O devices, etc.)
 - **Multiprogramming is expected** on modern machines
- Direct memory access and mono-programming are common in basic **embedded systems** and **modern consumer electronics**, e.g. washing machines, microwaves, car's ECUs, etc.

Mono-Programming

Simulating Multi-Programming

- Simulate multi-programming through **swapping**
 - **Swap process** out to the disk and load a new one (context switches would become **time consuming**)
 - Apply **threads** within the same process (limited to one process)
- Assumption that **multiprogramming** can **improve CPU utilisation?**
 - Intuitively, this is true
 - How do we model this?

Multi-Programming

A Probabilistic Model

- There are n **processes in memory**
- A process spends p percent of its time **waiting for I/O**
- **CPU Utilisation** is calculated as 1 minus the time that all processes are waiting for I/O: e.g., $p = 0.9$ then CPU utilisation = $1 - 0.9 \Rightarrow 0.1 (1 - p)$
- The probability that **all n processes are waiting for I/O** (i.e., the CPU is idle) is p^n , i.e. $p \times p \times p \dots$
- The **CPU utilisation** is given by $1 - p^n$



Multi-Programming

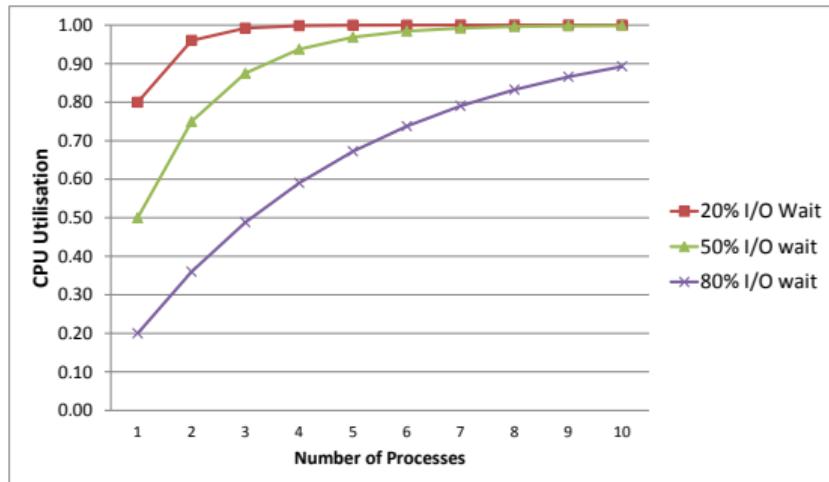
A Probabilistic Model

- With an **I/O wait time of 20%**, almost **100% CPU utilisation** can be achieved with four processes $(1 - 0.2^4)$
- With an **I/O wait time of 90%**, 10 processes can achieve about **65% CPU utilisation** $(1 - 0.9^{10})$
- CPU utilisation goes up with the number of processes and down for increasing levels of I/O**

Multi-Programming

A Probabilistic Model

# Processes	I/O Ratio		
	0.2	0.5	0.8
1	0.80	0.50	0.20
2	0.96	0.75	0.36
3	0.99	0.88	0.49
4	1.00	0.94	0.59
5	1.00	0.97	0.67
6	1.00	0.98	0.74
7	1.00	0.99	0.79
8	1.00	1.00	0.83
9	1.00	1.00	0.87
10	1.00	1.00	0.89



CPU utilisation as a function of the I/O ratio and the number of processes

Multi-Programming

A Probabilistic Model

- Assume that:
 - A computer has **1 MB** of memory
 - The **OS takes up 200KB**, leaving room for **four 200KB processes**
- Then:
 - If we have an **I/O wait time of 80%**, then we will achieve just under **60% CPU utilisation** ($1 - 0.8^4$)
 - If we add **1 MB of memory**, it would allow us to run another **five processes**
 - We can achieve about **87% CPU utilisation** ($1 - 0.8^9$)
 - If we add another **megabyte of memory** (fourteen processes) we will find that the CPU utilisation will increase to **about 96%** ($1 - 0.8^{14}$)

Multi-Programming

A Probabilistic Model

- Assume that:
 - A computer has **1 MB** of memory
 - The **OS takes up 200KB**, leaving room for **four 200KB processes**
- Then:
 - If we have an **I/O wait time of 80%**, then we will achieve just under **60% CPU utilisation** ($1 - 0.8^4$)
 - If we add **1 MB of memory**, it would allow us to run another **five processes**
 - We can achieve about **87% CPU utilisation** ($1 - 0.8^9$)
 - If we add another **megabyte of memory** (fourteen processes) we will find that the CPU utilisation will increase to **about 96%** ($1 - 0.8^{14}$)
- **Multi-programming** does enable to **improve resource utilisation**
 - ⇒ memory management should provide support for multi-programming

Multi-Programming

A Probabilistic Model

Caveats:

- This model assumes that **all processes are independent**, this is not true
- More complex models could be built using **queueing theory**, but we can still use this simplistic model to **make approximate predictions**

Partitioning

Fixed Partitions of equal size

- Divide memory into **static, contiguous** and **equal sized partitions** that have a **fixed size** and **fixed location**
 - Any process can take **any** (large enough) **partition**
 - Allocation of **fixed equal sized partitions** to processes is **trivial**
 - Very **little overhead** and **simple implementation**
 - The OS keeps a track of which partitions are being **used** and which are **free**

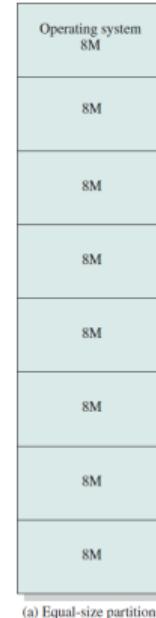
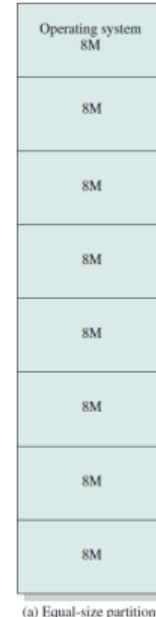


Figure: From Stallings

Partitioning

Fixed Partitions of equal size



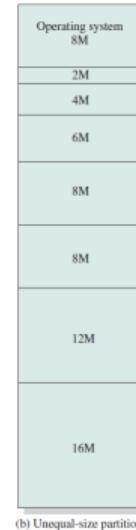
- **Disadvantages of static equal-sized partitions:**
 - **Low memory utilisation** and **internal fragmentation**:
partition may be unnecessarily large
 - **Overlays** must be used if a program does not fit into a partition (burden on programmer)

Figure: From Stallings

Partitioning

Fixed Partitions of non-equal size

- Divide memory into **static** and **non-equal sized partitions** that have a **fixed size** and **fixed location**
 - Reduces internal fragmentation**
 - The **allocation** of processes to partitions must be **carefully considered**



(b) Unequal-size partitions

Figure: From Stallings

Partitioning

Fixed Partitions (Allocation Methods)

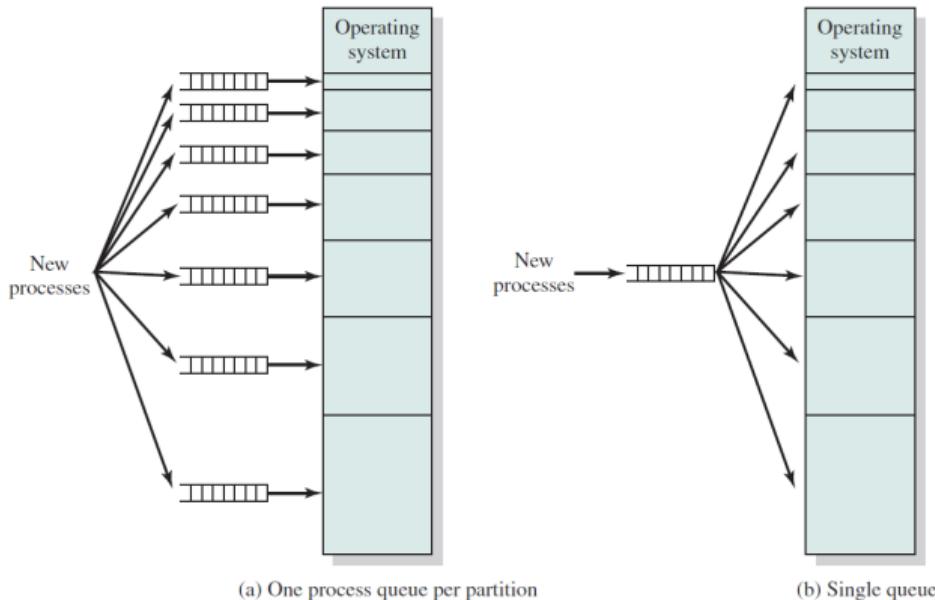


Figure: From Stallings

Partitioning

Fixed Partitions (Allocation Methods)

- One **private queue** per partition:
 - Assigns each process to the **smallest partition** that it would fit in
 - Reduces **internal fragmentation**
 - Can **reduce memory utilisation** (e.g., lots of small jobs result in unused large partitions) and result in **starvation**
- A single **shared queue** for all partitions can allocate small processes to **large partitions** but results in **increased internal fragmentation**

Test Your Understanding

- The compiler allocates **memory addresses**
- What is **the issue?**
- How would you **resolve it?**

Recap

Take-Home Message

- **Mono-programming** and **absolute addressing (no memory abstraction)**
- Why multi-programming: CPU utilisation modelling
- Memory management for **multi-programming**: fixed (non-)equal **partitions**

Operating Systems and Concurrency

Memory Management 2
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recall

Last Lecture

- **Mono-programming** with **absolute addressing**
- Modelling **CPU utilisation**
- **Multi-programming** with fixed (non-)equal **partitions** to improve **CPU utilisation**
- **Internal fragmentation**

Overview

Goals for Today

- Code **relocation** and **protection**
- **Dynamic partitioning & segmentation**
- **Managing free/occupied memory**

Relocation and Protection

Example

```
1 #include <stdio.h>
2 #include <unistd.h>
3 int iVar = 0;
4 int main() {
5     while(iVar++ < 10) {
6         printf("Addr:%p; Val:%d\n", &iVar, iVar);
7         sleep(1);
8     }
9     return 0;
10 }
```

- If **running the code** twice (simultaneously):
 - Will the **same or different addresses** be displayed for *iVar*?
 - Will the value of *iVar* in the first run **influence** the value of *iVar* in the second run?
- Note that this may not work on “newer” OSs that use **Address Space Layout Randomization**

Relocation and Protection

Principles

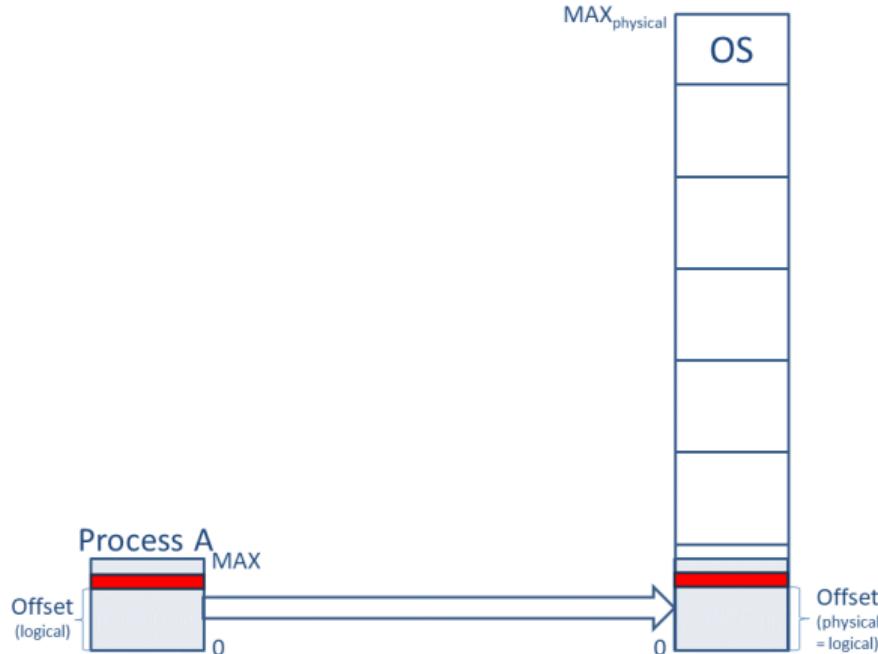


Figure: Address relocation

Relocation and Protection

Principles

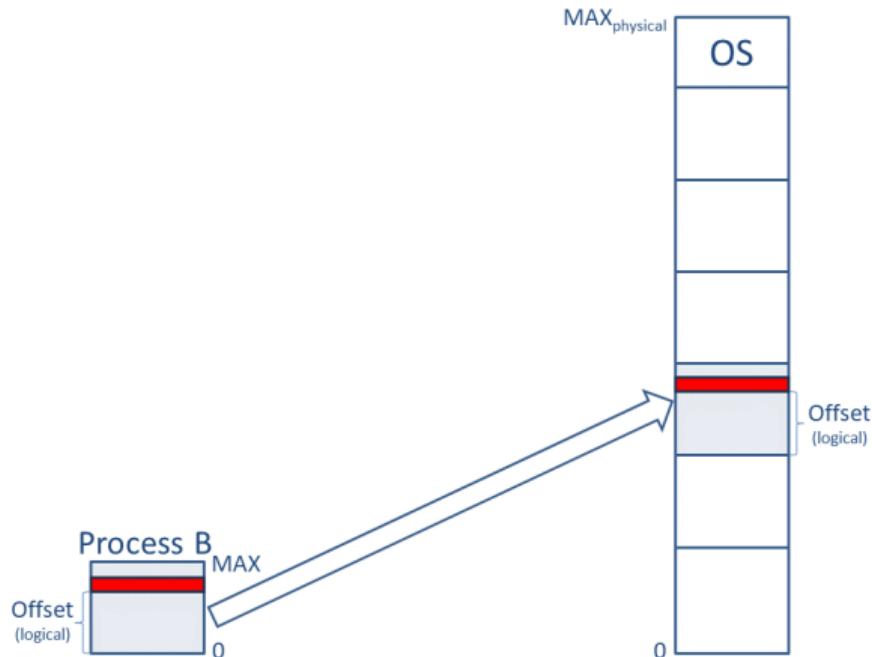


Figure: Address relocation

Relocation and Protection

Principles

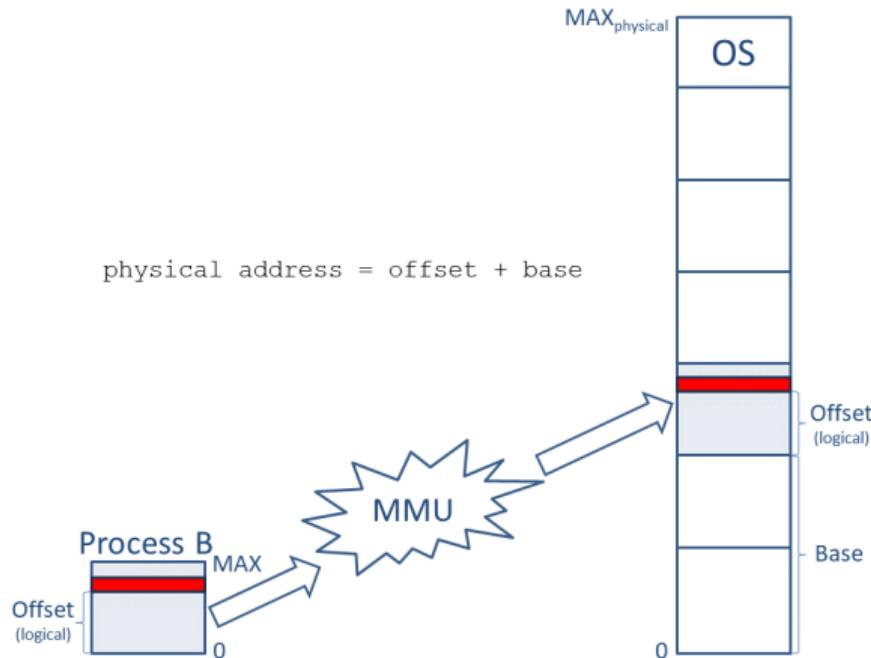


Figure: Address relocation

Relocation and Protection

Principles

- **Relocation:** One does not know at compile time which partition/addresses a process will occupy
 - **Compiler** assumes the process that it will run at 0
 - **Mapping** takes place once it is known where in physical memory the process resides
 - **Relocation** must be used in any operating system that allows processes to run at **changing locations in physical memory**
- **Protection:** once you can have multiple processes in memory at the same time, **protection** must be enforced

Relocation and Protection

Principles

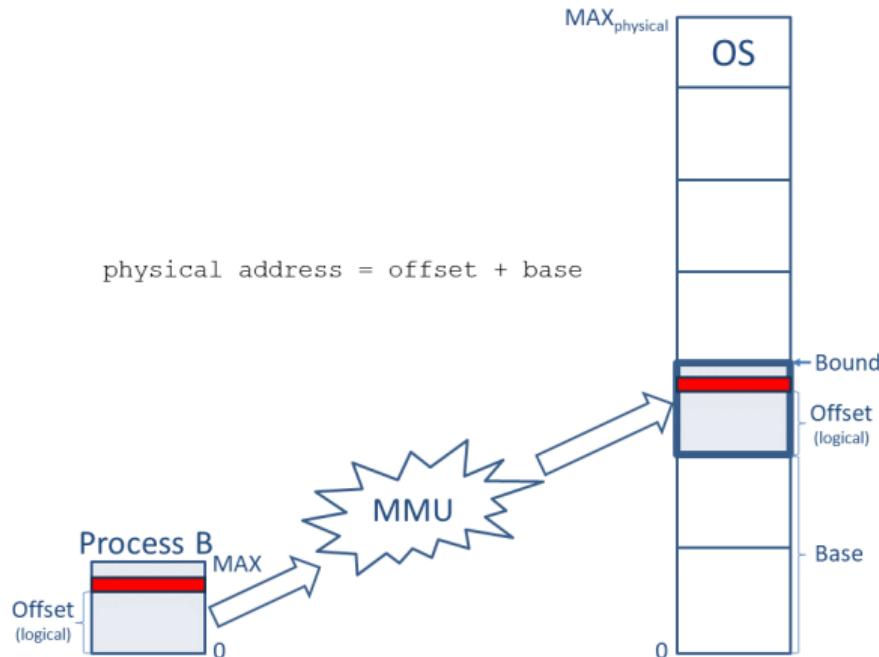


Figure: Address relocation

Relocation and Protection

Address Types

- A **logical address** is a memory address **seen by the process**
 - **Relative to the start of the program**
 - Assigned by the **compiler**
 - **Independent** of the current location in **physical memory**
- The **physical address** refers to an **actual location in main memory**
- The **logical address space** must be mapped onto the machine's **physical address space**

Relocation and Protection

Approaches

- ① **Static “relocation” at compile time:** a process has to be located at the same location every single time (impractical)
- ② **Dynamic relocation at load time**
 - An **offset** is added to every logical address to **account for its physical location** in memory
 - **Slows down** the loading of a process
 - Does not account for **swapping**
- ③ **Dynamic relocation at runtime with hardware support**

Relocation and Protection

At Runtime: Base and Bound Registers

- Two special purpose registers are maintained in the CPU (the **MMU**), containing a **base address** and **bound**
 - The **base register** stores the **start address** of the partition
 - The **bound register** holds the **size** of the partition
- **At runtime:**
 - The **base register** is added to the **logical (relative) address** to generate the **physical address**
 - The resulting address is **compared** against the **bound register**
- **Hardware support** was not always present in the early days!

Relocation and Protection

Base and Bound Registers

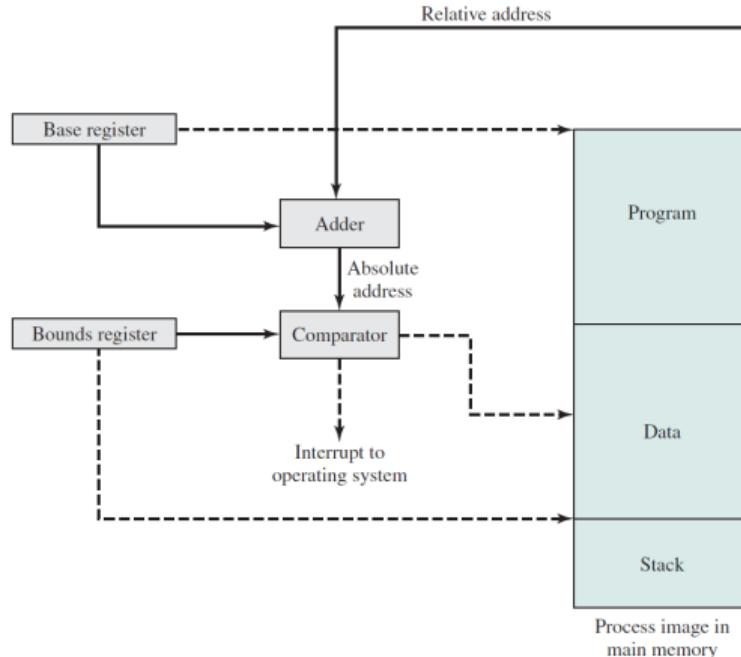


Figure: Address Relocation (Stallings)

Dynamic Partitioning

Context

- **Fixed partitioning** results in **internal fragmentation**:
 - An **exact match** for the process and may not always be **available**
 - The partition may **not be used completely**
- **Dynamic partitioning**:
 - A **variable number of partitions**
 - A process is allocated the **exact amount of contiguous** memory
 - **Reduces internal fragmentation**

Dynamic Partitioning

Example

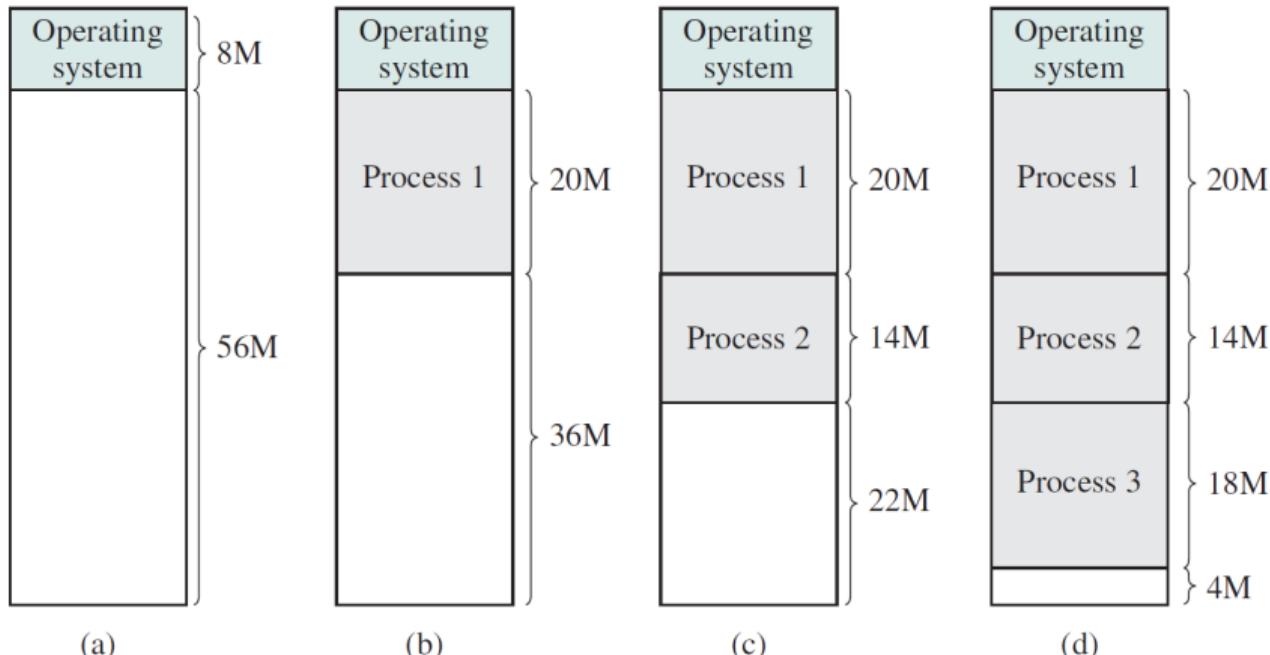


Figure: Dynamic partitioning (from Stallings)

Dynamic Partitioning

Swapping

- Swapping **moves processes** between the drive and main memory
- **Reasons** for swapping:
 - Some **processes** only **run occasionally**
 - The **total amount of memory** required **exceeds available memory**
 - **Memory requirements change** / cannot be **known in advance** (stack / heap)
 - We have **more processes** than **partitions** (assuming fixed partitions)
- Processes can be reloaded into a **different memory location** (base register changes)

Dynamic Partitioning

Swapping: Example

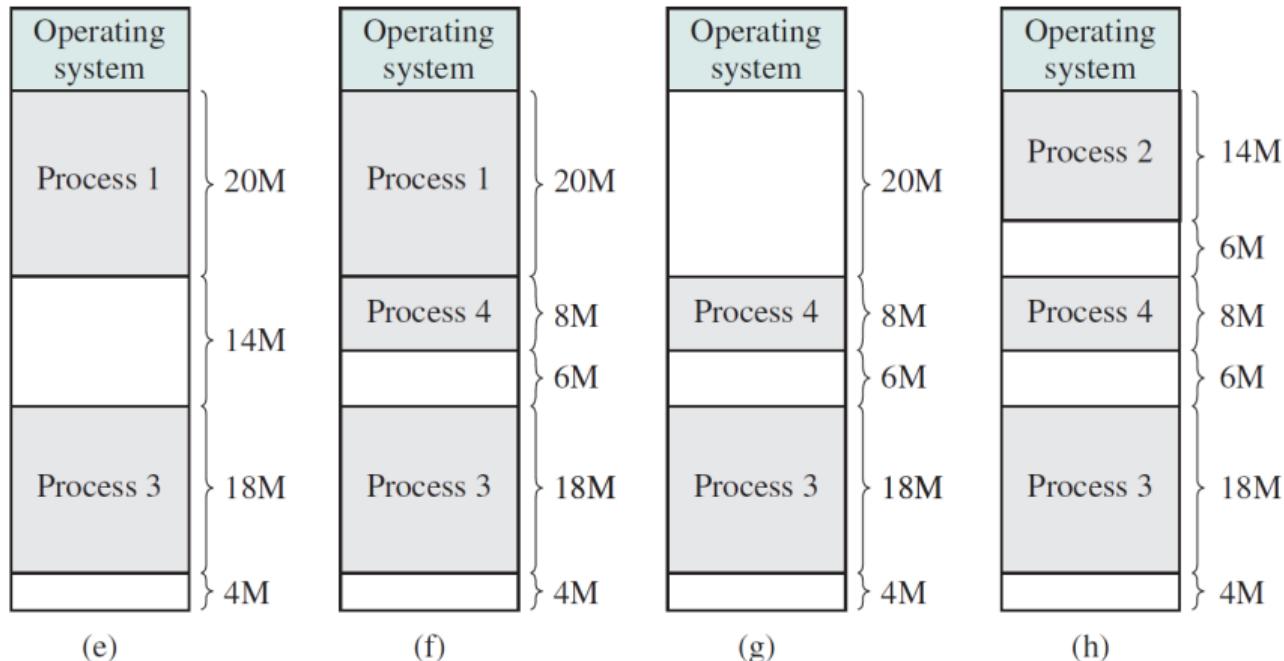


Figure: External fragmentation (from Stallings)

Dynamic Partitioning

Shortcomings

- **External fragmentation:**
 - **Swapping** a process out of memory will create “**a hole**”
 - A new process may not use the entire “hole”, leaving a small **unused block**
- A new process may be **too large** for a given a “hole”
- The **overhead** of memory **compaction** to **recover holes** can be **prohibitive** and requires **dynamic relocation**

Dynamic Partitioning

Shortcomings

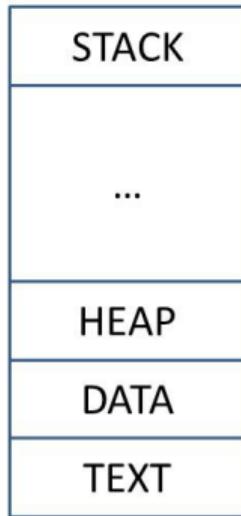


Figure: Memory layout (logical address space)

Dynamic Partitioning

Shortcomings

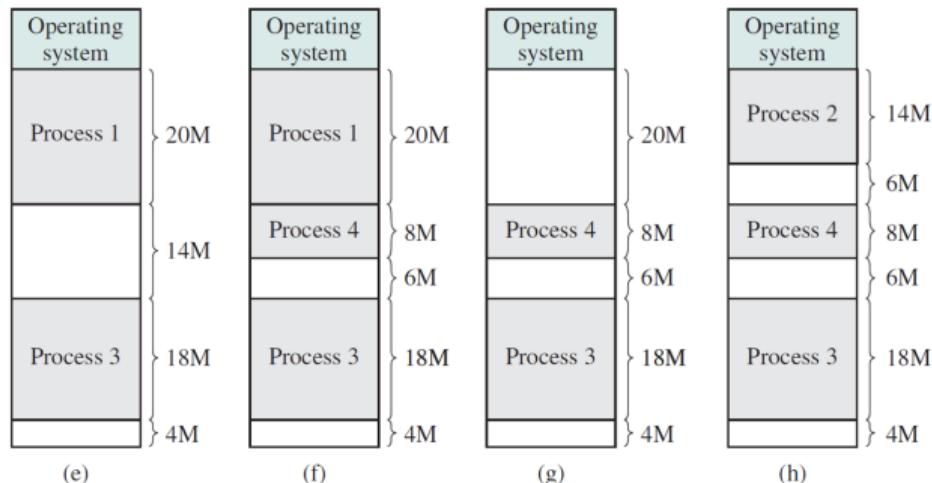


Figure: Fragmentation

Dynamic Partitioning

Shortcomings

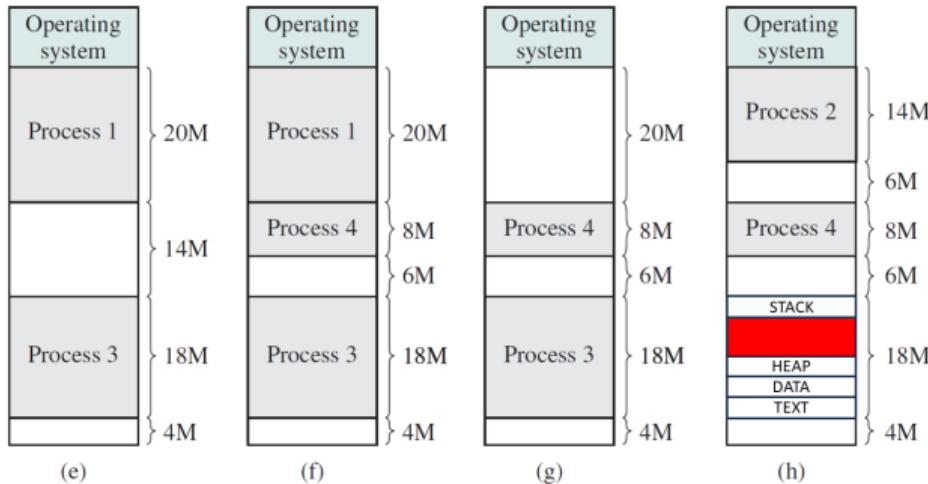


Figure: Fragmentation

Segmentation

Fine Grained Dynamic Partitioning

- **Dynamic partitioning** loads the **entire logical address space** into (**contiguous**) physical memory
 - Uses a **single base and bounds pair** per process
 - Results in **external fragmentation**
 - **Unused space** between stack and heap **takes up physical memory**
- **Segmentation** loads only the **relevant sections** into memory
 - Splits the logical address space into **separate contiguous segments** (code, data, heap, stack)
 - Each segment is **loaded separately** in (contiguous) memory
 - Each segment has a different **base** and **bound** pair
 - The base and bound pair are stored in the **segmentation table**
 - **Part of the logical address** is used as an **index** into the segmentation table

Dynamic Partitioning

Shortcomings

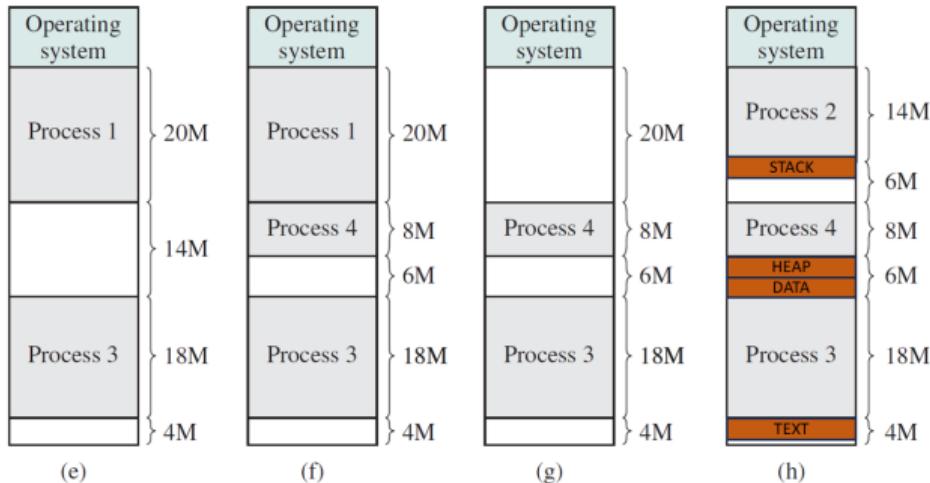


Figure: Segmentation

Segmentation

Segmentation Table

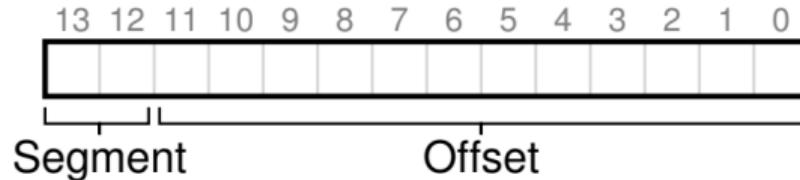


Figure: Logical address for segmentation (offset = position relative to start of segment)

Segment	index	Base	Bound	RWX
Code	00
Data	01
Heap	10
Stack	11

Segmentation

Segmentation Table & Challenges

- Segments can:
 - Have **protection bits** associated with them (RWX)
 - Be **shared** between processes (e.g. code segments)
- **MMU** must use the correct base / bound
- OS remains responsible for finding a sufficiently large **contiguous range of physical memory** for each segment (segments can be **fine-grained**)

Managing Free Space

Approaches

- How to keep track of **available memory**
 - Bitmaps
 - Linked lists
- What strategies can we use to (quickly) **allocate** processes to available memory ("holes")?

Managing Free Space

Allocation Structures: Bitmaps

- The simplest data structure that can be used is a form of **bitmap**
- **Memory is split into blocks** of say 4KB size
 - A bit map is set up so that each **bit is 0** if the **memory block is free** and **1** is the **block is used**, e.g.
 - 32MB memory $\Rightarrow 32 * 2^{20}$ / 4KB blocks $\Rightarrow 32 * 2^8$ bitmap entries (8192)
 - 8192 bits occupy $8192 / 8 = 1\text{KB}$ of storage (only!)
 - The size of this bitmap will depend on the **size of the memory** and **the size of the allocation unit**.

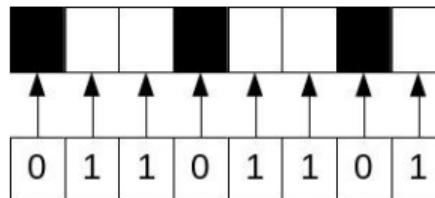


Figure: Memory management with bitmaps

Managing Free Space

Allocation Structures: Bitmaps (Cont'd)

- To find a hole of e.g. size 128KB, then a group of **32 adjacent bits set to zero** must be found, typically a **long operation** (esp. with smaller blocks)
- A **trade-off exists** between the **size of the bitmap** and the **size of blocks** exists
 - The **size of bitmaps** can become prohibitive for small blocks and may **make searching** the **bitmap slower**
 - Larger blocks may increase **internal fragmentation**
- **Bitmaps** are **rarely used** for this reason

Managing Free Space

Allocation Structures: Linked List

- A more **sophisticated data structure** is required to deal with **a variable number of free and used partitions**
- A **linked list** is one such possible data structure
 - A linked list consists of a **number of entries** ("links")!
 - Each link **contains data items**, e.g. **start of memory block**, **size**, free/allocated **flag**
 - Each link also contains a **pointer to the next** in the chain
- The **allocation** of processes to unused blocks becomes **non-trivial**

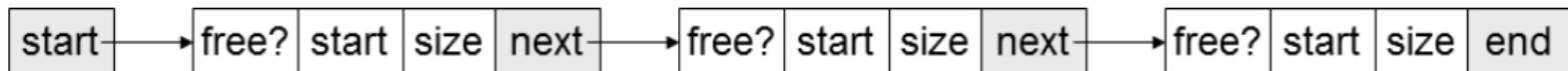


Figure: Memory management with linked lists

Managing Free Space

Allocation Structures

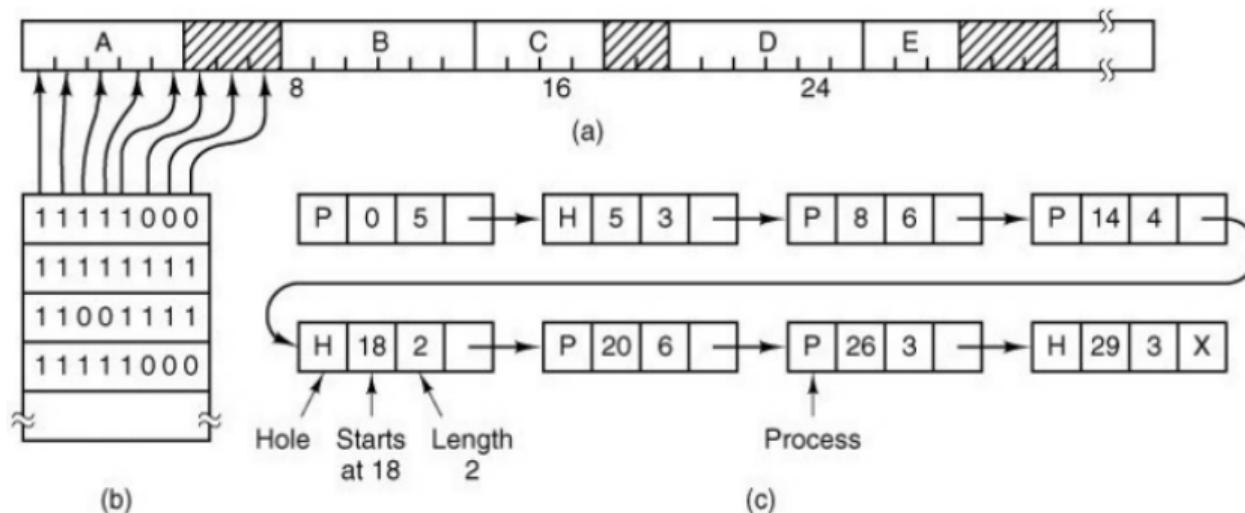


Figure: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list. (from Tanenbaum)

Overview

Goals for Today

- Code **relocation** and **protection**
- **Dynamic partitioning & segmentation**
- **Managing free/occupied memory**

Test your Understanding

Tracking Free Memory

Exercise

Compare the **memory needed** to keep track of **free memory** using **bitmap** vs. **linked list**. The size of main memory is **8GB** and the block size is **1MB**. You can assume that exactly **half of the memory is in use**, and that it contains an **alternating sequence** of occupied and free blocks. The linked list **only keeps track of free blocks**, and each node requires a 32-bit memory address, a 16-bit length, and a 32-bit next-node field.

How many bytes of storage are required for each method?

Operating Systems and Concurrency

Memory Management 3
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recall

Last Lecture

- **Dynamic relocation & protection, base & bound registers (logical \Rightarrow physical address)**
- **Dynamic partitioning & segmentation (internal \Rightarrow external fragmentation)**
- **Free space management (bitmaps and linked lists)**

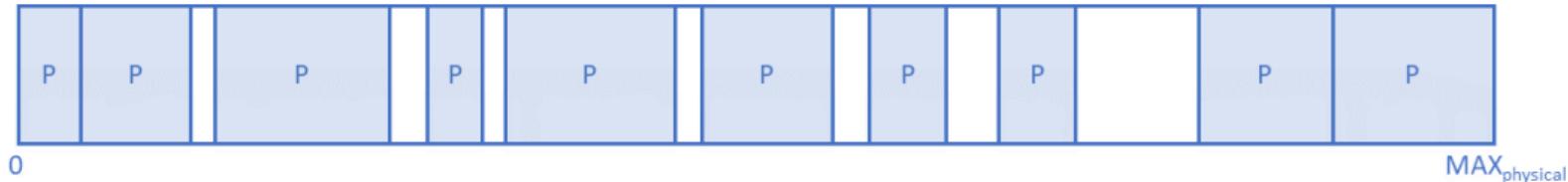
Sketch of the Scene

Where Did we Get To?

- **Contiguous memory allocation** methods:

- Fixed (non-)equal size partitions
- Dynamic partitioning
- Segmentation (individual segments are contiguous)
- `malloc()`, `free()`

- The **amount of memory** requested by processes can be **differ**
- Memory will become **fragmented** over time



Sketch of the Scene

What do we Need?

- A method to **keep track** of free memory
- A strategy to **allocate free** memory when requested by a process
- Approaches to **prevent fragmentation** whenever we can

Overview

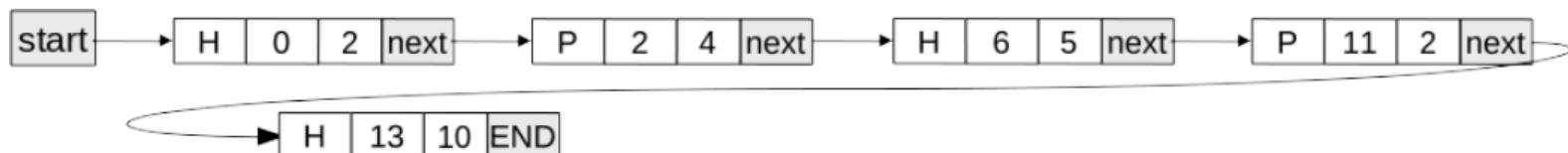
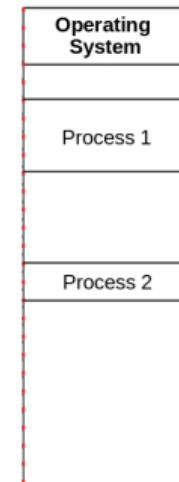
Goals for Today

- **Memory allocation:**
 - First fit, best fit, next fit, worst fit, quick fit
 - Buddy algorithm
- **Prevent fragmentation - non-contiguous** memory management
 - **(Segmentation)**
 - **Paging** - page tables, address translation

Memory Allocation

First Fit

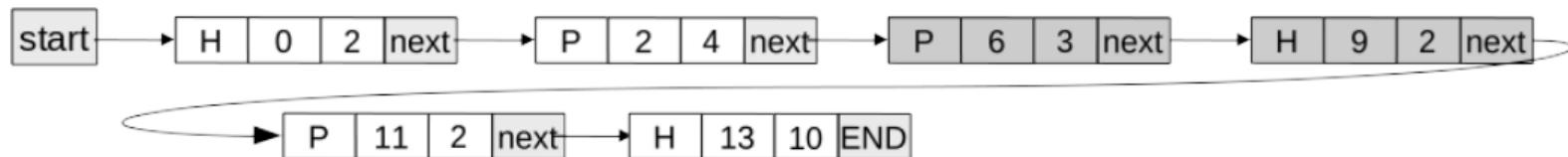
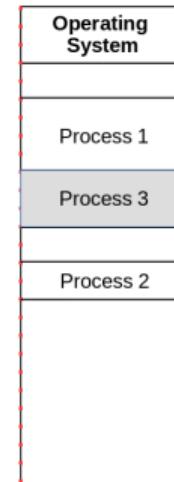
- **First fit** scans **from the start** of the list until an **sufficiently large gap** is found
 - **if** the space is **the exact size** then all the space is **allocated**
 - **else** the space is **split**:
 - The first entry is set to the **size requested** and marked “**used**”
 - The second entry is set to **remaining size** and marked “**free**”



Memory Allocation

First Fit

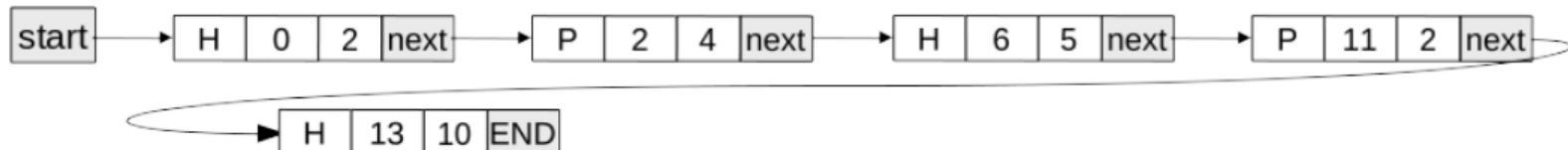
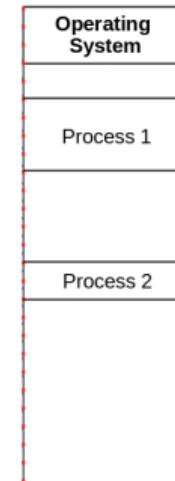
- **First fit** scans **from the start** of the list until an **sufficiently large gap** is found
 - **if** the space is **the exact size** then all the space is **allocated**
 - **else** the space is **split**:
 - The first entry is set to the **size requested** and marked “**used**”
 - The second entry is set to **remaining size** and marked “**free**”



Memory Allocation

Next Fit

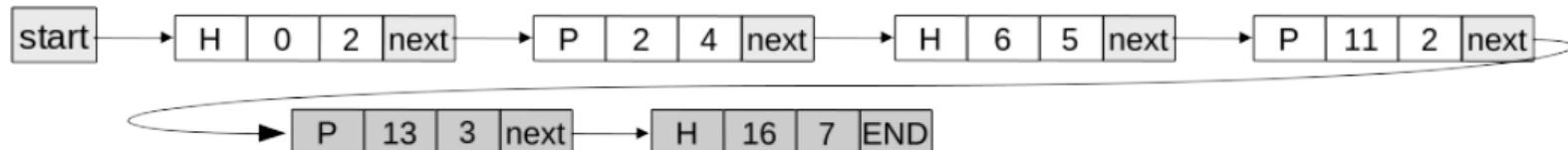
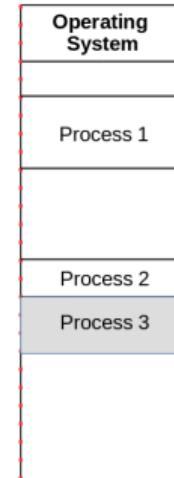
- The **next fit algorithm** maintains a record of where it got to:
 - It **restarts its search from where it stopped last time**
 - It gives an **even chance to all memory** to get allocated (first fit concentrates on the start of the list)
- Simulations have shown that next fit **performs worse** than first fit



Memory Allocation

Next Fit

- The **next fit algorithm** maintains a record of where it got to:
 - It **restarts its search from where it stopped last time**
 - It gives an **even chance to all memory** to get allocated (first fit concentrates on the start of the list)
- Simulations have shown that next fit **performs worse** than first fit



Memory Allocation

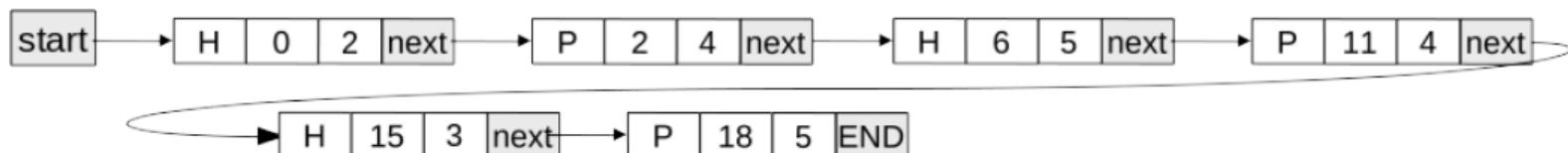
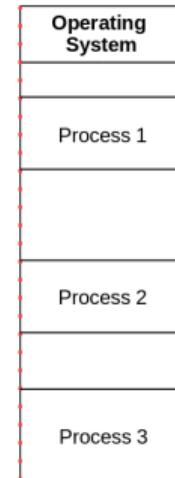
First vs. Next Fit

- **First fit** is a fast and looks for the **first available hole**
 - It does **not account** for a **better fitting** hole later
 - It may **break up a large holes** early in the list
- **Next fit** does not change this

Memory Allocation

Best Fit

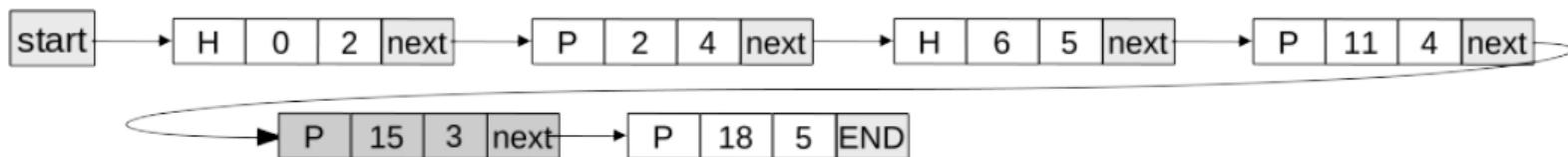
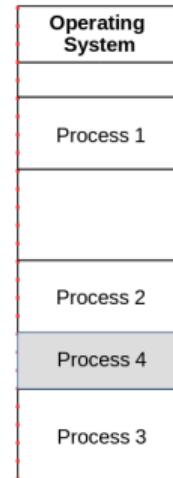
- **Best fit** searches the **entire linked** list for the **smallest hole** big enough to satisfy the request
 - It is **slower** than first fit
 - Result in **small leftover holes** (wastes memory)



Memory Allocation

Best Fit

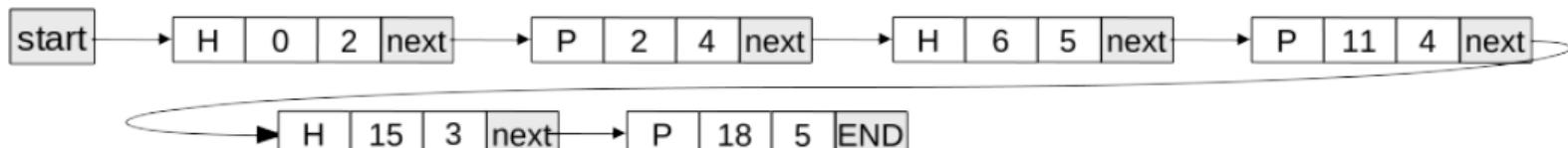
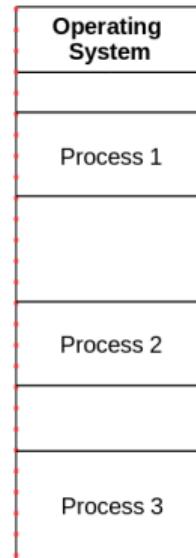
- **Best fit** searches the **entire linked** list for the **smallest hole** big enough to satisfy the request
 - It is **slower** than first fit
 - Result in **small leftover holes** (wastes memory)



Memory Allocation

Worst Fit

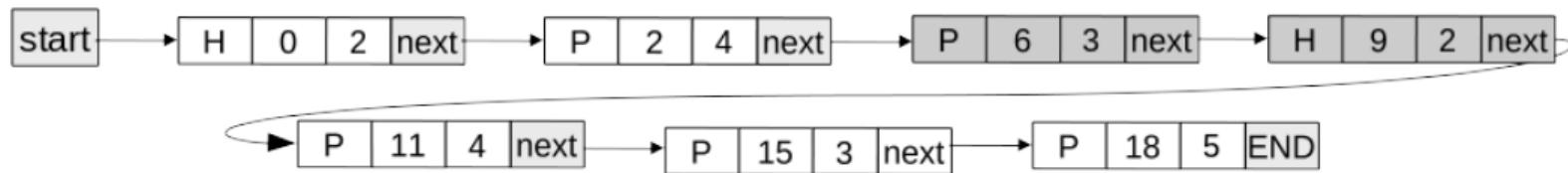
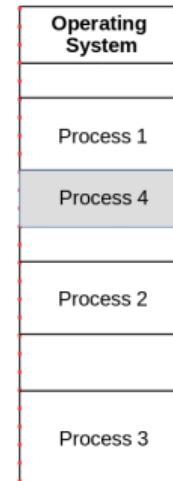
- Best fit split an empty partition in **small holes**
- Worst fit finds the **largest available partition** and splits it
 - The left over part **will still be large** (potentially more useful)
 - Simulations have shown that **worst fit is not good** in practice



Memory Allocation

Worst Fit

- **Tiny holes** are created when **best fit** split an empty partition
- The **worst fit algorithm** finds the **largest available empty partition** and splits it
 - The **left over part will still be large** (and **potentially more useful**)
 - Simulations have also shown that worst fit is **not very good either!**



Memory Allocation

Summary

- **First fit:** allocate **first block** that is **large enough**
- **Next fit:** allocate **next block** that is large enough, i.e. **starting from the current location**
- **Best fit:** choose block that **matches** required size **closest** - $O(N)$ complexity
- **Worst fit:** choose the **largest possible block** - $O(N)$ complexity

Dynamic Partitioning

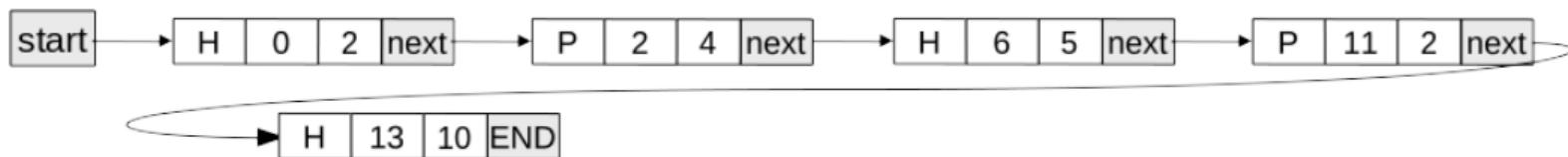
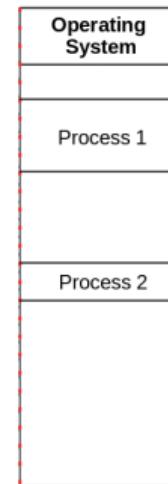
Allocating Available Memory: Quick Fit and Others

- **Quick fit** maintains **lists of commonly used sizes**
 - For example a separate list for each of 4K, 8K, 12K, 16K, etc., holes
 - **Odd sizes** can either go into the **nearest size** or into a **special separate list**
- It is **much faster** to find the required size hole using **quick fit**
- Similar to **best fit**, it has the problem of creating **many tiny holes**
- Finding neighbours for **coalescing** (combining empty partitions) becomes more difficult/time consuming

Memory Allocation

Coalescing

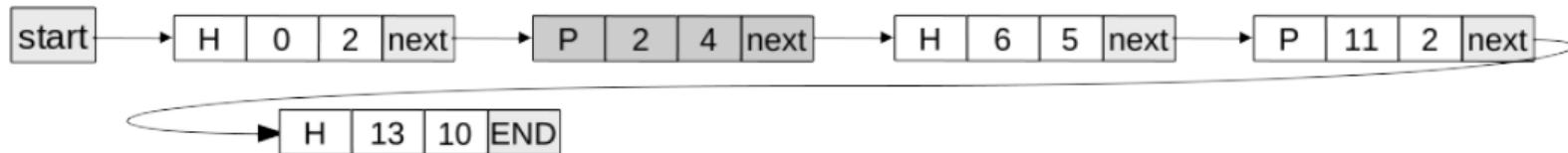
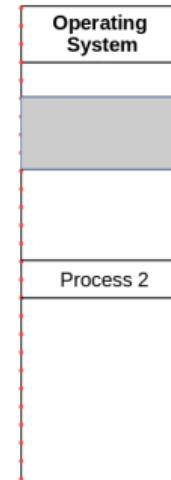
- **Coalescing** (joining together) takes place when two **adjacent entries** in the linked list **become free**
- Both **neighbours** are examined when a **block is freed**
 - if either (or both) are **also free**
 - then the two (or three) **entries are combined** into one larger block by adding up the sizes
 - The earlier block in the linked list gives the **start point**
 - The **separate links are deleted** and a single link inserted



Memory Allocation

Coalescing

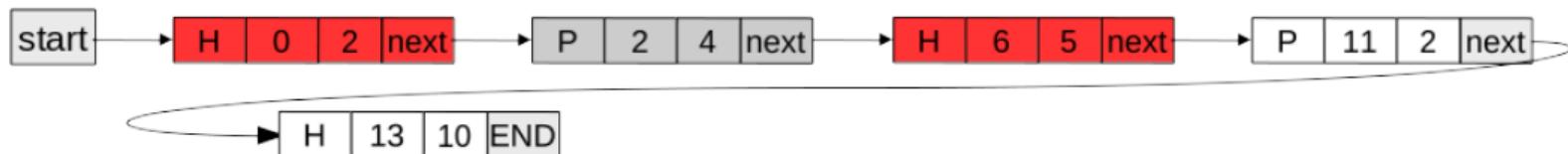
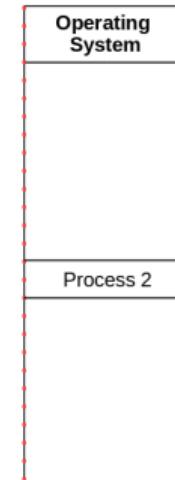
- **Coalescing** (joining together) takes place when two **adjacent entries** in the linked list **become free**
- Both **neighbours** are examined when a **block is freed**
 - if either (or both) are **also free**
 - then the two (or three) **entries are combined** into one larger block by adding up the sizes
 - The earlier block in the linked list gives the **start point**
 - The **separate links are deleted** and a single link inserted



Memory Allocation

Coalescing

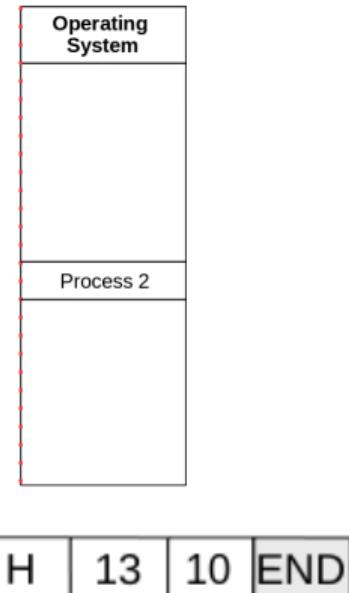
- **Coalescing** (joining together) takes place when two **adjacent entries** in the linked list **become free**
- Both **neighbours** are examined when a **block is freed**
 - if either (or both) are **also free**
 - then the two (or three) **entries are combined** into one larger block by adding up the sizes
 - The earlier block in the linked list gives the **start point**
 - The **separate links are deleted** and a single link inserted



Memory Allocation

Coalescing

- **Coalescing** (joining together) takes place when two **adjacent entries** in the linked list **become free**
- Both **neighbours** are examined when a **block is freed**
 - if either (or both) are **also free**
 - then the two (or three) **entries are combined** into one larger block by adding up the sizes
 - The earlier block in the linked list gives the **start point**
 - The **separate links are deleted** and a single link inserted



Memory Allocation

Compacting

- Even with coalescing, **free blocks** may still **distributed** across memory
- **Compacting** can be used but is harder to implement and **time consuming**
- Processes may have to be **moved/swapped out**, **free space coalesced**, and processes **swapped back in** at lowest available location

Contiguous Allocation Schemes

Overview and Shortcomings

- Different contiguous memory allocation schemes have different advantages/disadvantages
 - **Mono-programming** is easy but does result in **low resource utilisation**
 - **Fixed partitioning** facilitates **multi-programming** but results in **internal fragmentation**
 - **Dynamic partitioning & segmentation** facilitate **multi-programming**, reduce **internal fragmentation**, but result in **external fragmentation** (allocation methods, coalescing, and compacting help)
- Can we design a memory management scheme that **resolves the shortcomings of contiguous memory schemes?**

Paging

Principles

- **Paging** uses **fixed partitioning** and **code re-location** to devise a new ***non-contiguous*** management scheme:
 - Memory is split into **much smaller blocks**
 - A process is allocated **one or more blocks** (e.g., a 11KB process would take up three 4KB blocks)
 - Blocks do **not** have to be stored **in contiguous in physical memory**
 - The process **perceives** them to be **contiguous**
- Benefits of non-contiguous schemes include:
 - **Internal fragmentation** is reduced to the **last “block”** only
 - **No external fragmentation** (blocks in physical address space are **stacked** directly on top of each other)

Paging

Principles (Cont'd)

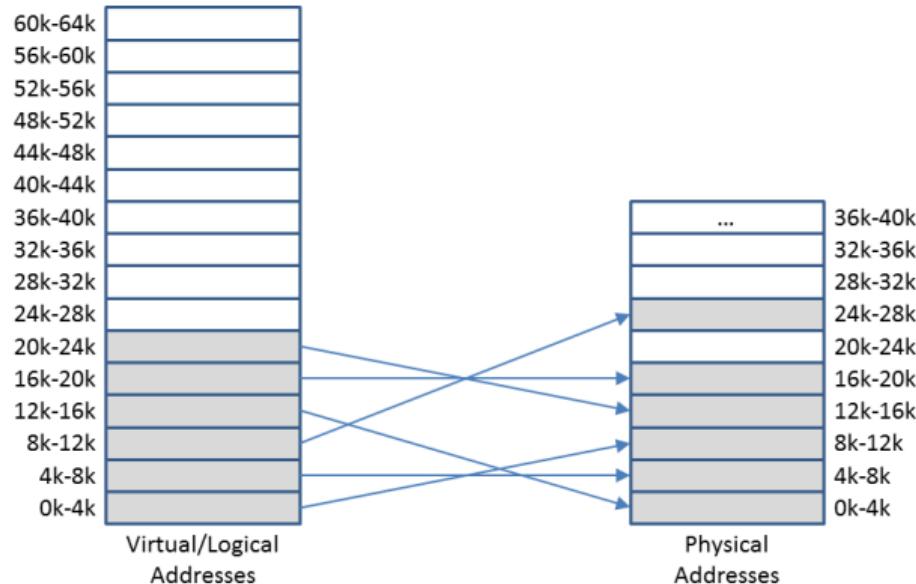


Figure: Paging in physical memory with multiple processes

Paging

Principles (Cont'ed)

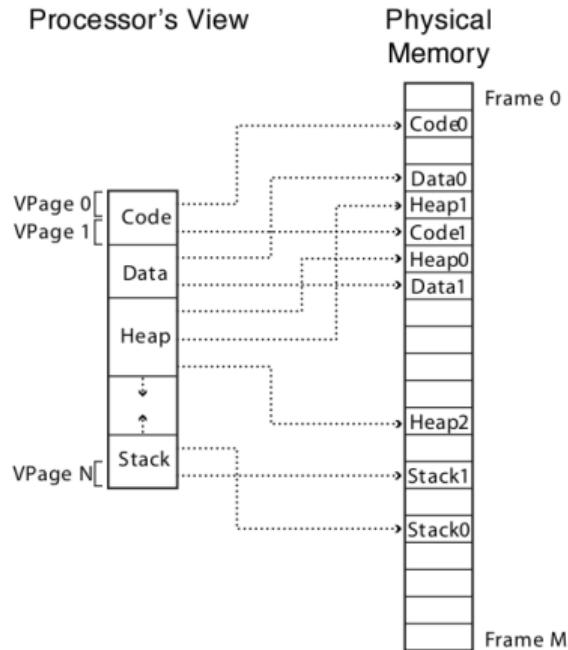


Figure: Paging in main memory with multiple processes (Anderson)

Paging

Principles (Cont'ed)

Main memory	
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(d) Load process C

Main memory	
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(e) Swap out B

Main memory	
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

(f) Load process D

Figure: Concept of Paging (Stallings)

Paging

Principles: Definitions

- A **page** is a small block of **contiguous memory** in the **logical address space**, i.e. as seen by the process
- A **frame** is a **small contiguous block** in **physical memory**
- Pages and frames (commonly) have the **same size**:
 - The size is usually a power of 2
 - Sizes range between 512 bytes and 1Gb

Paging

Relocation

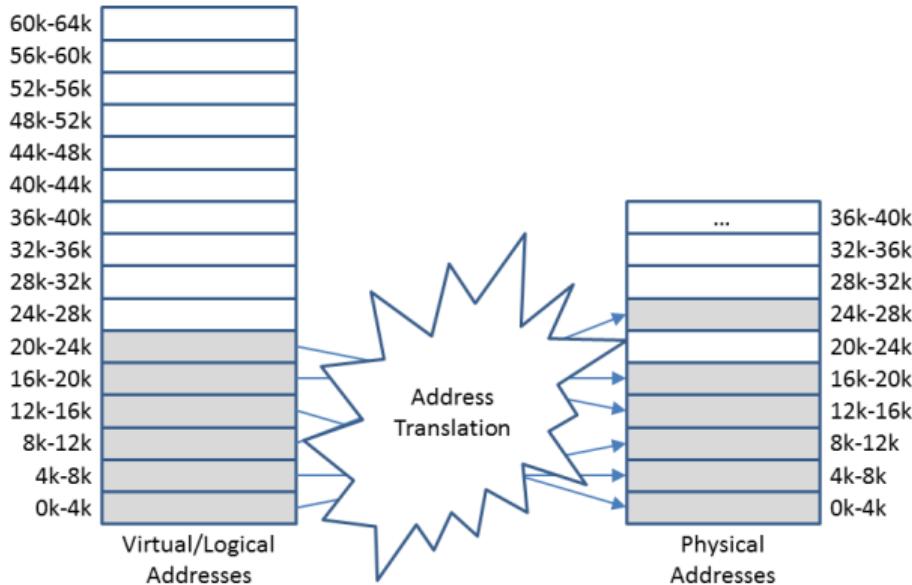


Figure: Address Translation

Segmentation

Segmentation Table

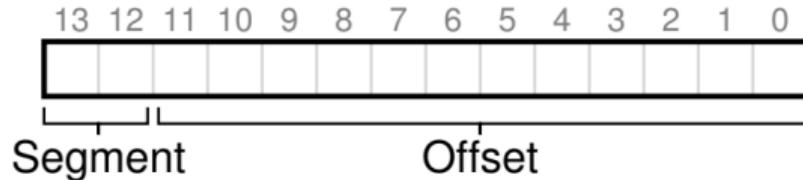


Figure: Logical address for segmentation (offset = position relative to start of segment)

Segment	index	Base	Bound	RWX
Code	00
Data	01
Heap	10
Stack	11

Paging

Page Table

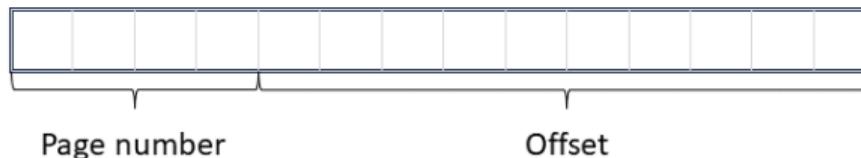


Figure: Logical address for page (offset = position relative to start of segment)

Page Number	Frame Number	RWX
0000	1010
0001	0100
0010	1111
...

Paging

Page Table



Figure: Logical address for page (offset = position relative to start of segment)

Page Number	Frame Number	RWX
0000	1010
0001	0100
0010	1111
...

Paging

Page Table



Figure: Logical address for page (offset = position relative to start of segment)

Page Number	Frame Number	RWX
0000	1010
0001	0100
0010	1111
...

Paging

Page Table



Figure: Logical address for page (offset = position relative to start of segment)

Page Number	Frame Number	RWX
0000	1010
0001	0100
0010	1111
...

Paging

Page Table



Figure: Logical address for page (offset = position relative to start of segment)

Page Number	Frame Number	RWX
0000	1010
0001	0100
0010	1111
...

Paging

Relocation

- **Logical address** (page number, offset within page) needs to be **translated** into a **physical address** (frame number, offset within frame)
- **Multiple “base registers”** will be required:
 - Each **page** has a “**base register**” that identifies the start of the associated **frame**
 - A **set of base registers** must be maintained for every process
- The **set of base registers** is stored in the **page table**

Paging

Relocation: Address Translation

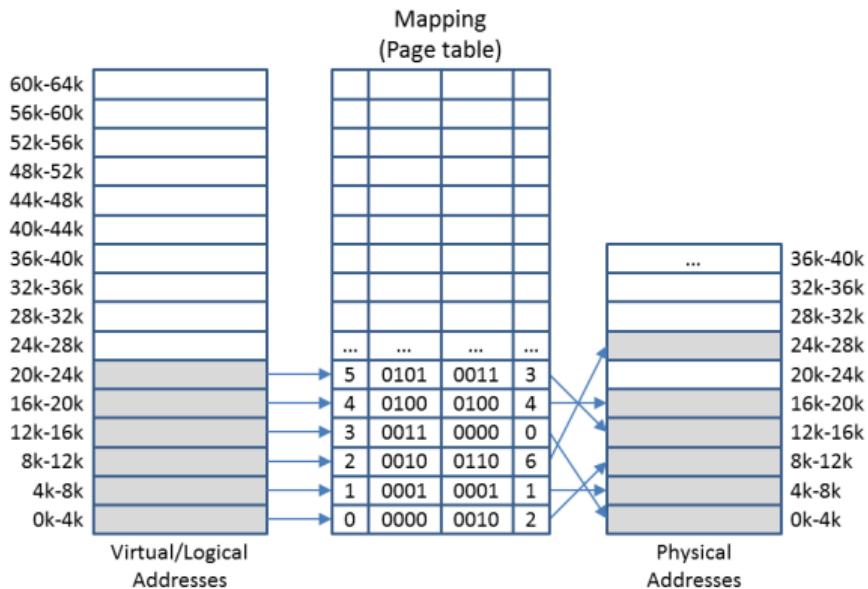


Figure: Address Translation

Paging

Relocation: Page Tables

- The page table maps the **page number** onto the **frame number** (logical \Rightarrow physical address)
 - $\text{frameNumber} = f(\text{pageNumber})$
- The **page number** is used as **index to the page table** that lists the **number of the associated frame**
- From the **frame number** one can calculate the base (offset remains unchanged) is used as **index to the page table** that lists the **number of the associated frame**
- Every process has its **own page table** containing its own **set of “base registers”**
- The **operating system** maintains a **list of free frames**

Paging

Address Translation: Implementation

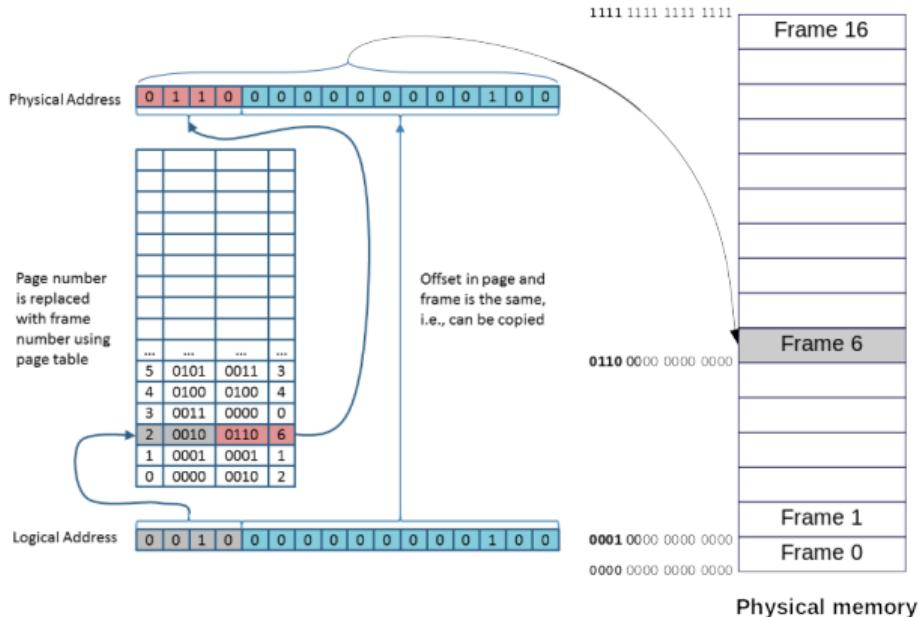


Figure: Address Translation

Test Your Understanding

Page tables and memory addressing

Page Tables

Given a **64-bit machine** that uses paging, and a page/frame size of **4KB**.

- What would be the maximum **number of frames?** $2^{64} / (4 * 1024)$
- How many **entries** will we have in the **page table?** $2^{64} / (4 * 1024)$
- How many **pages** do we have in a **17KB process?**
- How much **memory is wasted** in the last frame?

Recap

Take-Home Message

- Memory **allocation**, **coalescing** and **compacting** in dynamic partitioning
- **Paging**, **page tables**, and **address translation**

COMP2007: Operating Systems & Concurrency
Week 8 – 3:00pm Monday – 13 November 2023



valid for 65 minutes from 2:55pm
generated 2023-10-17 03:13

Figure: Attendance Monitoring

Operating Systems and Concurrency

Memory Management 4
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Goals for Today

Overview

- **Address translation** implementation (revisited)
- Principles behind **virtual memory**
- Complex/large **page tables**

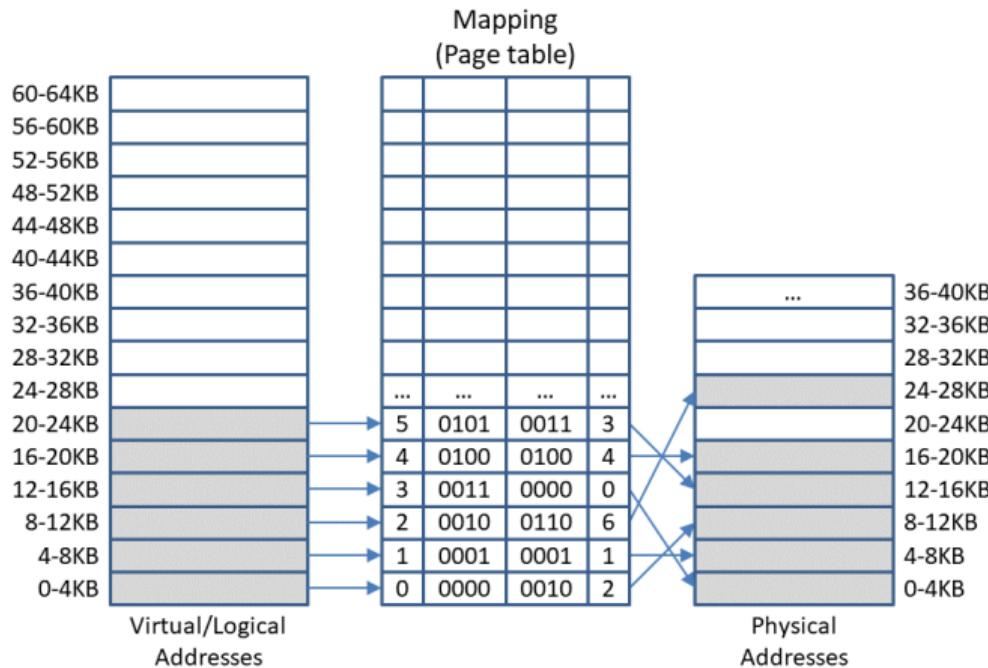
Recall

Last Lecture

- The **principles of paging** are:
 - **Logical address space** is divided into equal sized **pages**
 - **Physical address space** is divided into equal sized **frames**
 - A **page table** contains multiple “**relocation registers**” to map the pages on to frames
- The **benefits** of paging include:
 - Reduced **internal fragmentation**
 - No **external fragmentation**

Paging

Relocation: Address Translation



Recall

Memory as a Linear Array

- Memory is a **linear array of bytes**
- N **address lines** are used to specify 2^N addresses, e.g., 2^{16} for a 16 bit machine
- If each **memory cell** is a **byte**, we can address up to 64KB.
- If the memory is split into 16 blocks ($=2^4$), then block size = $2^{16}/2^4 = 2^{12} = 4\text{KB}$

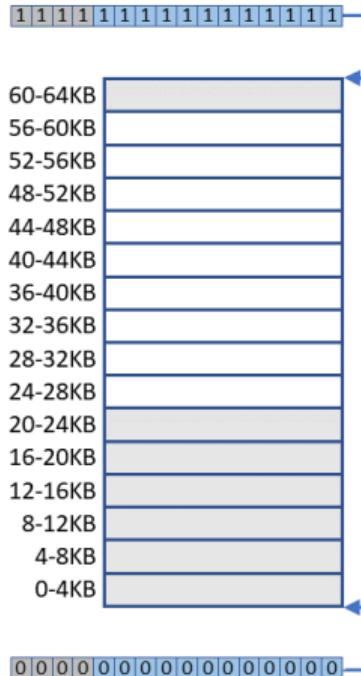
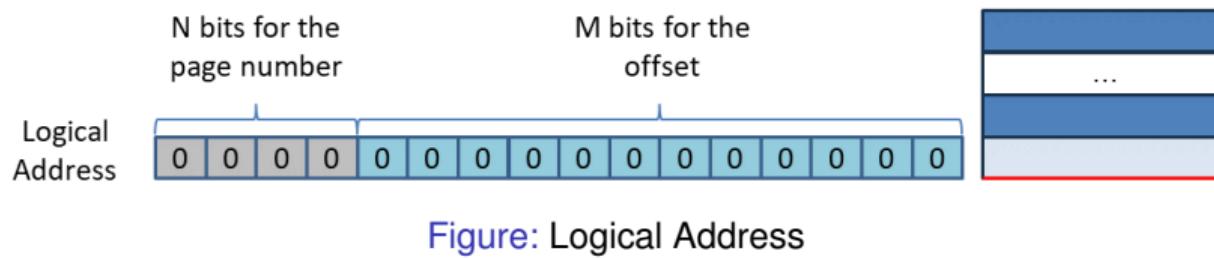


Figure: Linear address space (16-bit)

Paging

Address Translation: Implementation

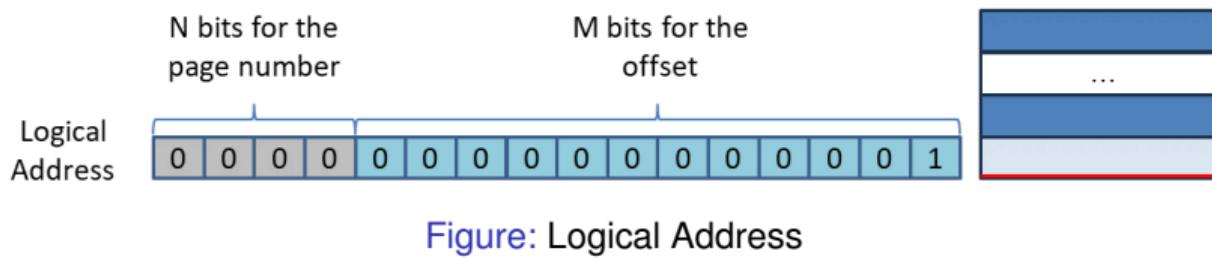
- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages

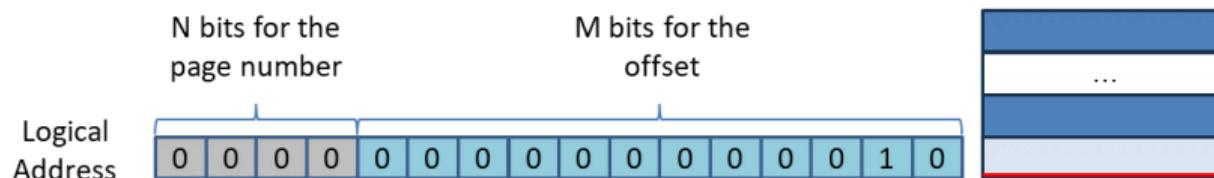


Figure: Logical Address

Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages

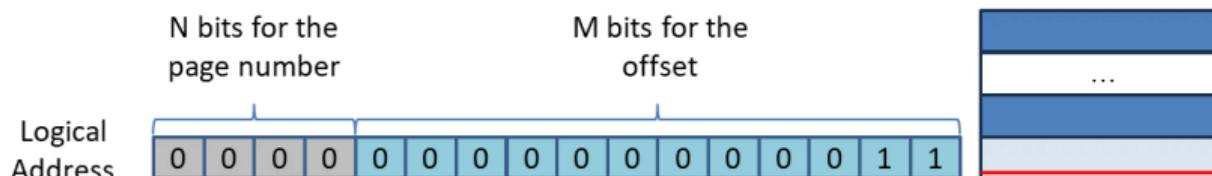


Figure: Logical Address

Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages

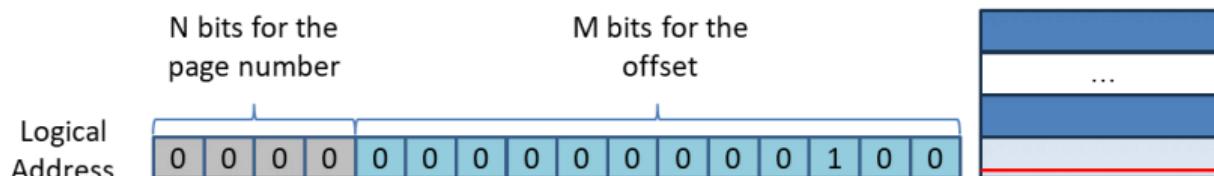
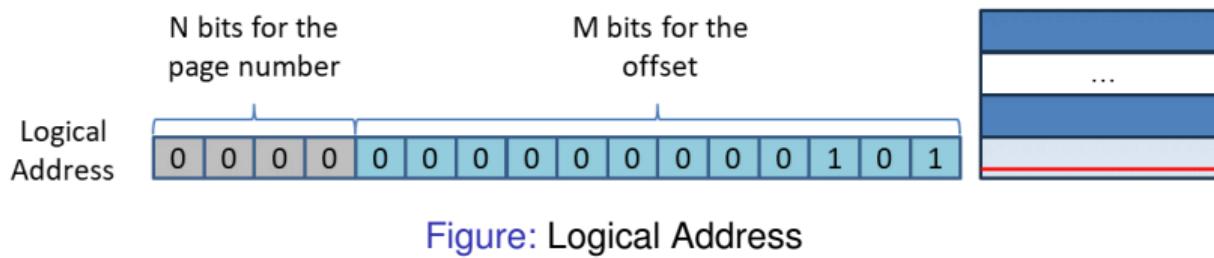


Figure: Logical Address

Paging

Address Translation: Implementation

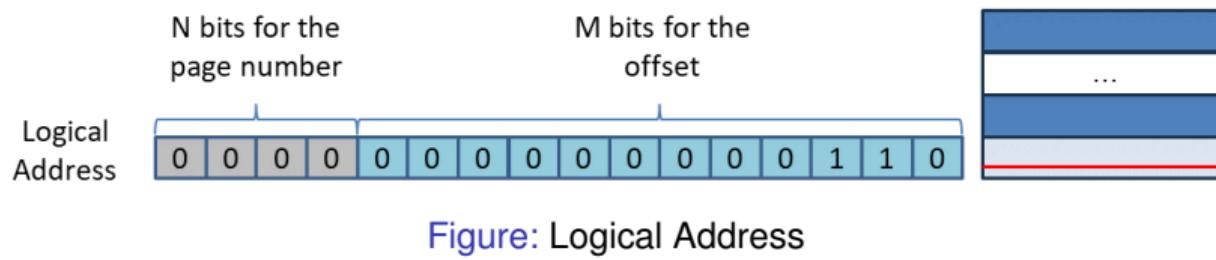
- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

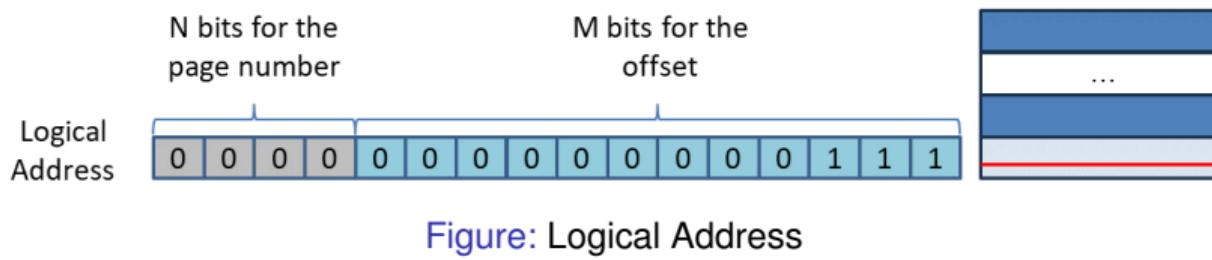
- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

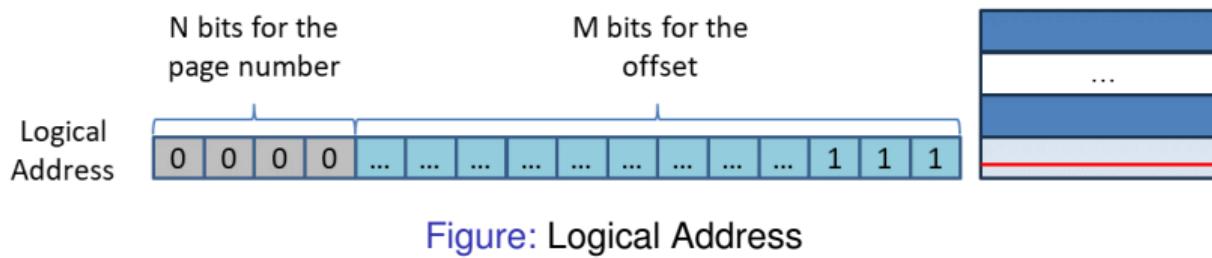
- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

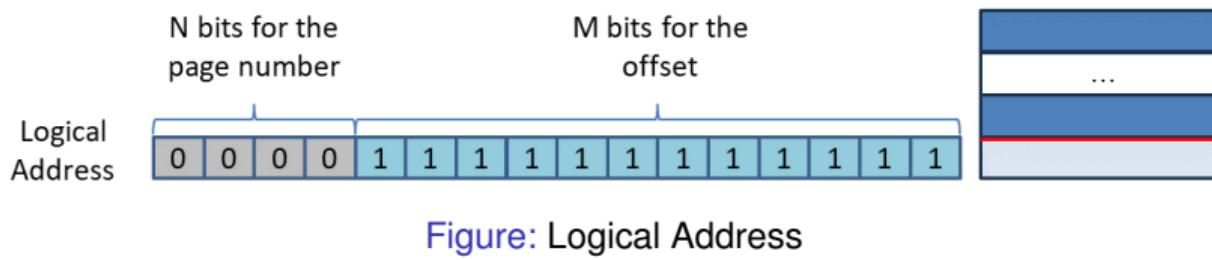
- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages

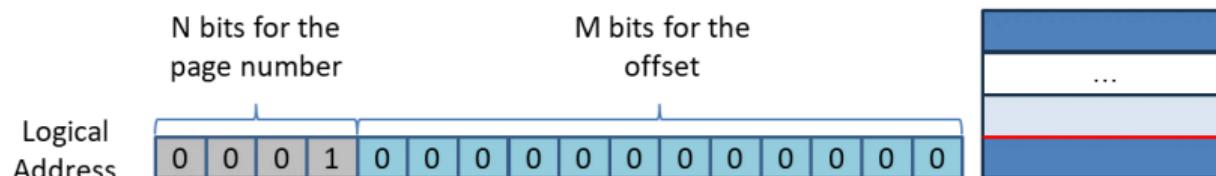


Figure: Logical Address

Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages

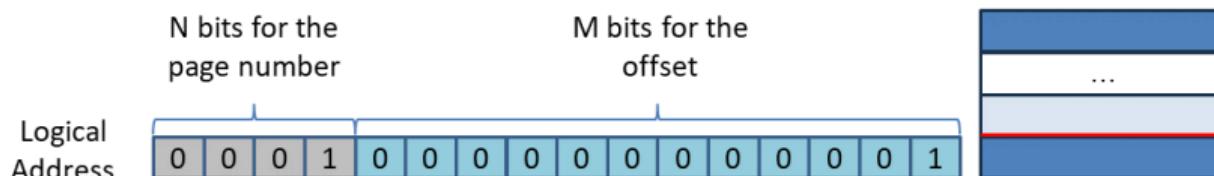


Figure: Logical Address

Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages

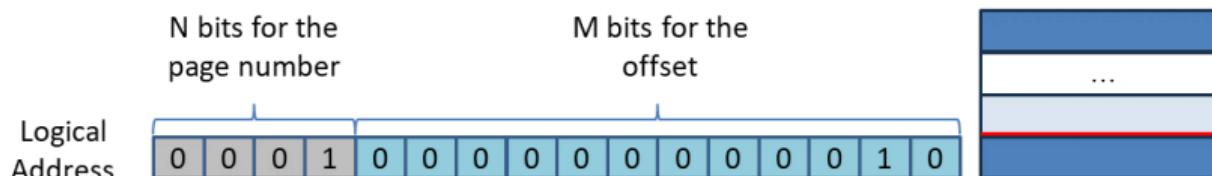
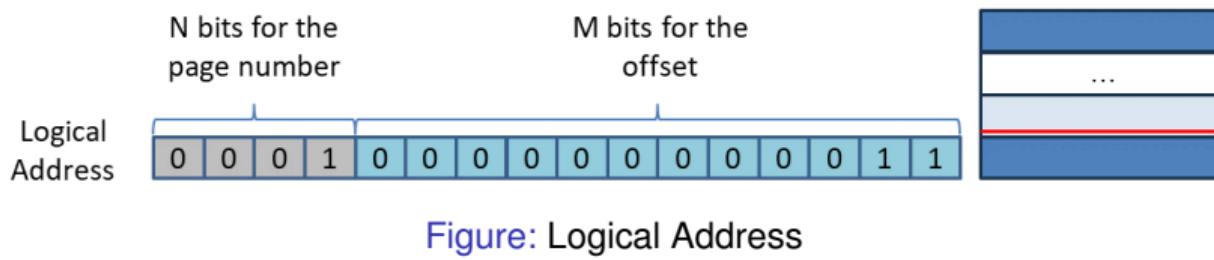


Figure: Logical Address

Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages

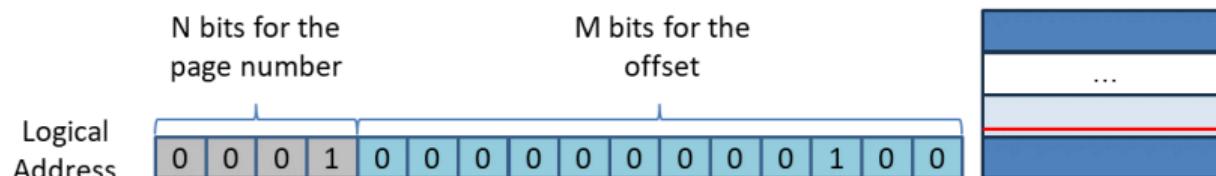
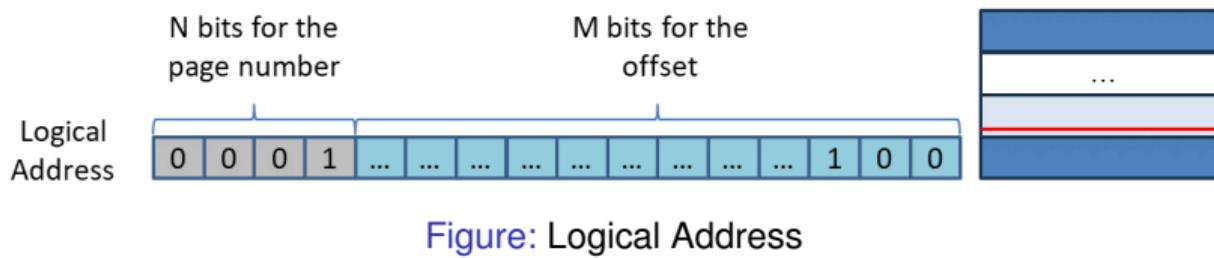


Figure: Logical Address

Paging

Address Translation: Implementation

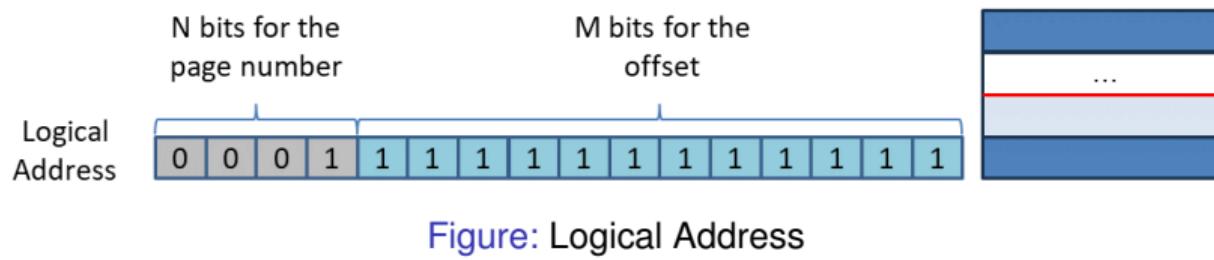
- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

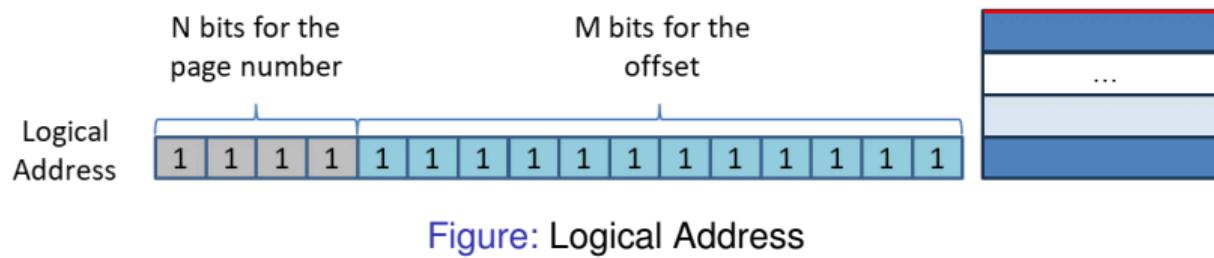
- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

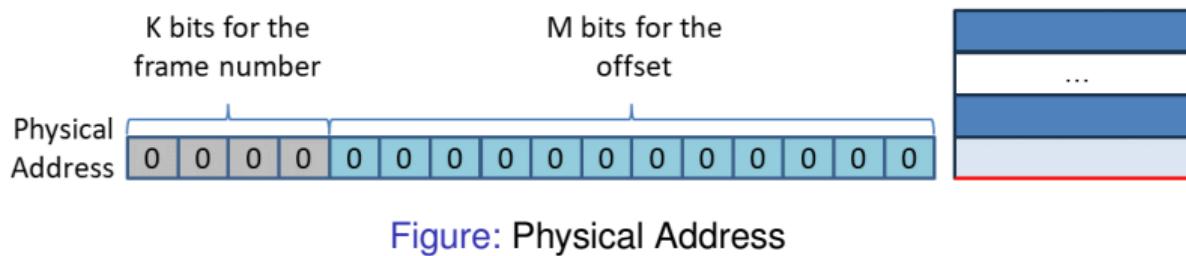
- A **logical address** is relative to the start of the **program**
- The **left most n bits** represent the **page number**
 - e.g. 4 bits for the page number \Rightarrow 16 pages
- The **right most m bits** represent the **offset within the page**
 - e.g. 12 bits for the offset \Rightarrow 4KB pages



Paging

Address Translation: Implementation

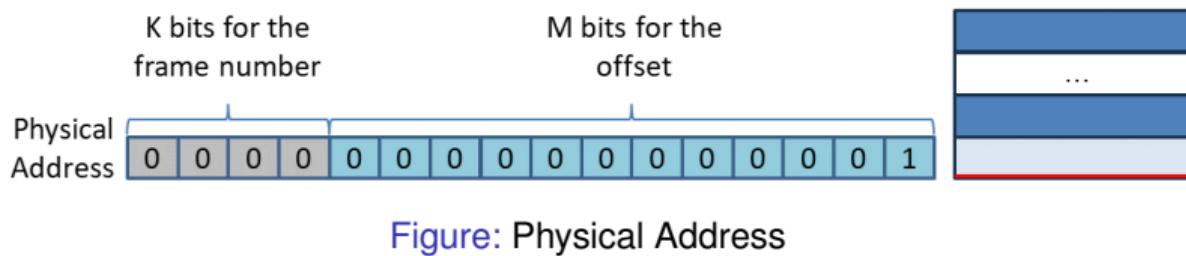
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

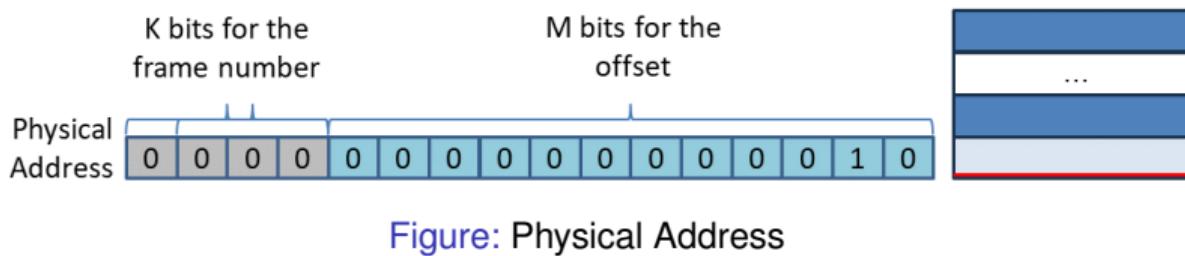
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

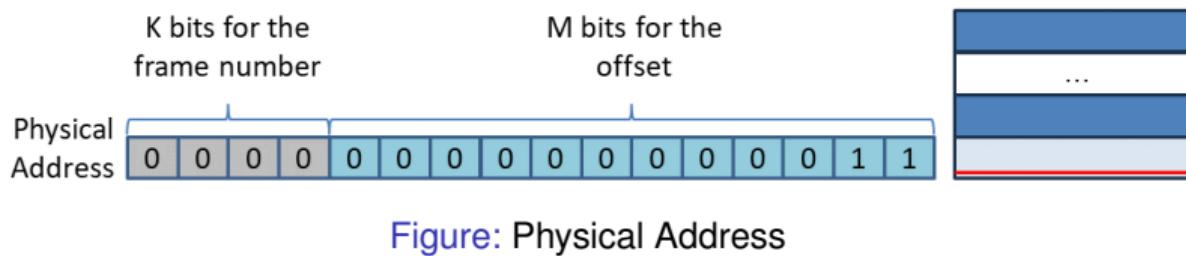
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

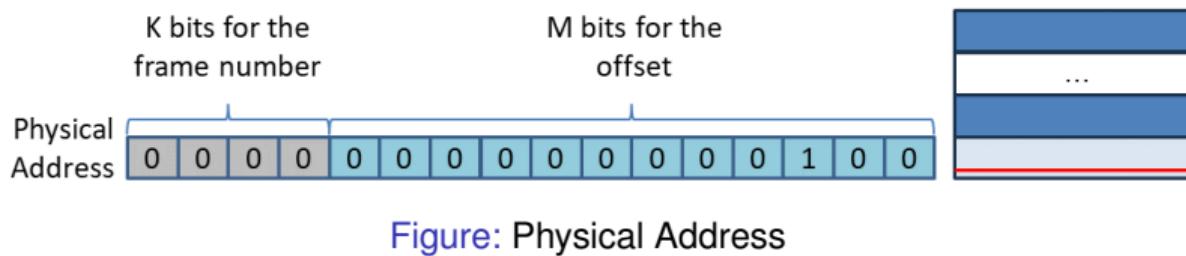
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

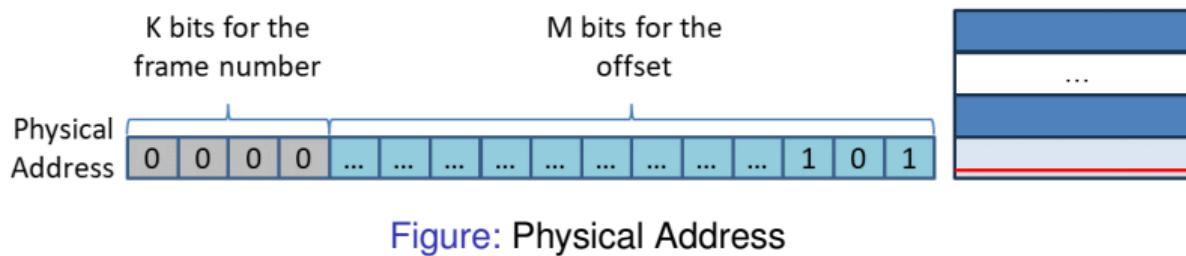
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

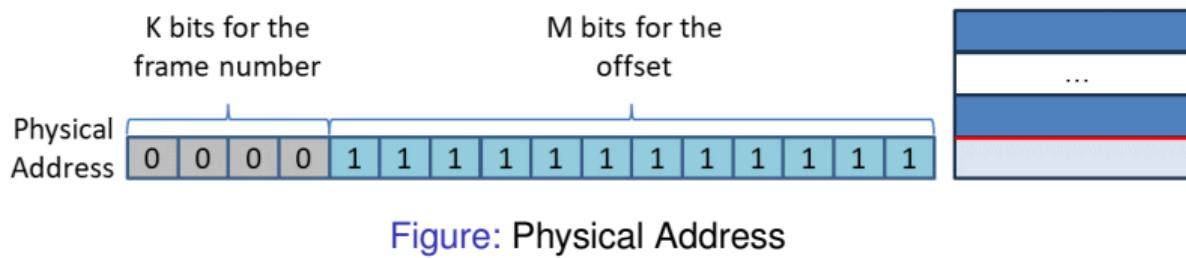
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

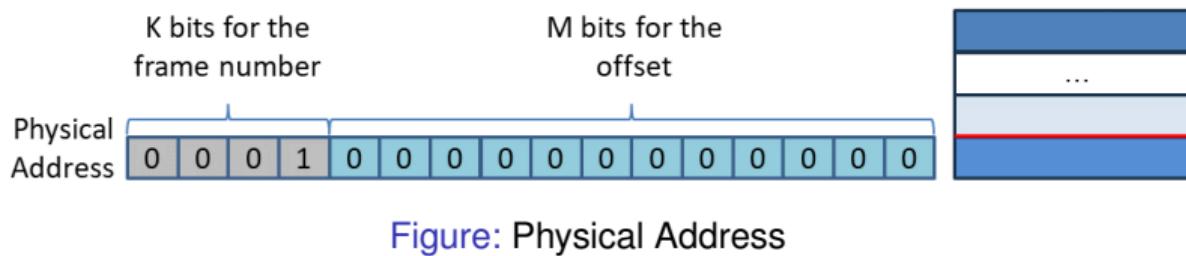
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

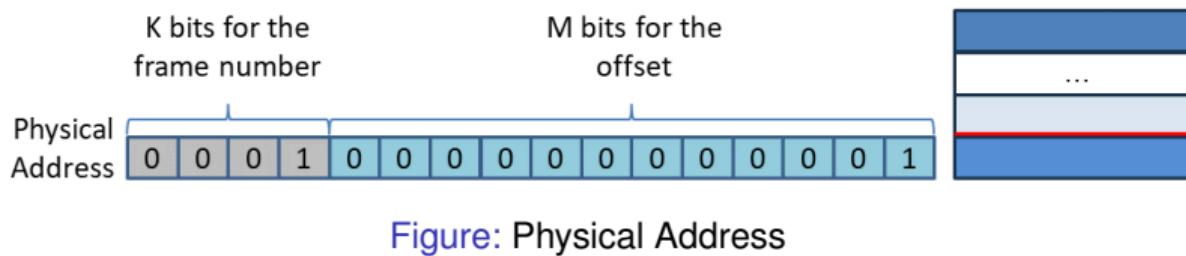
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

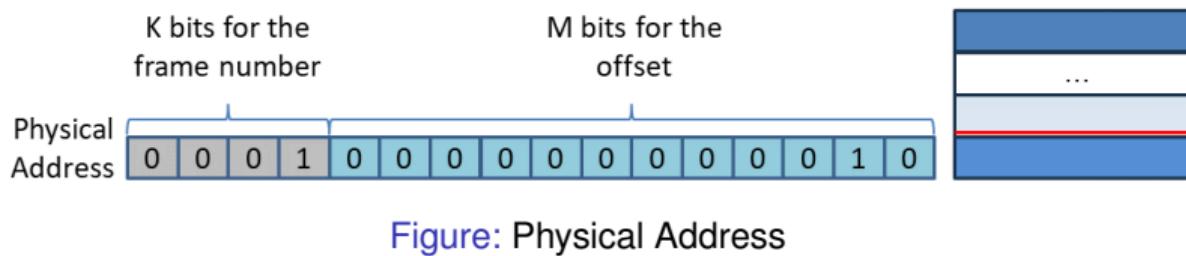
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

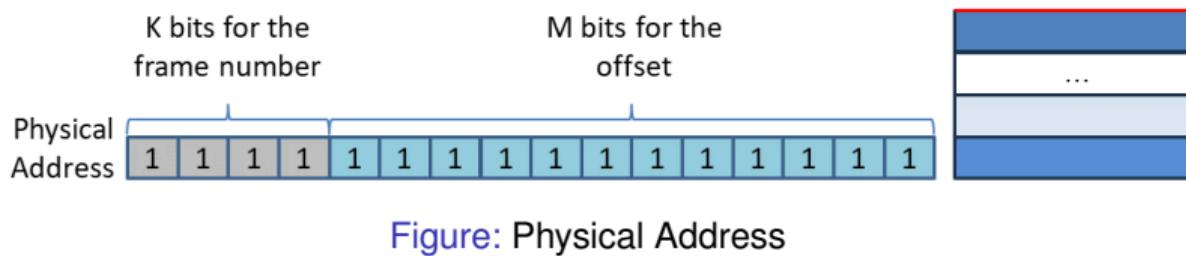
- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

- A **physical** is relative to the start of the **memory**
- The **left most k bits** represent the **frame number**
 - e.g. 3 bits for the frame number \Rightarrow 8 frames
- The **right most m bits** represent the **offset within the frame**
 - e.g. 12 bits for the offset \Rightarrow 4KB frames



Paging

Address Translation: Implementation

- The **offset** within the page and frame **remains the same** (they are the same size)
- The page number to frame number mapping is held in the **page table**

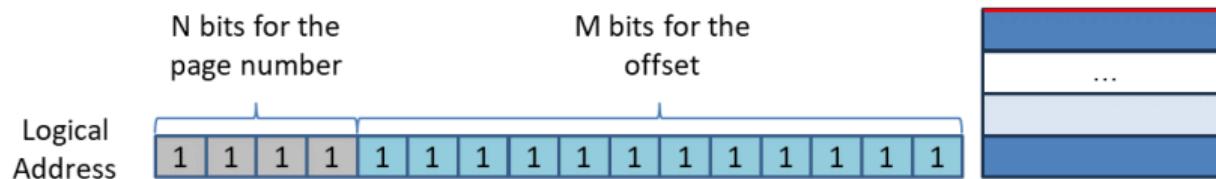


Figure: Logical Address

Paging

Address Translation: Implementation

- The **offset** within the page and frame **remains the same** (they are the same size)
- The page number to frame number mapping is held in the **page table**

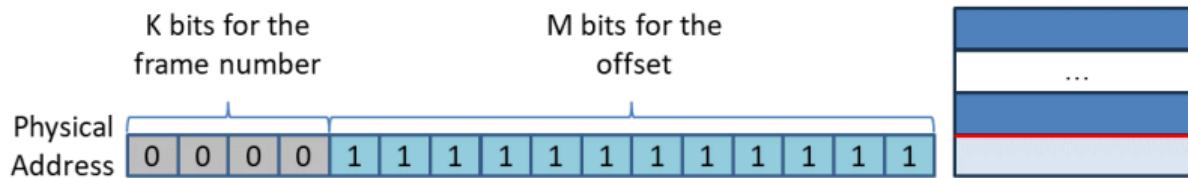
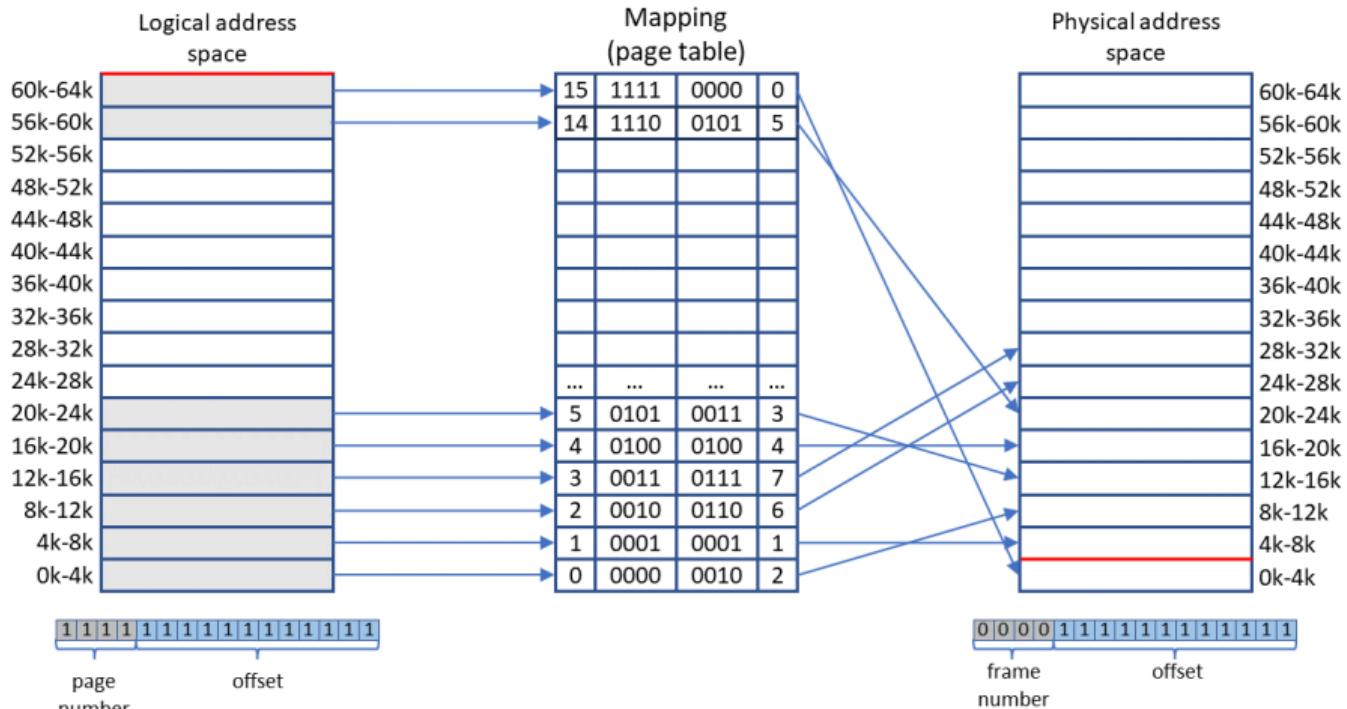


Figure: Logical Address

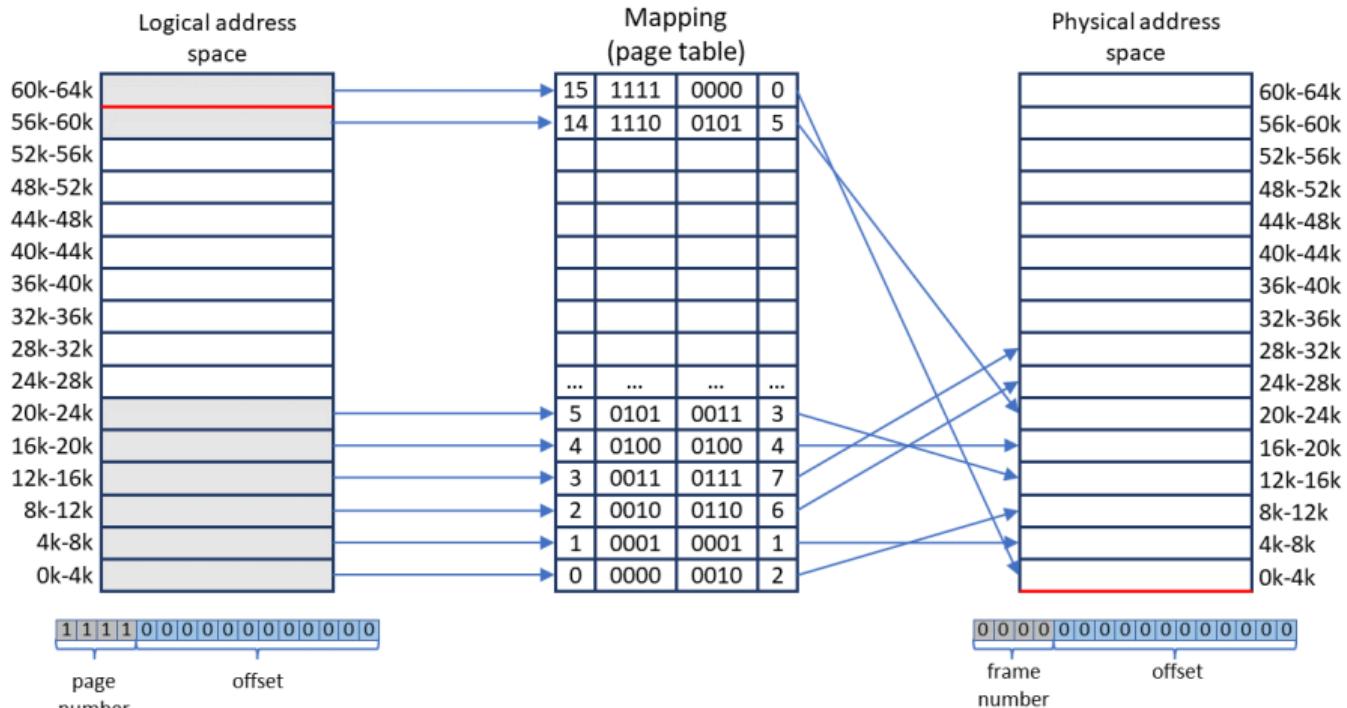
Paging

Address Translation: Implementation



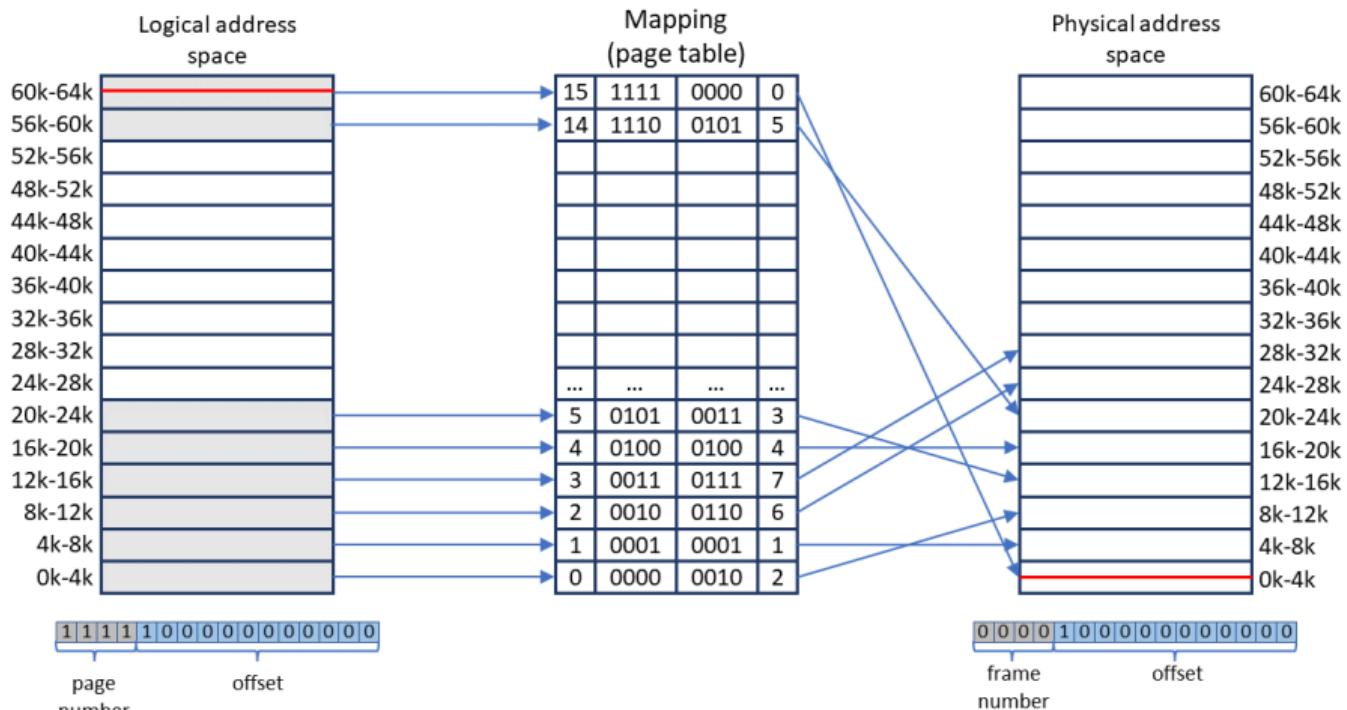
Paging

Address Translation: Implementation



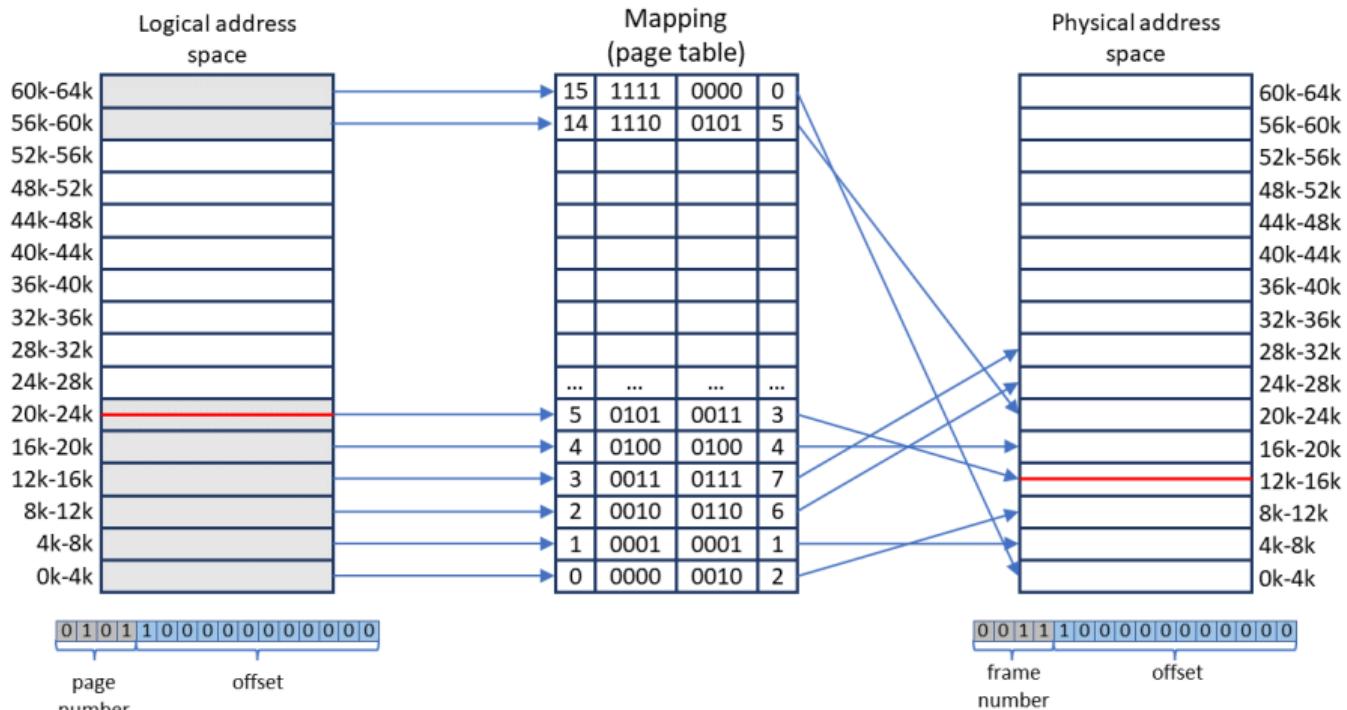
Paging

Address Translation: Implementation



Paging

Address Translation: Implementation



Paging

Relocation: Address Translation

- Steps in **address translation**:
 - ① Extract the **page number** from logical address
 - ② Use page number as an index to **retrieve the frame number** in the page table
 - ③ Add the “logical offset within the page” to the start of the physical frame
- **Hardware implementation** of address translation
 - ① The CPU’s **memory management unit** (MMU) intercepts logical addresses
 - ② MMU uses a page table as above
 - ③ The resulting **physical address** is put on the **memory bus**

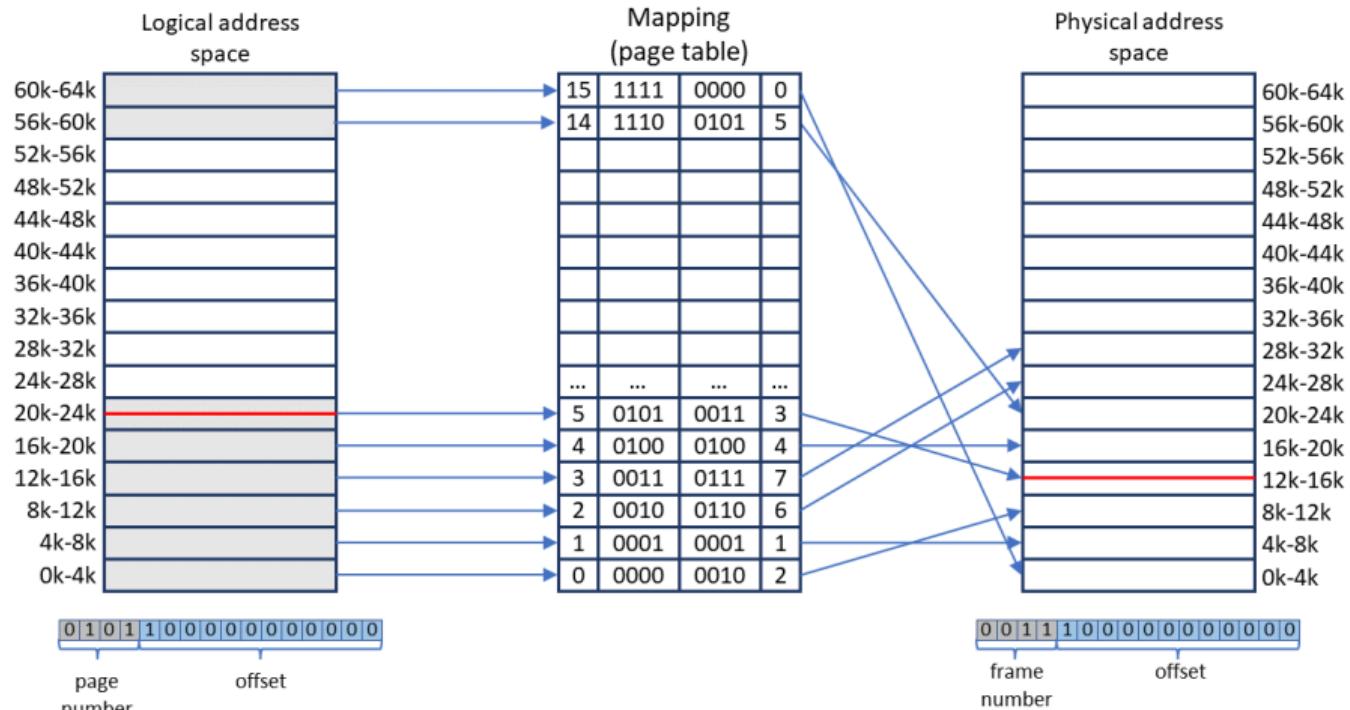
Virtual Memory

Principle of Locality

- Are there any other **benefits of paging?**
 - Principle of **locality**: **code** and **data references** are usually **clustered**
 - Code execution and data manipulation is usually restricted to a small **subset of pages** at a given point in time
- **Not all pages** have to be **loaded** in memory at the same time ⇒ **virtual memory**
 - Loading all program/data pages into memory is **wasteful**
 - Desired blocks could be **loaded on demand**

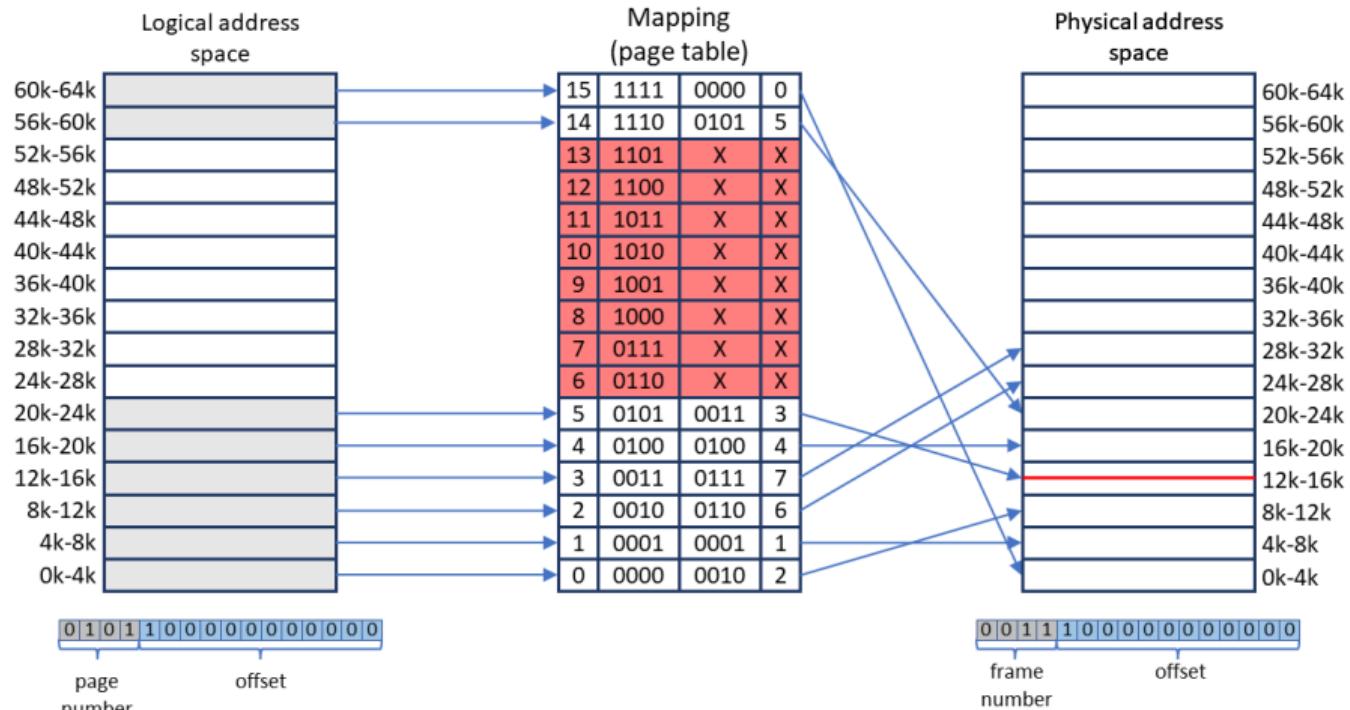
Virtual Memory

An Example



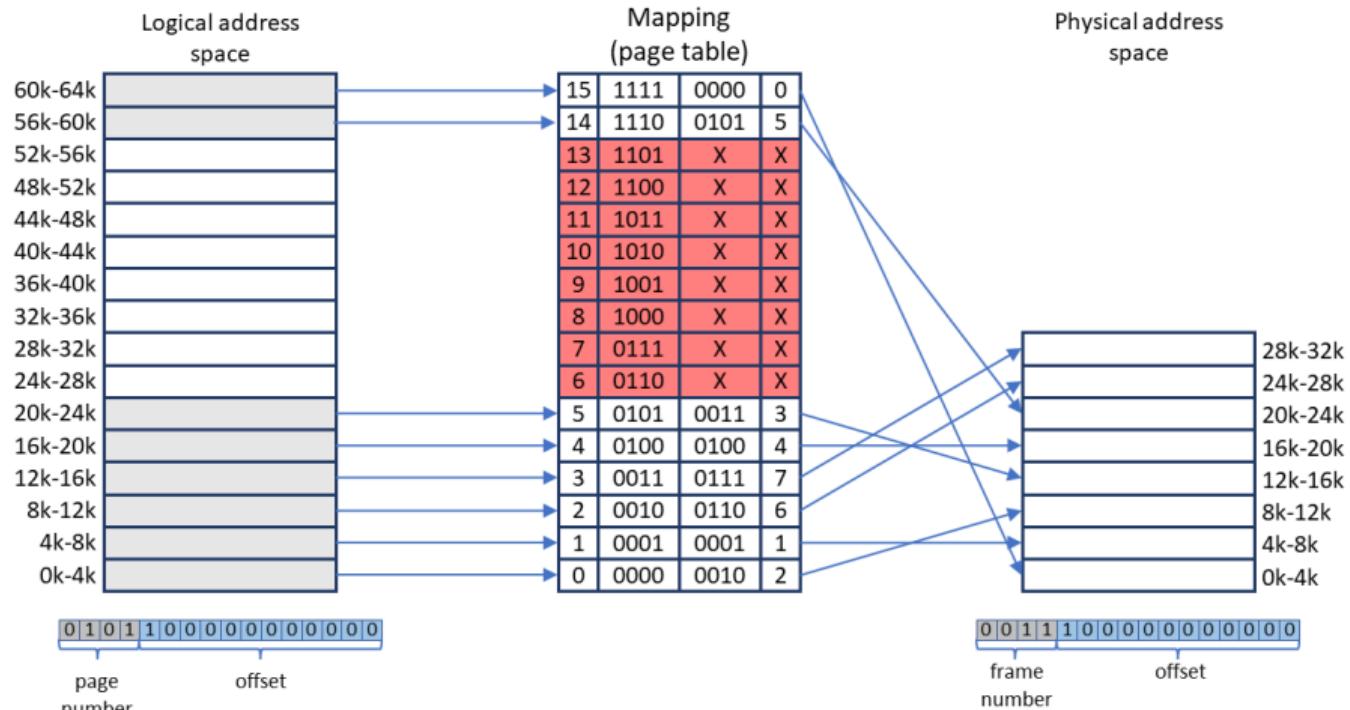
Virtual Memory

An Example



Virtual Memory

An Example



Virtual Memory

Page Faults

- The **resident set** refers to the pages that are loaded in main memory
- A **page fault** is generated if the processor accesses a page that is **not in memory**
 - A page fault results in an **interrupt** (process enters **blocked state**)
 - An **I/O operation** is started to bring the missing page into main memory
 - A **context switch** (may) take place
 - An **interrupt signals** that the I/O operation is complete (process enters **ready state**)

Demand Paging

Processing Page Faults

1. Trap operating system
 - Save registers/process state
 - Analyse interrupt (i.e., identify page fault)
 - Validate page reference, determine frame location
 - Issue disk I/O: queueing, seek, latency, transfer
2. Context switch (optional)
3. Interrupt for I/O completion
 - Store process state/registers
 - Analyse interrupt from disk
 - Update page table (page now in memory)
 - Wait for original process to be scheduled
4. Context switch to original process

Virtual Memory

The Benefits

① Virtual memory **improves CPU utilisation**

- Individual **processes take up less memory** (only partially loaded)
- **More processes** in memory
- More **efficient use of memory** (less internal fragmentation, no external fragmentation)

② The **logical address space** can be **larger than physical address space**

- 64 bit machine $\Rightarrow 2^{64}$ logical addresses (theoretically)

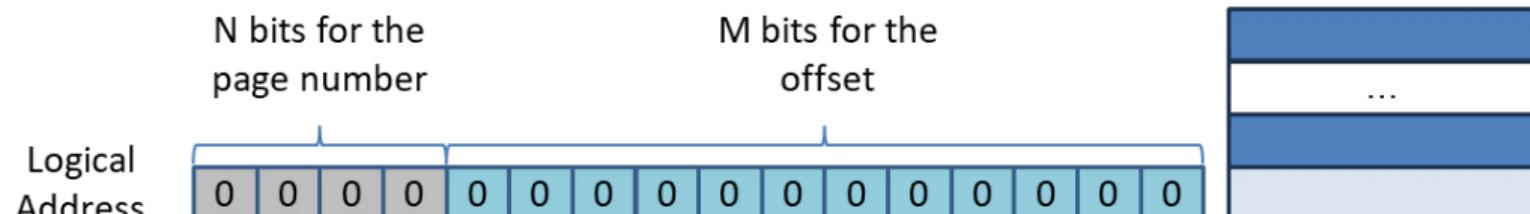


Figure: Logical Address

Virtual Memory

The Benefits

① Virtual memory **improves CPU utilisation**

- Individual **processes take up less memory** (only partially loaded)
- **More processes** in memory
- More **efficient use of memory** (less internal fragmentation, no external fragmentation)

② The **logical address space** can be **larger than physical address space**

- 64 bit machine $\Rightarrow 2^{64}$ logical addresses (theoretically)

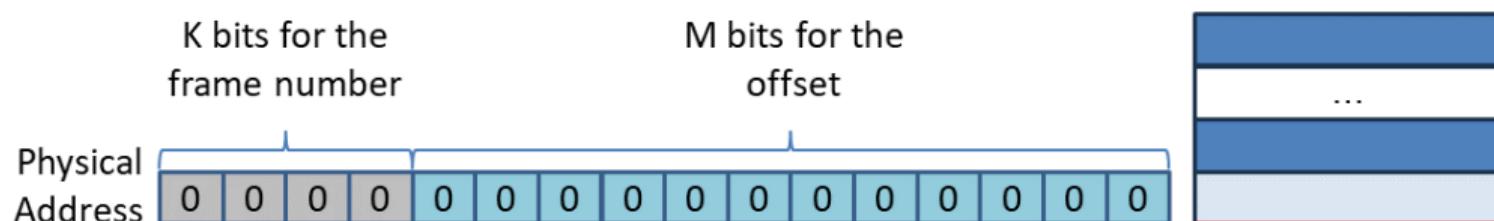


Figure: Logical Address

Virtual Memory

Page Tables Revisited: Contents of a Page Entry

- The “**present/absent bit**” is set if the **page/frame is in memory**
- The “**modified bit**” is set if the **page/frame has been modified**
 - Only modified pages have to be written back to disk when evicted
- The “**referenced bit**” that is set if the page is in use
- **Protection and sharing bits:** read, write, execute or combinations thereof

Other Info	Referenced	Modified	Protection	Present/Absent	Frame Number
...	0/1	0/1	RWX	0/1	

Figure: Page table entry

Virtual Memory

Page Tables Revisited: Page Table Size

- For a **16 bit machine**, the total address space is 2^{16}
 - Assuming that 10 bits are used for the offset (2^{10})
 - 6 bits can be used to number the pages
 - I.e., $2^6 = 64$ pages can be maintained
- For a **32 bit machine**, total address space is 2^{32}
 - Assuming pages of 2^{12} bytes (4KB)
 - 20 bits can be used to number the pages
 - I.e. 2^{20} pages (approx. 1 million) can be maintained
 - 4MB at 4 bytes per page table entry!
- For a **64 bit machine** ...

Virtual Memory

Page Tables Revisited: Dealing with Large Page Tables

① **Size:** how do we deal with **the increasing size of page tables**, i.e., where do we store them?

- Their size prevents them from being **stored in registers**
- They have to be stored in (virtual) **main memory**:
 - **Multi-level page tables**
 - **Inverted page tables** (for large virtual address spaces)

② **Speed:** address translation happens at every memory reference, it has to be fast!

- How can we maintain **acceptable speeds**?
- Accessing main memory results in **memory stalls**

Virtual Memory

Page Tables Revisited: Multi-level Page Tables

- **Solution:**

- Page the page table!
 - **Tree-like** structures to hold page tables
- The **page number** is divided into:
 - An **index** to a **page table** of second level
 - A **page within a second level page table**

- The page table is **not kept entirely in memory**

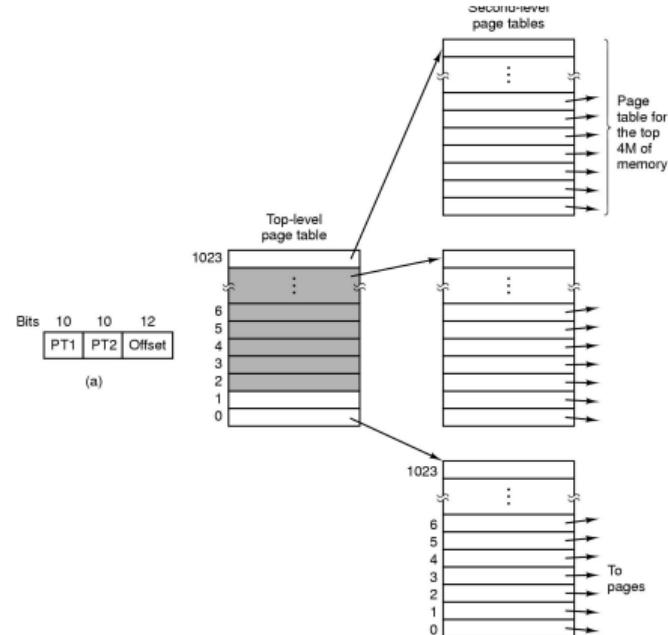


Figure: Multi-level page tables (from Tanenbaum)

Virtual Memory

Page Tables Revisited: Multi-level Page Tables

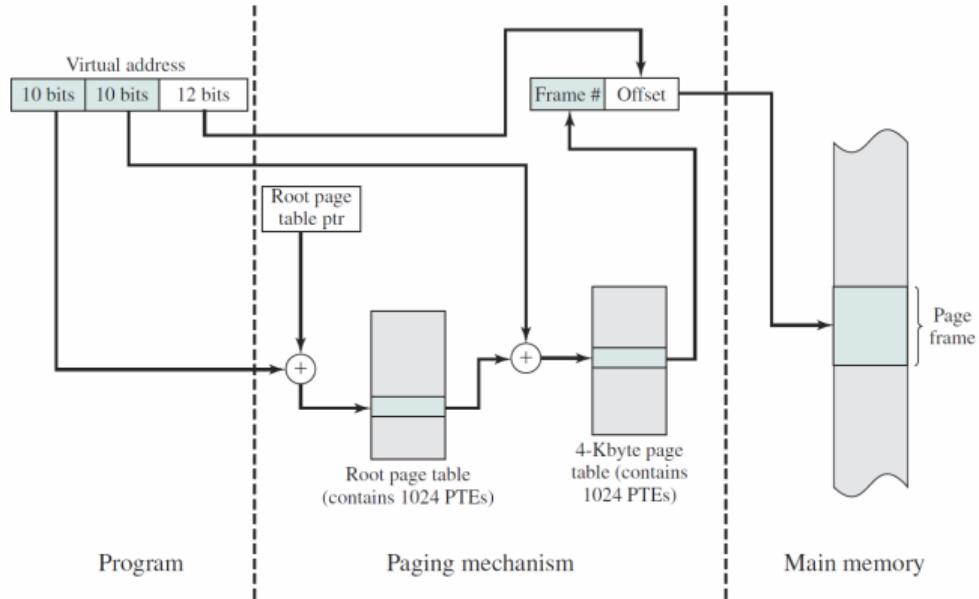


Figure: Multi-level Address Translation (from Stallings)

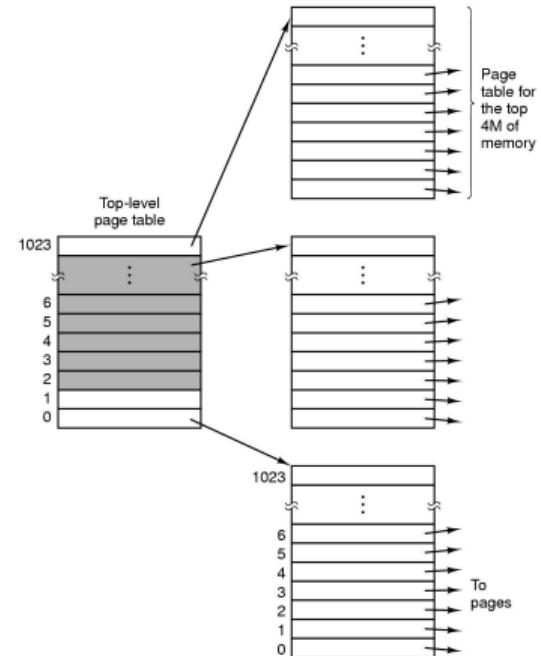


Figure: Multi-level page tables (from Tanenbaum)

Virtual Memory

Page Tables Revisited: Access Speed

- **Memory organisation** of multi-level page tables:
 - The **root page table** is always maintained in memory
 - Page tables themselves are maintained in **virtual memory** due to their size
- Assume that a **fetch** from main memory takes T nano seconds
 - With a **single page table level**, access is $2 \times T$
 - With **two page table levels**, access is $3 \times T$
 - ...

Test Your Understanding

Address Translation: Exercises

- Given a 4KB page/frame size, and a 16-bit address space, calculate:
 - Number **M** of bits for offset within a page
 - Number **N** of bits for representing pages
- What is the physical address for 0, 8192, 20500 using this page table?

Pages		Frames	
0	0000	0010	2
1	0001	0001	1
2	0010	0110	6
3	0011	0000	0
4	0100	0100	4
5	0101	0011	3
6	0110	X	X
7	0111	X	X
8	1000	X	X
9	1001	0101	5
10	1010	X	X
11	1011	0111	7
12	1100	X	X

Table: Page table

Summary

Take-Home Message

- **Paging** splits logical and physical address spaces into small **pages/frames** to reduce internal and external fragmentation
- **Virtual memory** exploits the principle of **locality** and allows for processes to be **loaded only partially** into memory, **large logical address spaces** require “different” approaches
- Reading: Tanenbaum Section 3.3



valid for 65 minutes from 8:55am
generated 2023-10-17 03:13

Figure: Attendance Monitoring

Operating Systems and Concurrency

Memory Management 5
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Goals for Today

Overview

- **Page tables: multi-level page tables, inverted page tables and performance**
- Several **key decisions** have to be made when **using virtual memory**
 - When are pages **fetched** ⇒ demand or pre-paging
 - **What pages** are **removed** from memory ⇒ **page replacement algorithms**
- The **optimal** and **FIFO** page replacement algorithm

Recall

Last Lecture

- **Virtual memory** relies on **localities** that constitute **groups of pages** which are **used together**, e.g., related to a function (code, data, etc.)
 - Processes **move between localities**
 - If all required pages are **in memory**, **no page faults** will be generated
- **Page tables** become **more complex** (present/absent bits, referenced/modified bits) and **larger**

Virtual Memory

Page Tables Revisited: Page Table Size

- For a **16 bit machine**, the total address space is 2^{16}
 - Assuming that 10 bits are used for the offset (2^{10})
 - 6 bits can be used to number the pages
 - I.e., $2^6 = 64$ pages can be maintained
- For a **32 bit machine**, total address space is 2^{32}
 - Assuming pages of 2^{12} bytes (4KB)
 - 20 bits can be used to number the pages
 - I.e. 2^{20} pages (approx. 1 million) can be maintained
 - 4MB at 4 bytes per page table entry!
- For a **64 bit machine** ...

Virtual Memory

Page Tables Revisited: Challenges

- ① **Size:** how do we deal with **the increasing size of page tables** and **where do we store** them?
 - Their size prevents them from being **stored in registers**
 - Increasing the **page size** reduces the **page table size** (e.g., $n+m=32$)
 - They have to be stored in (virtual) **main memory**:
 - **Multi-level page tables**
 - **Inverted page tables** (for large virtual address spaces)
- ② **Speed:** address translation happens at every memory reference, it has to be fast!
 - Accessing main memory results in **memory stalls**
 - How can we maintain **acceptable speeds**?

Virtual Memory

Page Tables Revisited: Multi-level Page Tables

- **Solution:**

- Page the page table!
 - **Tree-like** structures to hold page tables
- The **page number** is divided into:
 - An **index** to a **page table** of second level
 - A **page within a second level page table**

- The page table is **not kept entirely in memory**

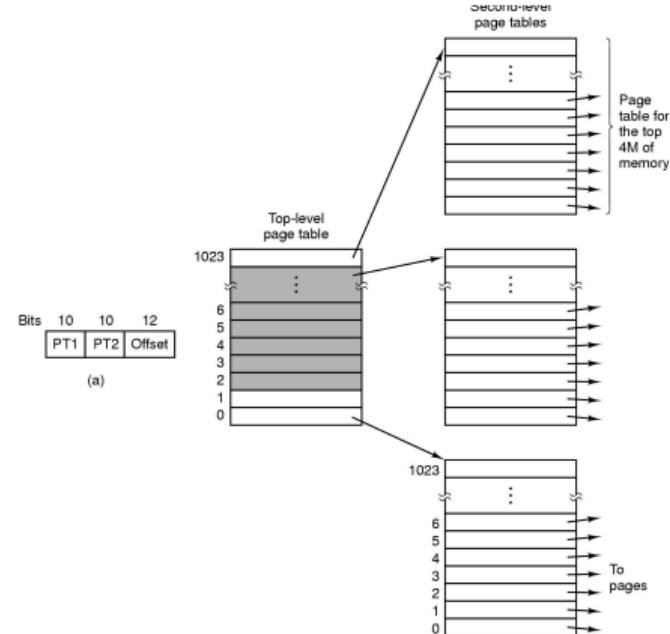


Figure: Multi-level page tables (from Tanenbaum)

Virtual Memory

Page Tables Revisited: Multi-level Page Tables

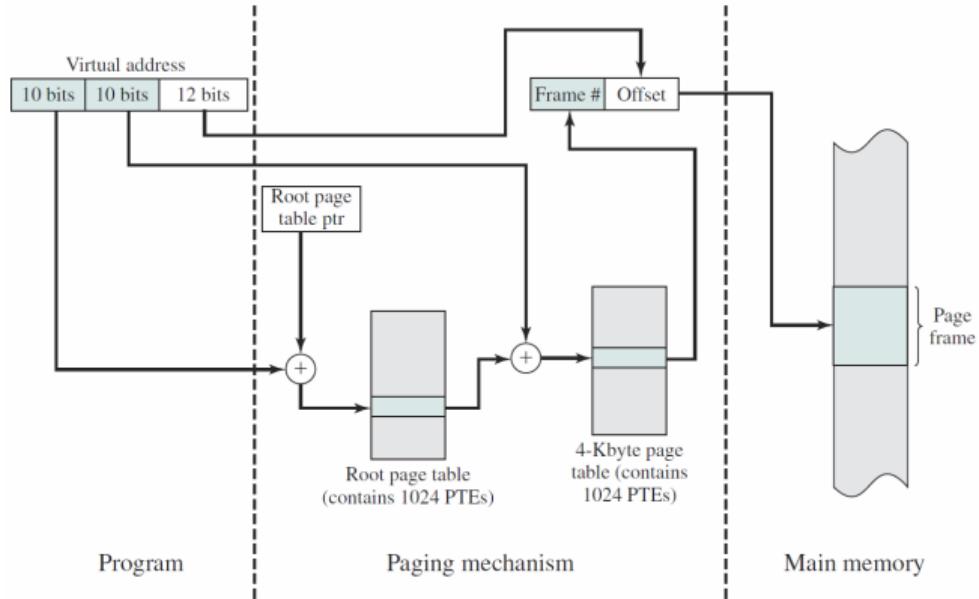


Figure: Multi-level Address Translation (from Stallings)

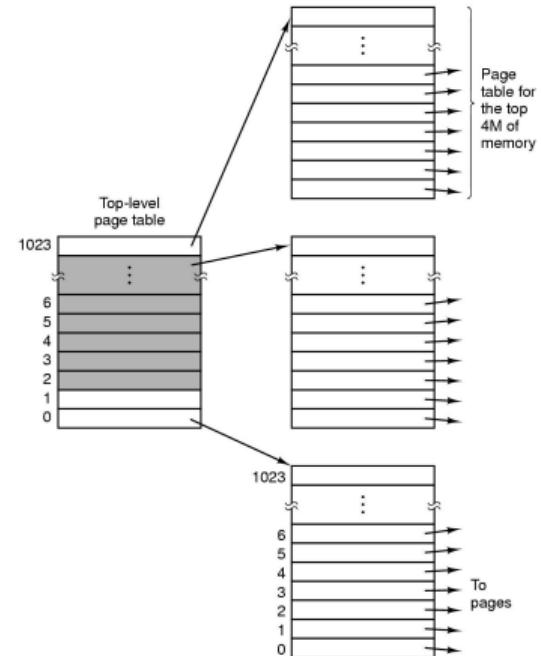


Figure: Multi-level page tables (from Tanenbaum)

Virtual Memory

Access Speed on [multi-level] Page Tables

- **Memory organisation** of multi-level page tables:
 - The **root page table** is always maintained in memory
 - Page tables themselves are maintained in **virtual memory** due to their size
- Assume that a **fetch** from main memory takes T nano seconds
 - With a **single page table level**, access is $2 \times T$
 - With **two page table levels**, access is $3 \times T$
 - ...
- With two levels, every memory reference already becomes **3 times slower**:
 - Assuming that the second level page table is **already in main memory**
 - Memory access already forms a **bottleneck** under normal circumstances

Virtual Memory

Page Tables Revisited: Translation Look Aside Buffers (TLBs)

- **Translation look aside buffers** (TLBs) are (usually) located inside the memory management unit
 - They **cache** the most frequently used page table entries
 - They can be searched **in parallel**
- The principle behind TLBs is similar to other types of **caching in operating systems**
- Remember: **locality** states that processes make a **large number of references** to a **small number of pages**

Virtual Memory

Translation Look Aside Buffers (TLBs)

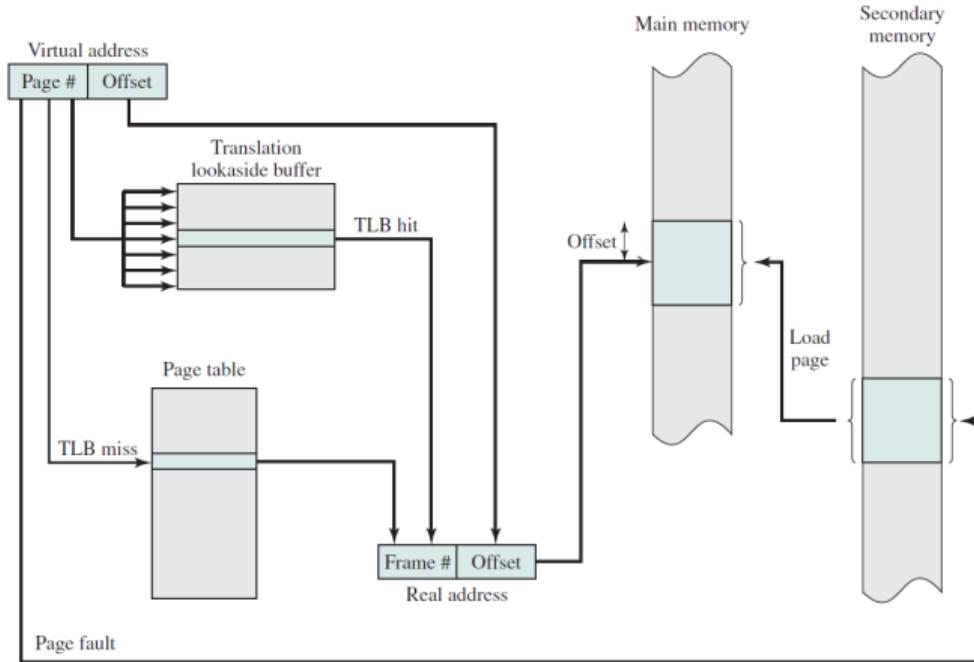


Figure: TLB Address Translation with a single-level page table(from Stallings)

Virtual Memory

Page Tables Revisited: Translation Look Aside Buffers (TLBs)

- Memory access with TLBs:
 - Assume a single-level page table
 - Assume a 20ns associative **TLB lookup**
 - Assume a 100ns **memory access time**
 - **TLB Hit** $\Rightarrow 20 + 100 = 120$ ns
 - **TLB Miss** $\Rightarrow 20 + 100 + 100 = 220$ ns
- Performance evaluation of TLBs:
 - For an 80% hit rate, the estimated access time is:
 $120 \times 0.8 + 220 \times (1 - 0.8) = 140$ ns (i.e. **40% slowdown** – relative to absolute addressing)
 - For a 98% hit rate, the estimated access time is:
 $120 \times 0.98 + 220 \times (1 - 0.98) = 122$ ns (i.e. 22% slowdown)
- Note that **page tables** can be **held in virtual memory** \Rightarrow further (initial) slow down due to page faults

Virtual Memory

Page Tables Revisited: Inverted Page Tables

- A “**normal**” **page table’s size** is proportional to the number of pages in the virtual address space (prohibitive for modern machines)
- An “**inverted**” **page table’s size** is proportional to the size of main memory
 - Contains one **entry for every frame** (i.e. not for every page)
 - Is **indexed by frame number/hash code** (not by page number)
 - When a process references a page, the OS must search the (entire) inverted page table for the corresponding entry (i.e. page and process id) \Rightarrow this could be too slow.
- *Solution:* Use a **hash function** that transforms page number (n bits) into an index in the inverted page table (hash table)

Virtual Memory

Page Tables Revisited: Inverted Page Table Entries

- Process Identifier (**PID**): the process that owns this page.
- Page Number (logical address space)
- **Protection** bits (Read/Write/Execution)
- **Link Pointer**: field points toward the **next entry** (hash table can have collisions)

PID	Page Number	RWX	Link Pointer	Frame Number
-----	-------------	-----	--------------	--------------

Figure: Example of an Inverted Page Table Entry (other info bits are not shown here)

Virtual Memory

Page Tables Revisited: Inverted Page Tables - Address translation

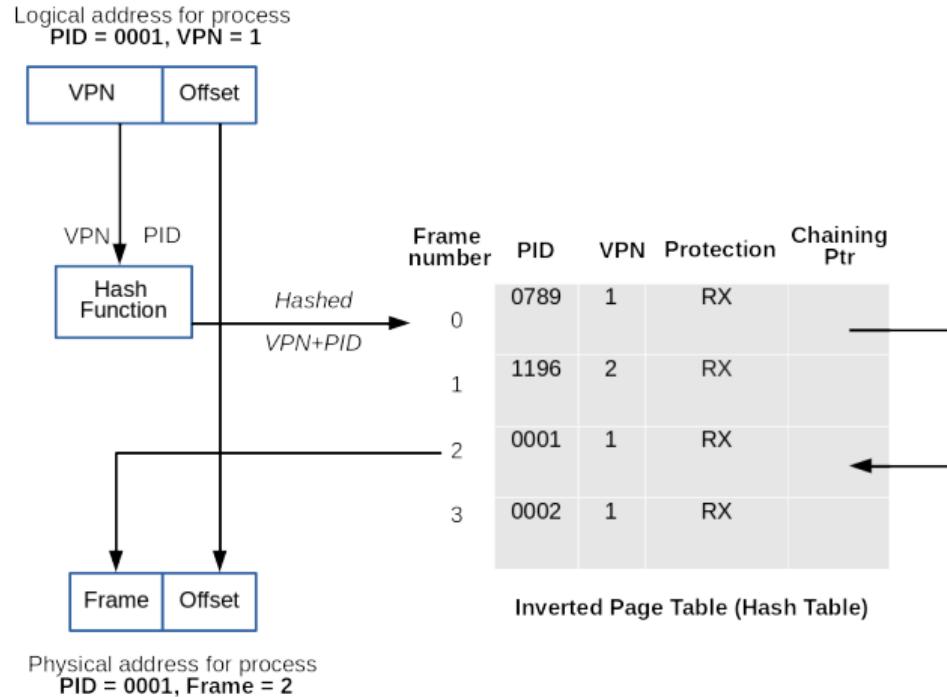


Figure: Address Translation with an Inverted Page Table

Virtual Memory

Page Tables Revisited: Inverted Page Tables

- Advantages:
 - The OS maintains a **single inverted page table** for all processes
 - **Saves space** (especially when the virtual address space is much larger than the physical memory)
- Disadvantages:
 - Logical-to-physical address translation becomes **harder/slower**.
 - **Collisions** have to be handled and will slow down address translation:
 - Hash tables eliminates searching the whole inverted table)
 - TLBs help to improve performance
- Commonly used on 64-bit machines (e.g. Windows 10)

[http://answers.microsoft.com/en-us/windows/forum/windows_10-performance/
physical-and-virtual-memory-in-windows-10/e36fb5bc-9ac8-49af-951c-e7d39b979938](http://answers.microsoft.com/en-us/windows/forum/windows_10-performance/physical-and-virtual-memory-in-windows-10/e36fb5bc-9ac8-49af-951c-e7d39b979938)

Virtual Memory

Page loading/replacement

- **Two key decisions** have to be made using virtual memory
 - What pages are **loaded** and when ⇒ predictions can be made
 - What pages are **removed** from memory and when ⇒ **page replacement algorithms**
- **Pages are shuttled** between primary and secondary memory

Demand Paging

On Demand

- **Demand paging** starts the process with **no pages in memory**
 - The first instruction will immediately cause **a page fault**
 - **More page faults** will follow, but they will **stabilise over time** until moving to the **next locality**
 - The set of pages that is currently being **used** is called its **working set** (\Leftrightarrow resident set)
- Pages are only **loaded when needed**, i.e. following **page faults**

Pre-Paging

Predictive

- When the process is started, all pages expected to be used (i.e. the working set) could be **brought into memory at once**
 - This can drastically **reduce the page fault rate**
 - Retrieving multiple (**contiguously stored**) pages **reduces transfer times** (seek time, rotational latency, etc.)
- Pre-paging** loads pages (as much as possible) **before page faults are generated** (\Rightarrow a similar mechanism is used when processes are **swapped out/in**)

Virtual Memory

Implementation Details

- Avoiding **unnecessary pages** and **page replacement** is important!
- Let:
 - ma denote the **memory access time** (multiple times for multi-level page table)
 - p denote the **page fault rate**
 - pft denote the **page fault time**
- The **effective access time** is then given by:

$$T_a = (1 - p) \times ma + pft \times p \quad (1)$$

- Note that we are not considering here TLBs.

Demand Paging

Performance Evaluation of Demand Paged Systems

- For a single-level page table:
 - Memory **access time** of 100ns (10^{-9})
 - Two memory accesses required (200ns)
 - A **page fault time** of 8ms (10^{-3} , recall that hard drives are slow)

$$T_a = (1 - p) \times 200 + p \times 8000000 \quad (2)$$

- The effective access time is **proportional to page fault rate**
 - Ideally, all pages would have to be loaded without demand paging

Page Replacement

Concepts

- The OS must choose a **page to remove** when a new **one is loaded** (and all are occupied)
- This choice is made by **page replacement algorithms** and **takes into account**
 - When the page is **last used/expected to be used again**
 - Whether the **page has been modified** (only modified pages need to be written)
- Replacement choices have to be **made intelligently** (\Leftrightarrow random) to **save time/avoid thrashing**

Page Replacement

Algorithms

- ① **Optimal** page replacement
- ② **FIFO** page replacement
 - Second chance replacement
 - Clock replacement
- ③ **Not recently used** (NRU)
- ④ **Least recently used** (LRU)

Page Replacement

Optimal Page Replacement

- In an **ideal/optimal** world
 - Each page is labeled with the **number of instructions** that will be executed/length of time before it is **used again**
 - The page which **is not going to be referenced** for the **longest time** is the optimal one to remove
- The **optimal approach** is **not possible to implement**
 - It can be used for **post-execution analysis** ⇒ what would have been the minimum number of page faults
 - It provides a **lowerbound** on the **number of page faults** (used for comparison with other algorithms)

Test Your Understanding

Address Translation

- Given a 4KB page/frame size, and a 16-bit address space, calculate:
 - Number **M** of bits for offset within a page
 - Number **N** of bits for representing pages
- What is the physical address for 0, 8192, 20500 using this page table?

Pages		Frames	
0	0000	0010	2
1	0001	0001	1
2	0010	0110	6
3	0011	0000	0
4	0100	0100	4
5	0101	0011	3
6	0110	X	X
7	0111	X	X
8	1000	X	X
9	1001	0101	5
10	1010	X	X
11	1011	0111	7
12	1100	X	X

Table: Page table

Recap

Take-Home Message

- Translation look aside buffers to speed up access to page tables
- Inverted page tables
- Fetching policies (demand paging, pre-paging)
- Page replacement strategies
- Reading: Tanenbaum Section 3.3, 3.4

COMP2007: Operating Systems & Concurrency
Week 9 – 3:00pm Monday – 20 November 2023



valid for 65 minutes from 2:55pm
generated 2023-10-17 03:13

Figure: Attendance Monitoring

Operating Systems and Concurrency

Memory Management 6
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Goals for Today

Overview

- Several **key decisions** have to be made when **using virtual memory**
 - When are pages **loaded** into memory
 - **What pages** are **removed** from memory ⇒ page replacement algorithms
 - **How many pages** are allocated to a process and are they **local or global**
 - **When** are pages **removed** from memory ⇒ paging daemons
- **What problems** may occur in virtual memory ⇒ thrashing

Page Replacement

First-In, First-Out (FIFO)

- FIFO maintains a **linked list** and **new pages** are added at the end of the list
- The **oldest page** at the **head of the list** is evicted when a page fault occurs
- The **(dis-)advantages** of FIFO include:
 - It is **easy** to understand/implement
 - It **performs poorly** ⇒ heavily used pages are just as likely to be evicted as a lightly used pages

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1																								
PF2																								
PF3																								
PF4																								

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4																
PF1	0															
PF2	-															
PF3	-															
PF4	-															

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4																	
PF1	0	0															
PF2	-	2															
PF3	-	-															
PF4	-	-															

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4															
PF1	0	0	0												
PF2	-	2	2												
PF3	-	-	1												
PF4	-	-	-												

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF2	-	0	0	0	0																			
PF3	-	-	1	1																				
PF4	-	-	-	3																				

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4															
PF1	0	0	0	0	0	5									
PF2	-	2	2	2	2	2									
PF3	-	-	1	1	1										
PF4	-	-	-	3	3										

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4															
PF1	0	0	0	0	0	5	5								
PF2	-	2	2	2	2	2	4								
PF3	-	-	1	1	1	1	1								
PF4	-	-	-	3	3	3									

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4																	
PF1	0	0	0	0	5	5	5										
PF2	-	2	2	2	2	4	4										
PF3	-	-	1	1	1	1	6										
PF4	-	-	-	3	3	3	3										

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5
PF2	-	2	2	2	2	4	4	4	4						
PF3	-	-	1	1	1	1	6	6							
PF4	-	-	-	3	3	3	3	3							

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4															
PF1	0	0	0	0	5	5	5	5	5						
PF2	-	2	2	2	2	4	4	4	4						
PF3	-	-	1	1	1	1	6	6	6						
PF4	-	-	-	3	3	3	3	3	7						

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5									
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4										
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6											
PF4	-	-	-	3	3	3	3	3	7	7	7													

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	5	5	5	5	5	5	3									
PF2	-	2	2	2	2	2	4	4	4	4	4	4	4	4	4									
PF3	-	-	1	1	1	1	1	6	6	6	6	6	6	6	6									
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7										

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	5	5	5	5	3	3										
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4									
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	6									
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7											

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

PF1	0	0	0	0	5	5	5	5	5	5	5	3	3	3	5	5	3
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4	5	
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	6	6	
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7	7	7	

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

PF1	0	0	0	0	5	5	5	5	5	5	5	3	3	3	5	5	3
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	5	5	5
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	6	6	6
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7	7	7	7

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

PF1	0	0	0	0	5	5	5	5	5	5	5	3	3	3	5	5	3
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	5	5	5
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	6	6	1
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7	7	7	7

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

PF1	0	0	0	0	5	5	5	5	5	5	3	3	3	3	3	3	3
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	5	5	5	5
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	1	1	1
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7	7	7	7

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	5	5	5	3	3	3	3	3	3	3	3	3	3	3	3	
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	6	6	1	1	1	1	1	1	1	
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7	7	7	7	7	7	7	7	2		

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	5	5	5	3	3	3	3	3	3	3	3	3	3	3	3	
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	6	1	1	1	1	1	1	1	1	
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7	7	7	7	7	7	7	2	2	2	

Figure: FIFO Page Replacement

Page Replacement

FIFO Simulation

- Assume we have a system with **eight logical pages** and **four physical frames** (PFs)
- Consider the following page **references in order**:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **13**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	5	5	5	3	3	3	3	3	3	3	3	3	3	3	3	4
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5
PF3	-	-	1	1	1	1	1	6	6	6	6	6	6	6	6	6	1	1	1	1	1	1	1	1
PF4	-	-	-	3	3	3	3	3	7	7	7	7	7	7	7	7	7	7	7	7	2	2	2	2

Figure: FIFO Page Replacement

Page Replacement

Second Chance FIFO

- Second chance FIFO:
 - If a page at the front of the list has **not been referenced** it is **evicted**
 - If the reference bit is set, the page is **placed at the end** of list and its **reference bit reset**
- Second chance FIFO **works better** than FIFO, but is **costly to implement** (list changes constantly) and **can degrade to FIFO** if all pages were initially referenced

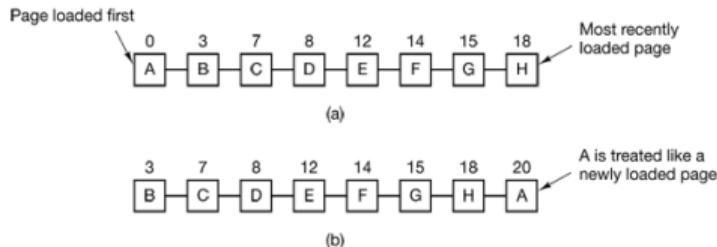


Figure: Second-Chance Algorithm (Tanenbaum)

Page Replacement

The Clock Replacement Algorithm

- **Second-chance FIFO** can be improved by maintaining a circular list
 - A **pointer** points to the last page “visited”
 - The algorithm is called (one-handed) **clock**
 - It is **slow for long lists**
- The **time spent** on maintaining the list is **reduced**

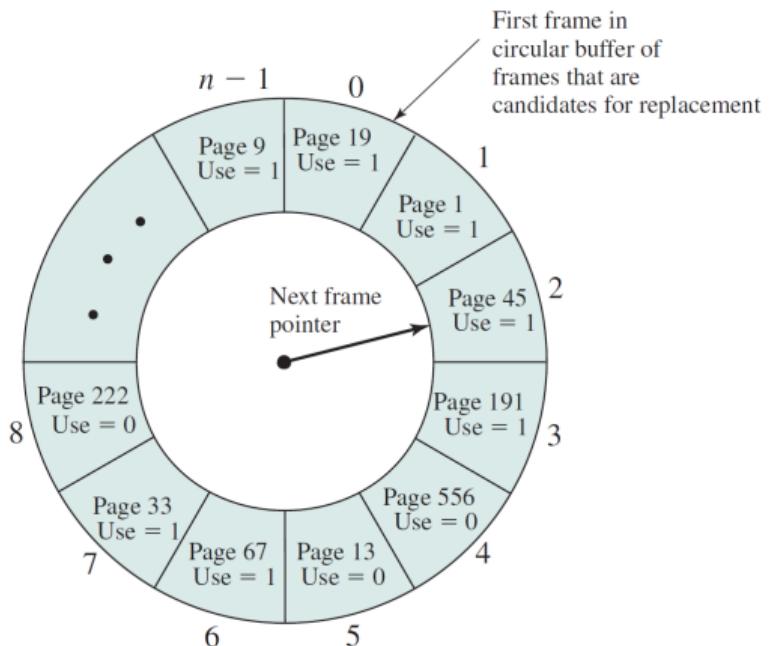


Figure: Clock Replacement Algorithm (Stallings)

Page Replacement

Not Recently Used (NRU)

- **Referenced** and **modified** bits are kept in the page table
 - Referenced bits are set to 0 at the start, and **reset periodically** (e.g. system clock interrupt or when searching the list)
- Four different **page “types”** exist
 - class 0: not referenced and not modified
 - class 1: not referenced and modified
 - class 2: referenced and not modified
 - class 3: referenced and modified

Page Replacement

Not Recently Used (NRU, Cont'd)

- **Page table entries** are inspected upon every **page fault**
- Can be implemented as:
 - ① Find a page from **class 0** to be removed
 - ② If step 1 fails, scan again looking for **class 1** and set the reference bit to 0 for each page visited
 - ③ If step 2 fails, start again from step 1 (elements from class 2 and 3 have now become class 0 or 1).
- The NRU algorithm provides **good performance** and is **easy to understand and implement**

Page Replacement

Least-Recently-Used

- Least recently used **evicts the page** that has **not been used the longest**
 - The OS **keeps track** of when a page was **last used**
 - Every **page table entry** contains a **field for the counter**
 - **Costly implementation** since it requires a **list of pages sorted** in the order in which they have been used
- The algorithm can be **implemented in hardware** using a **counter** that is incremented after each instruction

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1																								
PF2																								
PF3																								
PF4																								

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4																
PF1	0															
PF2	-															
PF3	-															
PF4	-															

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1
PF2	-	2															
PF3	-	-															
PF4	-	-															

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF2	-	0	0	0																				
PF3	-	-	2	2																				
PF4	-	-	-	1																				

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF2	-	0	0	0	0																			
PF3	-	-	1	1																				
PF4	-	-	-	3																				

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF2	-	0	2	2	2	2																		
PF3	-	-	1	1	1																			
PF4	-	-	-	3	3																			

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5
PF2	-	2	2	2	2	2	4								
PF3	-	-	1	1	1	1									
PF4	-	-	-	3	3	3									

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF2	-	0	0	0	0	5	5	5																
PF3	-	-	1	1	1	1	1	6																
PF4	-	-	-	3	3	3	3																	

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

PF1	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5
PF2	-	2	2	2	2	4	4	4	4						
PF3	-	-	1	1	1	1	6	6							
PF4	-	-	-	3	3	3	3	3							

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7															
PF2	-	2	2	2	2	4	4	4	4															
PF3	-	-	1	1	1	1	6	6	6															
PF4	-	-	-	3	3	3	3	3	3															

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7									
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4									
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	6									
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3									

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
- Consider the following **page references** in order:
0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7									
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4									
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	6	5									
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3									

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
 - Consider the following **page references** in order:
- 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	5	5	5								
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
 - Consider the following **page references** in order:
- 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7					
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4	4	4	4	4	1				
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	5	5	5	5	5						
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3					

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
 - Consider the following **page references** in order:
- 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4	4	4	4	1	1	1	1	1	
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	5	5	5	5	5	5	5	5	5	5	
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
 - Consider the following **page references** in order:
- 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4	4	4	1	1	1	1	1	1	
PF3	-	-	1	1	1	1	6	6	6	6	6	6	6	5	5	5	5	5	5	5	2			
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
 - Consider the following **page references** in order:
- 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4	4	1	1	1	1	1	1	1	
PF3	-	-	1	1	1	1	6	6	6	6	6	6	5	5	5	5	5	5	5	2	2	2		
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	

Figure: Least Recently Used

Page Replacement

Least-Recently-Used

- Assume we have a system with eight logical address pages & four physical page frames
 - Consider the following **page references** in order:
- 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 1 4
- The number of **page faults** that are generated is **12**

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1	0	0	0	0	5	5	5	5	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	4
PF2	-	2	2	2	2	4	4	4	4	4	4	4	4	4	4	4	1	1	1	1	1	1	1	1
PF3	-	-	1	1	1	1	6	6	6	6	6	6	5	5	5	5	5	5	5	2	2	2	2	2
PF4	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Figure: Least Recently Used

Page Replacement Algorithms

Summary

- ① **Optimal** page replacement: optimal but not practical
- ② **FIFO** page replacement: poor performance but easy to implement
 - Second chance replacement: improved performance but poor implementation
 - Clock replacement: improved implementation but can still be slow
- ③ **Not recently used** (NRU): easy to understand, moderately efficient (approximation of LRU)
- ④ **Least recently used** (LRU): close to optimal but more difficult to implement

Resident Set

Size of the Resident Set

- How many pages should be allocated to individual processes:
 - **Small resident sets** enable to store **more processes** in memory ⇒ improved CPU utilisation
 - **Small resident sets** may result in **more page faults**
 - **Large resident sets** may **no longer reduce the page fault rate (diminishing returns)**
- A trade-off exists between the **sizes of the resident sets** and **system utilisation**

Resident Set

Size of the Resident Set

- Resident set sizes may be **fixed** or **variable** (i.e. adjusted at runtime)
- For **variable sized** resident sets, **replacement policies** can be:
 - **Local**: a page of the same process is replaced
 - **Global**: a page can be taken away from a different process
- Variable sized sets require **careful evaluation of their size** when a **local scope** is used (often based on the **working set** or the **page fault frequency**)

Resident Set

Size of the Resident Set: Local vs. Global approaches

A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0	
A1	
A2	
A3	
A4	
A6	
B0	
B1	
B2	
B3	
B4	
B5	
B6	
C1	
C2	
C3	

(b)

A0	
A1	
A2	
A3	
A4	
A5	
B0	
B1	
B2	
A6	
B4	
B5	
B6	
C1	
C2	
C3	

(c)

Figure: Local vs. global page replacement. (a) Original config, number at the right represents loading time (b) Local (c) Global (Tanenbaum)

Working Sets

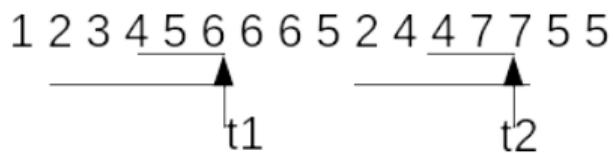
Defining and Monitoring Working Sets

- The **resident set** comprises the set of pages of the process that **are in memory**
- The **working set** $W(t, k)$ comprises the set referenced pages in the last k (= working set window) virtual time units for the process
- k can be defined as "**memory references**" or as "**actual process time**"
 - The the set of most recently used pages
 - The set of pages used within a pre-specified time interval
- The **working set size** can be used as a guide for the number frames that should be allocated to a process

Working Sets

Monitoring Working Sets: Example

- Consider the following page **references in order**:



- If $k = 3$:
 - At t_1 , $W(t_1, 3) = \{4, 5, 6\}$
 - At t_2 , $W(t_2, 3) = \{4, 7\}$
- If $k = 5$:
 - At t_1 , $W(t_1, 5) = \{2, 3, 4, 5, 6\}$
 - At t_2 , $W(t_2, 5) = \{2, 4, 7\}$

Working Sets

Defining and Monitoring Working Sets

- The working set is a **function of time t** :
 - Processes **move between localities**, hence, the pages that are included in the working set **change over time**
 - **Stable** intervals alternate with intervals of **rapid change**
- $|W(t, k)|$ is then variable in time. Specifically:

$$1 \leq |W(t, k)| \leq \min(k, N) \tag{1}$$

where N is the total number of pages of the process.

Working Sets

Monitoring Working Sets

- Choosing the right value for k is paramount:
 - Too **small**: inaccurate, pages are missing
 - Too **large**: too many unused pages present
 - **Infinity**: all pages of the process are in the working set
- Working sets can be used to guide the **size of the resident sets**
 - Monitor the working set
 - Remove pages from the resident set that are not in the working set
- The working set is costly to maintain \Rightarrow **page fault frequency (PFF)** can be used as an approximation
 - If the PFF is increased -> we need to increase k
 - If PFF is very reduced -> we may try to decrease k

Resident Sets

Local vs. Global Replacement

- **Global replacement policies** can select frames from the entire set, i.e., they can be “taken” from other processes
 - Frames are **allocated dynamically** to processes
 - Processes cannot control their own page fault frequency, i.e., the **PFF** of one process is **influenced by other processes**
- **Local replacement policies** can only select frames that are allocated to the current process
 - Every process has a **fixed fraction of memory**
 - The locally “**oldest page**” is not necessarily the globally “oldest page”
- Windows uses a **variable approach** with **local replacement**
- Page replacements algorithms explained before can use both policies.

Paging Daemon

Pre-cleaning (\Leftrightarrow demand-cleaning)

- It is more efficient to **proactively** keep a number of **free pages** for **future page faults**
 - If not, we may have to **find a page** to evict and we **write it to the drive** (if modified) first when a page fault occurs
- Many systems have a background process called a **paging daemon**
 - This process **runs at periodic intervals**
 - It inspects the state of the frames and, if too few frames are free, it **selects pages to evict** (using page replacement algorithms)

Paging Daemon

Pre-cleaning (↔ demand-cleaning)

- It is more efficient to **proactively** keep a number of **free pages** for **future page faults**
 - If not, we may have to **find a page** to evict and we **write it to the drive** (if modified) first when a page fault occurs
- Many systems have a background process called a **paging daemon**
 - This process **runs at periodic intervals**
 - It inspects the state of the frames and, if too few frames are free, it **selects pages to evict** (using page replacement algorithms)
- Paging daemons can be combined with **buffering** (free and modified lists) ⇒ write the modified pages **but keep them in main memory** when possible

Thrashing

Defining Thrashing

- Assume **all available pages are in active use** and a new page needs to be loaded:
 - The page that will be **evicted** will have to be **reloaded soon afterwards**, i.e., it is still active
- **Thrashing** occurs when pieces are swapped out and loaded again immediately

Thrashing

A Vicious Circle?

- CPU utilisation is too low ⇒ scheduler **increases degree of multi-programming**
 - ⇒ Frames are allocated to new processes and **taken away from existing processes**
 - ⇒ I/O **requests are queued** up as a consequence of page faults
- CPU **utilisation drops further** ⇒ scheduler increases degree of multi-programming

Thrashing

Causes/Solutions

- **Causes** of thrashing include:
 - The degree of multi-programming is too high, i.e., the total **demand** (i.e., the sum of all **working set** sizes) **exceeds supply** (i.e. the available frames)
 - An individual process is allocated **too few pages**
- This can be **prevented** by, e.g., using good **page replacement policies**, reducing the **degree of multi-programming** (medium term scheduler), or adding more memory
- The **page fault frequency** can be used to detect that a system is thrashing

Test Your Understanding

FIFO vs. optimal page replacement

- Compare FIFO/LRU with **the optimal page replacement** algorithm. The process starts up with none of its pages in memory.
- What would be the minimum number of **page faults** that would be generated by the optimal approach?

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	1	4
PF1																								
PF2																								
PF3																								
PF4																								

Figure: Optimal Page Replacement

Summary

Take-Home Message

- Second Chance FIFO, Clock Replacement, NRU, LRU page replacement
- Page allocations to processes (variable, fixed, local, global)
- Page Daemons
- Thrashing
- Reading: Tanenbaum Section 3.4, 3.5.1, 3.5.8

Operating Systems and Concurrency

File Systems 1
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Goals for Today

Overview

- Construction **rotational** and **solid state** drives
- **Access times** for hard drives
- **Disk scheduling**

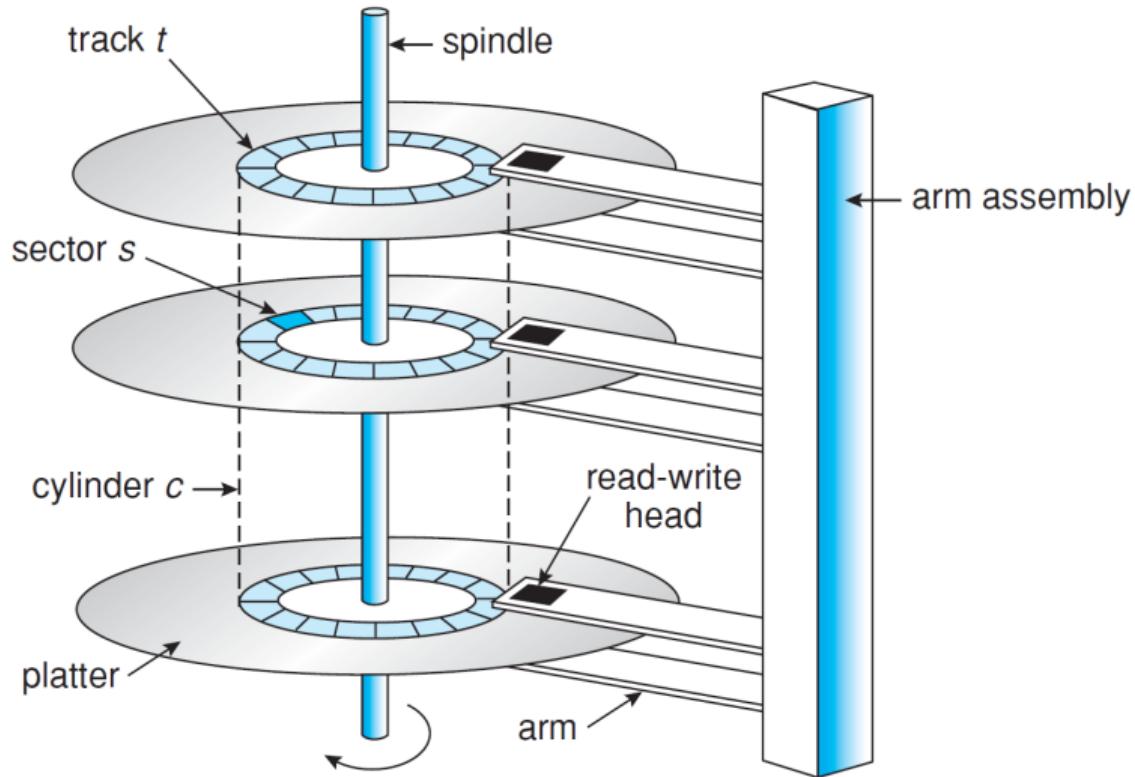
Rotational Hard Drives

Construction

- Rotational hard drives are made of aluminium/glass **platters** covered with **magnetisable material**
 - **Read/write heads** fly just above the surface (0.2 – 0.07mm) and are connected to a single **disk arm** controlled by a single **actuator**
 - **Data** is stored on both sides
 - Common **diameters** range from 1.8 to 3.5 inches
 - They **rotate** at a **constant speed** (speed near the spindle is less than on the outside)
- A **disk controller** abstracts the low level interface
- **Rotational hard drives** are approx. **4 orders of magnitude slower than main memory**

Rotational Hard Drives

Construction



Rotational Hard Drives

Low Level Format

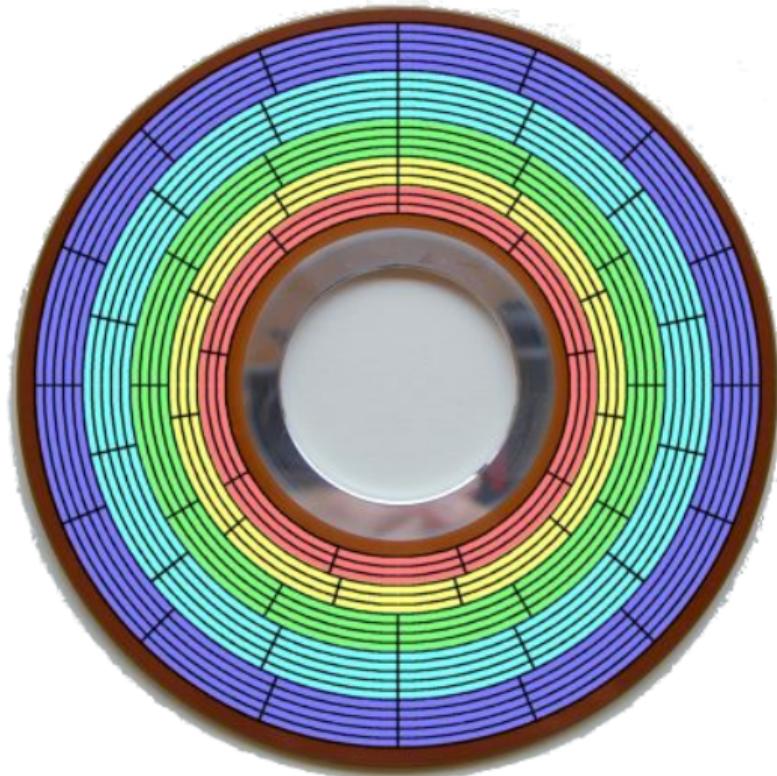
- Disks are organised in:
 - **Cylinders**: all tracks in the same position relative to the spindle
 - **Tracks**: a concentric circle on a single platter side
 - **Sectors**: segments of a track (usually 512B or 4KB in size)
- **Sectors** usually have an **equal number of bytes** in them (**preamble, data, error correcting code - ECC**)
- The **number of sectors increases** from the inner side of the disk to the outside

Preamble	Data	ECC
----------	------	-----

Figure: Disk Sector

Rotational Hard Drives

Organisation of Rotational Drives



Rotational Hard Drives

Organisation

- **Cylinder skew** is an **offset** that is added to sectors in adjacent tracks to account for the seek time
- In the past, consecutive **disk sectors were interleaved** to account for transfer time
- **Disk capacity** is reduced due to **low level formatting** (preamble, ECC, etc.)

Rotational Hard Drives

Access Times

- **Access time** = seek time + rotational delay + transfer time
 - **Seek time**: time needed to move the arm to the cylinder (dominant)
 - **Rotational latency**: time before the sector appears under the head
 - **Transfer time**: time to transfer the data

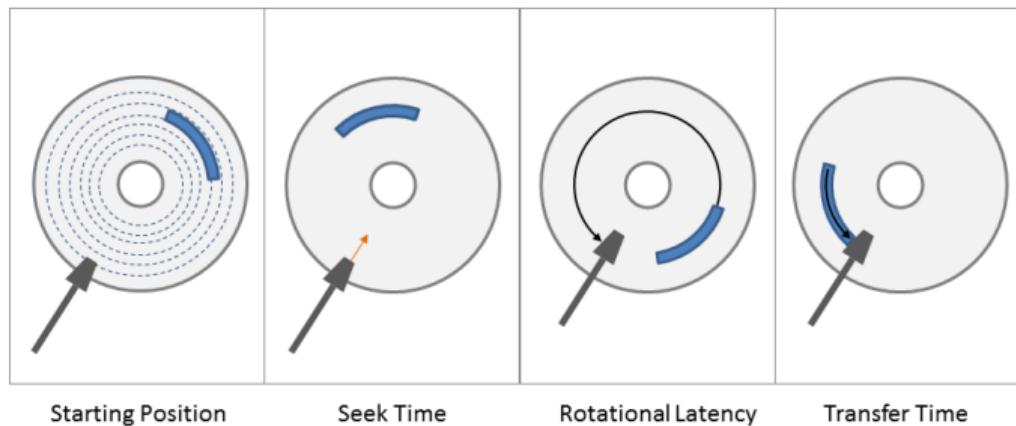
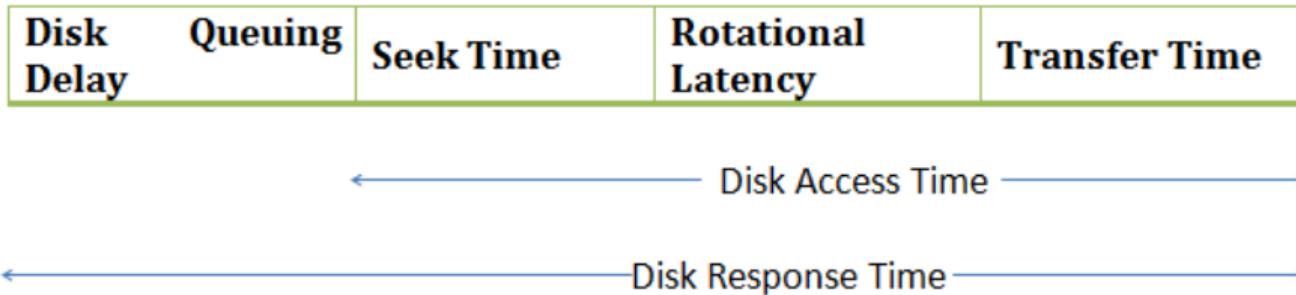


Figure: Access time to Disk (Source: www.studiodialy.com/)

Rotational Hard Drives

Access Times

- Multiple requests may be happening at the same time (concurrently). Thus, access time may be increased by a **queueing time**
- Dominance of seek time leaves room for **optimisation** by carefully considering the order of read operations



Rotational Hard Drives

Access Times

- The **estimated seek time** T_s to move the arm from one track to another is approximated by:

$$T_s = n \times m + s \quad (1)$$

- In which:
 - n the **number of tracks** to be crossed
 - m the **crossing time per track**
 - s any **additional startup delay**

Rotational Hard Drives

Access Times

- Assume a disk that **rotates at 3600rpm** (common rotation speeds are between 3600 and 15000rpm)
 - One rotation takes approx. 16.7ms ($\frac{60000}{3600}$)
 - The average **rotational latency** (T_r) is half a rotation on average ($\frac{16.7}{2} \approx 8.3ms$)
- Let b denote the **number of bytes transferred**, N the **number of bytes per track**, and rpm the **rotation speed** in revolutions per minute
- The **transfer time** T_t is given by:
 - N bytes take 1 revolution (16.7ms)
 - b contiguous bytes takes $\frac{b}{N}$ revolutions

$$T_t = \frac{b}{N} \times ms \text{ per revolution} \quad (2)$$

Rotational Hard Drives

Access Times: Example

- To read a file of **size 256 sectors** with:
 - $T_s = 20$ ms (average seek time)
 - 32 sectors/track
- If the file is stored **contiguously**:
 - The first track: $20 + 8.3 + 16.7 = 45ms$ (seek + rotational delay + transfer time)
 - The remaining tracks (assuming no cylinder skew and negligible seeks time between neighbouring tracks): $8.3 + 16.7 = 25ms$ (rotational delay + transfer time)
- The total time is then $45 + 7 \times 25 = 220ms = 0.22s$

Rotational Hard Drives

Access Times: Example

- In case the access is not sequential but at **random for the sectors**, we get:
 - Time per sector = $T_s + T_r + T_t = 20 + \frac{16.7}{2} + \frac{16.7}{32} = 28.8ms$
 - Total time for 256 sectors = $256 \times 28.8ms = 7.37s$
- Observation: **sectors must be positioned carefully** and avoid **disk fragmentation**

Disk Scheduling

Concepts

- The OS/hardware must:
 - **Position/organise** files and sectors strategically
 - Optimise **disk requests** to **minimise overhead** from seek time and rotational delays
- **I/O requests happen over time** and go through a **system calls** and are **queued**:
 - They are kept in a **table of requested sectors per cylinder**
 - This allows the operating system to **intercept** and **re-sequence** them

Disk Scheduling

Concepts

- **Disk scheduling algorithms** determine the order in which disk requests are processed to **minimise overhead**
 - They commonly use **heuristic approaches**
 - That is, **none** of the algorithms discussed here are **optimal algorithms**
- Assume a disk with 36 cylinders, numbered 1 to 36

Disk Scheduling

First-Come, First-Served

- **First come first served:** process the requests in the order that they arrive
- Consider the following sequence of disk requests (cylinder locations):
11 1 36 16 34 9 12
- In the order of arrival (FCFS) the total length is:
 $|11-1| + |1-36| + |36-16| + |16-34| + |34-9| + |9-12| = 111$

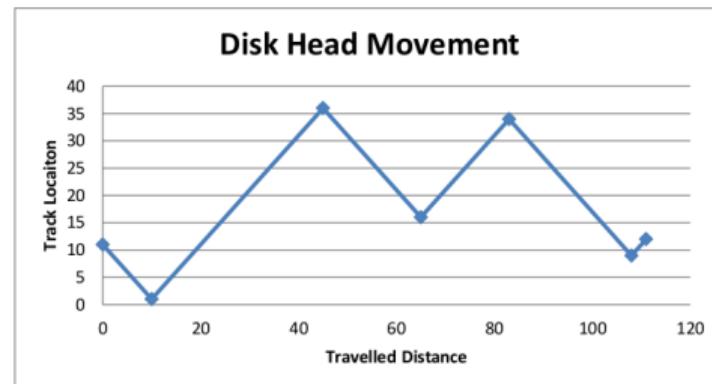


Figure: Head movement for FCFS

Disk Scheduling

Shortest Seek Time First

- **Shortest seek time first** selects the request that is **closest to the current head position**
- In the order “**shortest seek time first**” (SSTF) we gain approx. 50%:

11 1 36 16 34 9 12:

$$|11-12| + |12-9| + |9-16| + |16-1| + |1-34| + |34-36| = 61$$

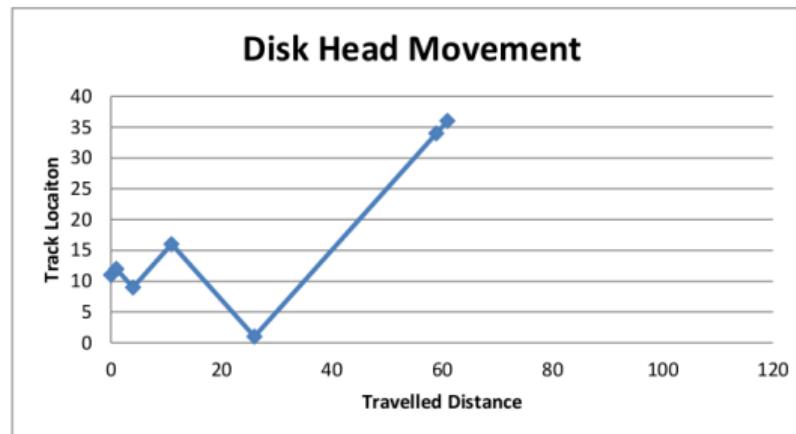


Figure: Head movement for shortest seek time

Disk Scheduling

Shortest Seek Time First

- Shortest seek time first could result in **starvation**:
 - The **arm stays in the middle of the disk** in case of heavy load (edge cylinders are poorly served)
 - Continuously arriving requests for the same location could **starve** other regions

Disk Scheduling

SCAN

- “Lift algorithm, **SCAN**”: **keep moving in the same direction** until end is reached (start upwards):
 - It continues in the current direction, **servicing all pending requests** as it passes over them
 - When it gets to the **last cylinder**, it **reverses direction** and services all the pending requests (until it reaches the first cylinder)
- (Dis-)advantages include:
 - The **upper limit** on the “waiting time” is $2 \times$ number of cylinders, i.e. **no starvation occurs**
 - The **middle cylinders are favoured** if the disk is heavily used (max. wait time is N tracks, $2N$ for the cylinders on the edge)

Disk Scheduling

SCAN

- “Lift algorithm, SCAN”:

11 1 36 16 34 9 12:

$$|11-12| + |12-16| + |16-34| + |34-36| + |36-9| + |9-1| = 60$$

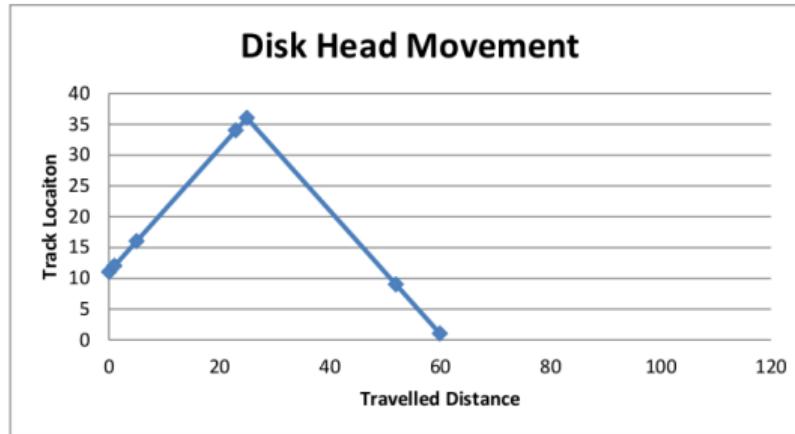


Figure: Head movement for SCAN

Disk Scheduling

Other SCAN variations: LOOK-SCAN, N-step-SCAN

- Look-SCAN moves to the cylinder containing **the first/last request** (as opposed to the first/last cylinder on the disk with SCAN)
- Seekes are **cylinder by cylinder** and one cylinder contains multiple tracks
- The arm can **stick to a cylinder**
- **N-step-SCAN** only services N requests every single sweep.

Disk Scheduling

C-SCAN

- Once the outer/inner side of the disk is reached, the **requests at the other end of the disk have been waiting longest**
- SCAN can be improved by using a **circular scan** approach \Rightarrow C-SCAN
 - The disk arm moves in **one direction** servicing requests until the **last cylinder** is reached
 - It **reverses direction** but **does not service requests** when returning
 - Once it gets back to the **first cylinder** it reverses direction and **services requests**
 - It is **fairer** and **equalises response times** across a disk
- The C-SCAN algorithm (for 11 1 36 16 34 9 12):
 $|11-12| + |12-16| + |16-34| + |34-36| + |36-1| + |1-9| = 68$

Disk Scheduling

Observations

- Look-SCAN and variations are reasonable choices for the algorithms
- Performance of the **algorithms is dependent on the requests/load of the disk**
 - One request at a time \Rightarrow FCFS will perform equally well as any other algorithm
- **Optimal algorithms** are difficult to achieve if requests arrive over time (they need perfect knowledge of information)

Disk scheduling in Unix/Linux

Modifying the disk scheduler

- In Linux, we can modify the disk scheduler by modifying the file:
`/sys/block/sda/queue/scheduler`
- We have got three policies:
 - noop: this is FCFS
 - deadline: N-step-SCAN
 - cfq: Complete Fairness Queueing from Linux.
- The one between brackets is the current policy.

```
pszgd@severn:~$ cat /sys/block/sda/queue/scheduler
noop [deadline] cfq
```

Rotational Hard Drives

Driver Caching

- For most current drives, **the time required to seek a new cylinder is more than the rotational time** (remember **pre-paging** in this context!)
- It makes sense, therefore, to **read more sectors than actually required**
 - **Read** sectors during the **rotational delay** (i.e. that accidentally pass by)
 - **Modern controllers read multiple sectors** when asked for the data from one sector:
track-at-a-time caching

SSD drives

Architecture

- Solid State Drives (SSDs):
 - Have **no moving parts**, store data using **single level** (SLC), **multiple level** (MLC), **triple level** (TLC) electrical circuits, and suffer from **wear out** and **disturbance**
 - Are organised into **banks**, **blocks**, **pages** and have some **volatile cache memory** (buffering, mapping tables)
 - Often use multiple **banks in parallel** to improve performance

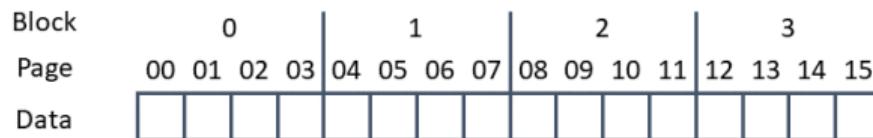


Figure: Layout of an SSD

SSD drives

Architecture

- Solid State Drives (SSDs):

- Have **no moving parts**, store data using **single level** (SLC), **multiple level** (MLC), **triple level** (TLC) electrical circuits, and suffer from **wear out** and **disturbance**
- Are organised into **banks**, **blocks**, **pages** and have some **volatile cache memory** (buffering, mapping tables)
- Often use multiple **banks in parallel** to improve performance

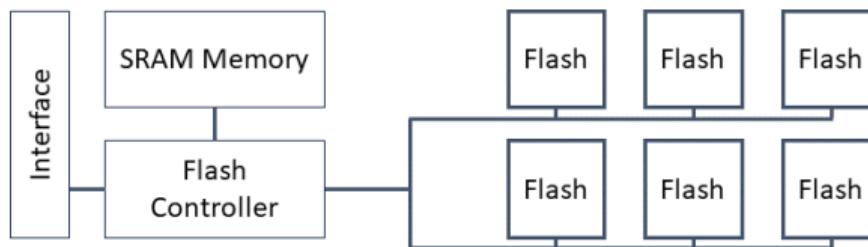


Figure: Layout of an SSD

SSD drives

Reads/Writes

- The **Flash Translation Layer** that maps **logical blocks** onto **physical pages**
- The following operations are supported:
 - Read: uniformly fast random access to any **page** to any location (10s of microseconds)
 - Erase: entire **blocks** containing multiple pages (milliseconds magnitude)
 - Program: write a **page** (100s of microseconds, block must be erased first)

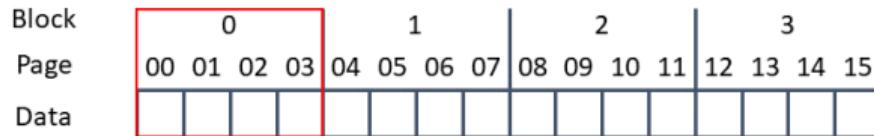


Figure: SSD Write Operation

SSD drives

Reads/Writes

- The **Flash Translation Layer** that maps **logical blocks** onto **physical pages**
- The following operations are supported:
 - Read: uniformly fast random access to any **page** to any location (10s of microseconds)
 - Erase: entire **blocks** containing multiple pages (milliseconds magnitude)
 - Program: write a **page** (100s of microseconds, block must be erased first)

Block	0			
Page	00	01	10	11
Data	00101001	10100101	11101011	00001010

Figure: SSD Write Operation

SSD drives

Reads/Writes

- The **Flash Translation Layer** that maps **logical blocks** onto **physical pages**
- The following operations are supported:
 - Read: uniformly fast random access to any **page** to any location (10s of microseconds)
 - Erase: entire **blocks** containing multiple pages (milliseconds magnitude)
 - Program: write a **page** (100s of microseconds, block must be erased first)

Block	0			
Page	00	01	10	11
Data	11111111	11111111	11111111	11111111

Figure: SSD Write Operation

SSD drives

Reads/Writes

- The **Flash Translation Layer** that maps **logical blocks** onto **physical pages**
- The following operations are supported:
 - Read: uniformly fast random access to any **page** to any location (10s of microseconds)
 - Erase: entire **blocks** containing multiple pages (milliseconds magnitude)
 - Program: write a **page** (100s of microseconds, block must be erased first)

Block	0			
Page	00	01	10	11
Data	10101010	11111111	11111111	11111111

Figure: SSD Write Operation

SSD drives

Direct Mapping

- **Logical pages** (seen by the OS) are **directly mapped** on to **physical pages**
 - ① Read the entire block for the given page
 - ② Erase the entire block
 - ③ Write the new page and remaining old pages back
- **Write performance is bad** (write amplification) and **wear is increased** (some blocks are used more than others) ⇒ a different **log structured approach** is needed

Device	Read (μ s)	Program (μ s)	Erase (μ s)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure: SSD Performance (from Arpaci-Dusseau)

SSD drives

Direct Mapping

Device	Random		Sequential	
	Reads (MB/s)	Writes (MB/s)	Reads (MB/s)	Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure: SSD / HDD Comparison (from Arpaci-Dusseau)

Test Your Understanding

Problem (from Tanenbaum)

Disk Access Times

Disk requests come in to the disk driver for cylinders 10, 22, 20, 2, 40, 6 and 38, in that order.

- A seek takes 6ms per cylinder.
- How much seek time is needed for: FCFS, SSTF and Look-SCAN (initially moving upward)
- In all cases, the arm is initially at cylinder 20.

Summary

Take-Home Message

- Construction and organisation of **rotational hard drives**
- **Access times** of rotational hard drives
- **Disk scheduling** & caching
- **Solid state drives**



valid for 65 minutes from 2:55pm
generated 2023-10-17 03:13

Figure: Attendance Monitoring

Operating Systems and Concurrency

File Systems 2
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

- **Challenges** arising from the inherent nature of devices
 - **Delays** due to seek time, rotational latency, transfer times (hard drives)
 - **Block erasing** for page writing (SSDs)
- Two level **performance improvement**:
 - Disk scheduling and cylinder skew
 - File system implementation

File Systems

What Can an OS Do For Me?

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class Demo1 {
    public static void main(String[] args) throws IOException {
        FileWriter fw =
            new FileWriter(("C:/Program Files (x86)/test.txt"));
        PrintWriter pw = new PrintWriter(fw);
        pw.close();
    }
}
```

File Systems

- **File system abstraction:** the logical file system is mapped onto the physical one (abstraction from the physical level)
- **Abstraction from the device:** uniform view of very different underlying storage mechanism
- **Concurrency:** what if multiple processes **access the file simultaneously**
- **Security:** why is the **access denied**

File Systems

File systems allow data to be stored, located, and retrieved easily and efficiently.

Disk Layout

Boot Sector and Partitions

- Drive is a **collection of sectors** (0 - N)
- **Boot record** is located at start of the drive:
 - Used to boot the computer (BIOS reads and executes boot sector)
 - Contains **partition table** at its end with **active partition**
 - One partition is listed as **active** containing a **boot block** to **load the operating system**
- The drive is commonly split into **multiple partitions**:
 - A different **file/operating system** may exist on each partition (occasionally none)



Figure: Disk Layout

Disk Layout

Partition Layout (File System Dependent)

- **Boot block** containing code to boot the operating system (for every partition irrespective containing an OS)
- **Super block** containing stats about the partition (partition size, number of FCBs, location of free list, ...)
- **Free space management** contains **data structures** to indicate **free FCBs** or **data blocks**
- **Meta data** or **File Control Blocks** (e.g. i-nodes)
- **Data blocks**, including the **root directory** (the top of the file-system tree)



Figure: Disk Layout

File Systems

OS Abstractions

- A **user view** that defines a file system in terms of the **abstractions** (system calls) that the operating system provides (**files** and **directories**)
- An **implementation view** that defines the file system in terms of its **low level implementation**

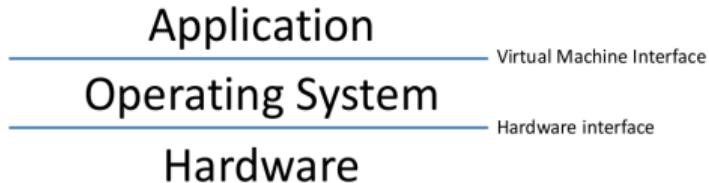


Figure: User vs. Implementation View

File System Layers

Logical Layers

- Shared layers:
 - **I/O control** interacts with the **device controller/registers** (device drivers, interrupt handlers)
 - **Basic file system** instructs device drivers “blocks”, **schedules I/O**, and manages **buffers** and **caches** for (meta-)data
- File system specific layers:
 - **File organisation** models **logical blocks** for files and **free space**
 - **Logical file system** manages **file control blocks**, **directory structures**, and **protection**
- **Application programs** define the structure of the files



Figure: File System Layers

Files

Types

- Both Windows and Unix (including OS X) have **regular files** and **directories**:
 - **Regular files** contain user data in **ASCII** or **binary** (well defined) **format**
 - **Directories** group files together (but are files on an implementation level)
- Unix also has **character** and **block special files**:
 - **Character special files** are used to model **serial I/O devices** (e.g. keyboards, printers)
 - **Block special files** are used to model, e.g. hard drives
- Files are **sequential, random (direct) access, indexed access**

Files

File Control Blocks & Tables

- **File control blocks (FCBs)** are **kernel** data structures
 - Allowing user applications to access them directly could **compromise their integrity**
 - **System calls** enable a **user application** to ask the **operating system** to carry out an **action** on its behalf (in kernel mode)
- FCBs are kept in the **per process** and **system wide open file table** (array) indexed using a process specific file handle

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure: File control block (FCB) (Silberschatz)

Files

File Control Blocks & Tables

- The per **process file table** contains **process specific information**, e.g.:
 - All **files currently open** to the process
 - Read/write/current **pointers**
 - A **reference** to the relevant entry in the system wide file table
- The **system wide file table** contains **general information**, e.g.:
 - One entry per open file
 - Location on disk
 - Access times
 - Reference count**

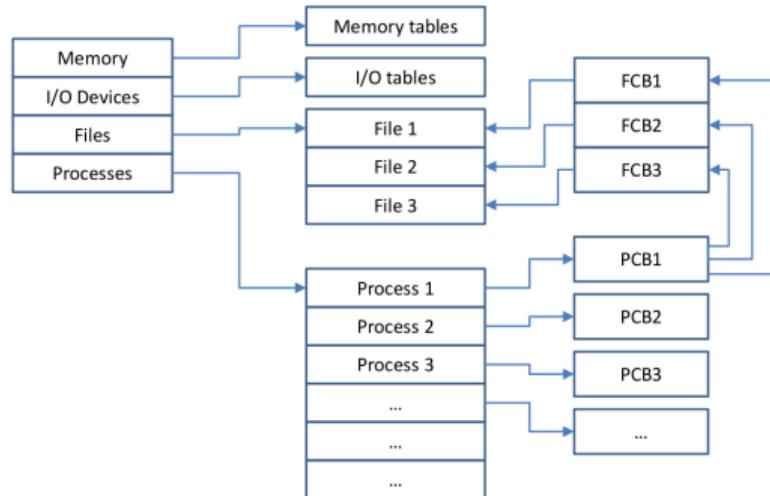


Figure: File Tables

Files

System Calls

- **System calls** for file manipulation include: `create()`, `open()`, `close()`, `read()`, `write()`, ...
- For instance:
 - The `open()` system call:
 - 1 Maps the **logical name** onto the **low level name** identifying the **file control block**
 - 2 **Retrieves** the “FCB” (from the drive)
 - 3 Adds it to the **process/system open file table** (increments the **reference count**)
 - 4 Returns a **process specific file handle** (index into the table)
 - The `close()` system call:
 - 1 Decrement the **reference count**
 - 2 **Synchronise FCB** with disk
 - 3 **Removes FCB** from process/system file tables (when reference count = 0)

Files

System Calls – Illustration Using “strace” (dtruss on MacOS)

```
# command = strace cat helloWorld.txt > /dev/null

execve("/usr/bin/cat", ["cat", "helloWorld.txt"], 0x7fffccb21658 /* 34 vars */) = 0
...
open("helloWorld.txt", O_RDONLY)      = 3
...
read(3, "Hello World\n", 1048576)    = 12
write(1, "Hello World\n", 12)        = 12
read(3, "", 1048576)                = 0
...
close(3)                           = 0
close(1)                           = 0
close(2)                           = 0
exit_group(0)                      = ?
```

Directories

Implementations

- Directories are special files that group files together and of which the **structure is defined** by the **file system**
 - A **bit is set** to indicate that they are directories
 - They map **human readable “logical” names** onto **unique identifiers** for **file control blocks** that detail **physical locations** and **file attributes**
 - Two approaches exist:
 - All attributes are **stored in the directory file** (e.g. file name, disk address – Windows)
 - **A pointer** to the data structure (e.g. **i-node**) that contains the file attributes (Unix)

Figure: Directory Implementations

Directories

Implementations

- Directories are special files that group files together and of which the structure is defined by the file system
 - A bit is set to indicate that they are directories
 - They map human readable “logical” names onto unique identifiers for file control blocks that detail physical locations and file attributes
 - Two approaches exist:
 - All attributes are stored in the directory file (e.g. file name, disk address – Windows)
 - A pointer to the data structure (e.g. i-node) that contains the file attributes (Unix)

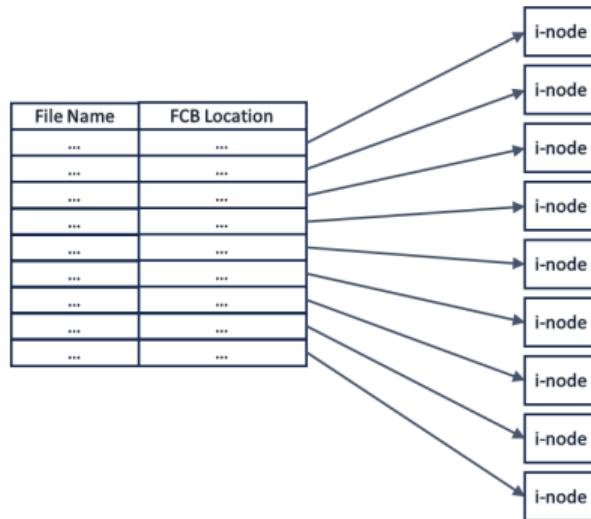


Figure: Directory Implementations

Directories

Implementations

- Directories enable to build **directed acyclic-graphs** (generalisation of a **tree structure** – links can compromise this)

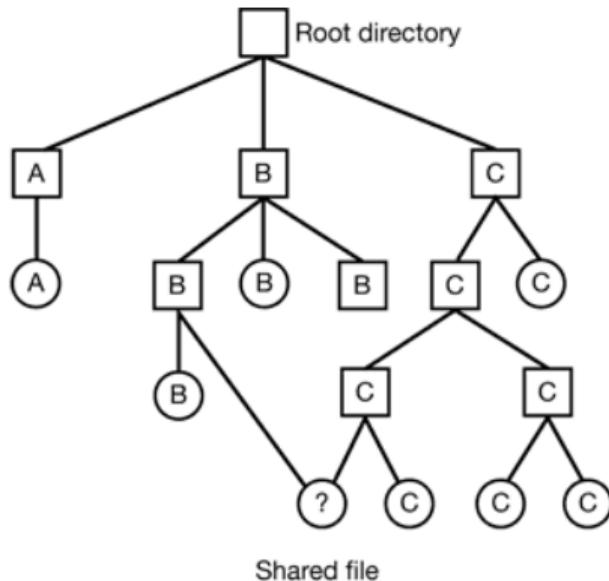


Figure: DAG Directory Implementation
(Tanenbaum)

Directories

System Calls

- Similar to files, **directories** are manipulated using **system calls**
 - create/delete: a new directory is created/deleted
 - opendir, closedir: add/free directory to/from internal tables
 - readdir, return the next entry in the directory file
 - Others: rename, link, unlink, list, update
- Common **operations** include, creating, deleting, searching, listing, **traversing**, ...

File Access

Reading /home/pszgd/COMP2007/helloWorld.txt

```
#Steps to read /home/pszqd/COMP2007/helloWorld.txt
```

- read FCB for /
 - find FCB location for /home
 - read FCB for /home
 - find FCB location of /home/pszgd
 - read FCB for /home/pszgd
 - find FCB location for /home/pszgd/COMP2007
 - read FCB for /home/pszgd/COMP2007
 - find FCB location for /home/pszgd/COMP2007/helloWorld.txt
 - read FCB /home/pszgd/COMP2007/helloWorld.txt
 - => update per process/system file tables
 - read data for /home/pszgd/COMP2007/helloWorld.txt
 - close the file
 - => update per process/system file tables

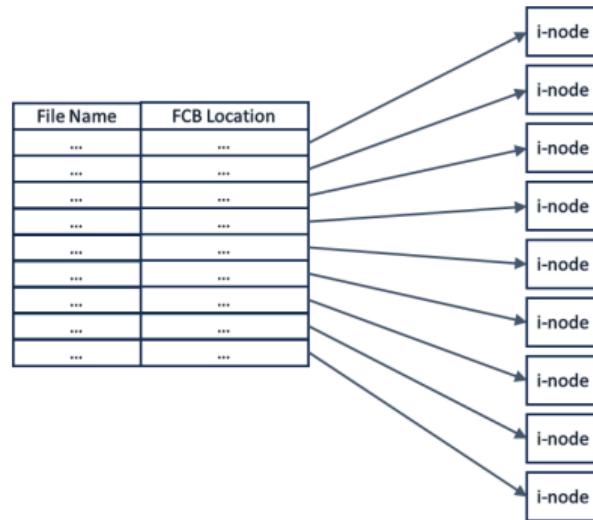


Figure: Directory Implementations

Directories

System Calls

- Retrieving a file comes down to **searching a directory file** as fast as possible
- A **simple random order of directory** entries might be insufficient (search time is linear as a function of the number of entries)
- Indexes or **hash tables** can be used for large directories

Free Space Management

Bitmaps

- Similar to memory management, **bitmaps** and **linked lists** can be used for free space management
- **Bitmaps** represent each block by a single bit in a map
 - The **size of the bitmap** grows with the size of the disk but is constant for a given disk
 - Bitmaps take **comparably less space than linked lists**



Figure: Disk Layout

Free Space Management

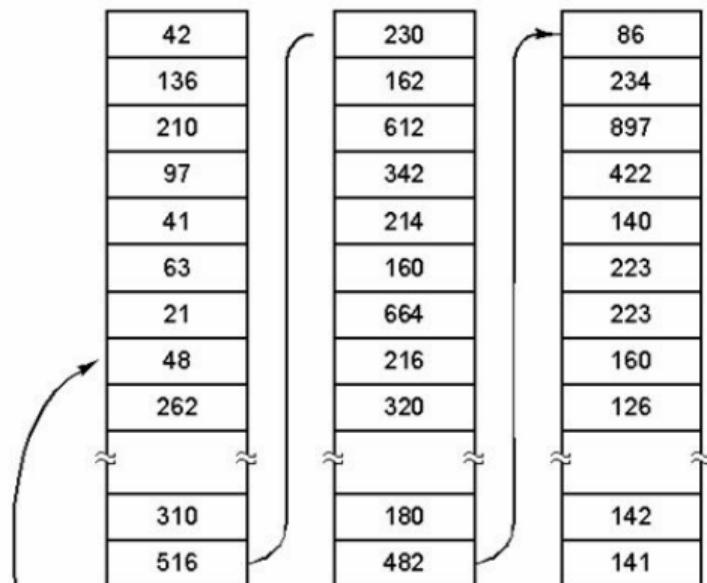
Linked Lists

- **Linked List** of list free groupings
 - Use **free blocks** to hold the **locations of the free blocks** (hence, they are no longer free)
 - The size of the list **grows with the size of the disk** and **shrinks with the size of the blocks**
 - E.g., with a 1KB block a 32-bit/4 byte disk block number, each block will hold 255 free blocks (one for the pointer to the next block)
 - Since the free list shrinks when the disk becomes full, this is not wasted space
 - **Blocks are linked together**, i.e., multiple blocks list the free blocks
- Linked lists can be modified by **keeping track of the number of consecutive free blocks** for each entry (known as Counting)

Free Space Management

Comparison

Free disk blocks: 16, 17, 18



A 1 KB disk block can hold 256
32-bit disk block numbers

(a)

A bit map

(b)

Free Space Management

Bitmap vs. linked list

- Bitmaps:
 - Require extra space. E.g: If block size = 2^{12} bytes (4KB) and disk size = 2^{30} bytes (1 GB) \Rightarrow bitmap size: $2^{30}/2^{12} = 2^{18}$ (32KB)
 - Keeping it in main memory is possible only for small disks.
- Linked lists:
 - **Grows** with the number of empty blocks
 - **No waste** of disk space (uses empty space)
 - We only need to keep in memory **one block of pointers** (load a new block when need).

Summary

Take-Home Message

- File System Layers and Disk layouts
- Implementation of files, directories, and OS data structures
- Free space management, partitions, boot sectors, etc.



valid for 65 minutes from 8:55am
generated 2023-10-17 03:13

Figure: Attendance Monitoring

Operating Systems and Concurrency

File Systems 3
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

1 **User view** of file systems

- System calls
- Structures, organisation, file types

2 **Implementation view** of file systems

- Disk and partition layout
- File tables
- Free space management
- ...

Disk Scheduling

Observations

- The **layout of the file system** and the **file allocation method** heavily influence the **seek movements**
 - Contiguous files reduce seek movements
- The **OS can prioritise/order requests** when implementing disk scheduling (vs. controller)

Goals for Today

Overview

- File system **implementations**
 - 1 Contiguous
 - 2 Linked lists
 - 3 File Allocation Table (FAT)
 - 4 i-nodes (lookups)
- Hard and soft links

File access

Sequential vs. Random Access

- **Files** will be composed of a number of **blocks**
- Files are **sequential or random access** (essential for database systems)

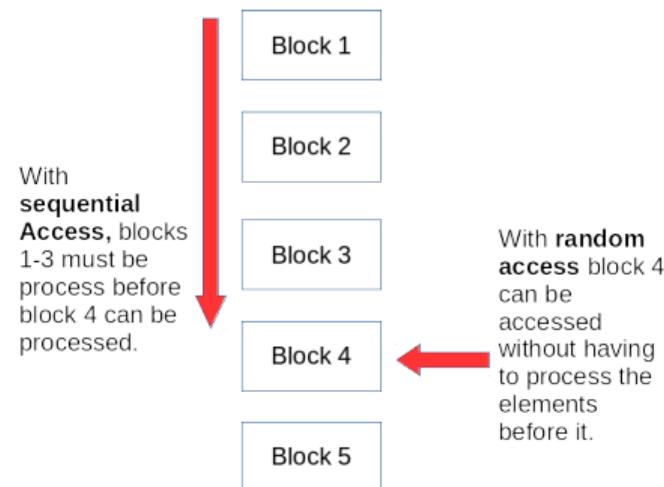


Figure: Types of access to a file

Contiguous Allocation

Concept

- **Contiguous file systems** are similar to **dynamic partitioning** in memory allocation:
 - Each file is stored in a single group of **adjacent blocks** on the hard disk
 - E.g. 1KB blocks, 100KB file, we need 100 contiguous blocks
- Allocation of free space can be done using **first fit**, **best fit**, **next fit**, etc.

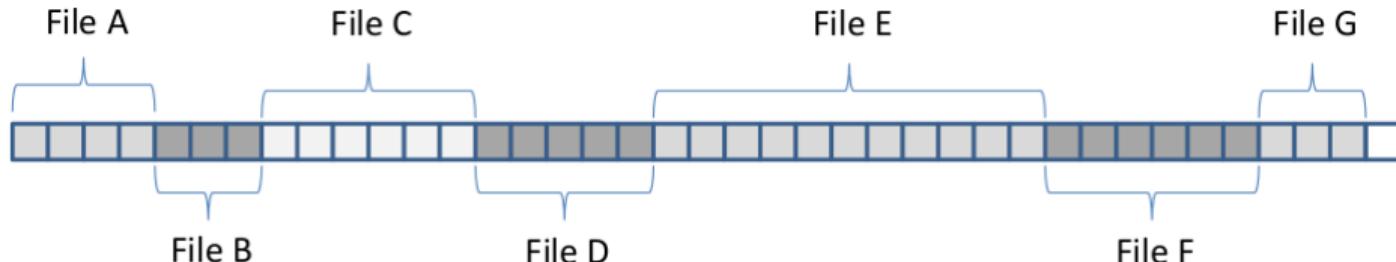


Figure: External Fragmentation when Removing Files

Contiguous Allocation

Advantages

- **Easy to implement:** only location of the first block and the length of the file must be stored (in the FCB)
- **Optimal read/write performance:** blocks are co-located/clustered in nearby/adjacent sectors (seek time is minimised)

File	Start	Length
req1.c	0	12
req1.o	30	10
req1	15	5
req1.txt	41	20

Figure: Directory table

Contiguous Allocation

Disadvantages

- **Disadvantages** of contiguous file systems include:
 - The **exact size** of a file (process) is not always known beforehand
 - **Allocation algorithms** are needed to decide which free blocks to allocate (e.g., first fit, best fit)
 - Deleting a file results in **external fragmentation** which requires **de-fragmentation** (which is **slow**)
- Contiguous allocation used for **CD-ROMS/DVDs**
 - External fragmentation is less of an issue since they are **write once**

Linked Lists

Concept

- To avoid external fragmentation, files are stored in **separate blocks** (similar to paging) that are **linked to one another**
- Only the **address of the first** is stored in the FCB
- Each block contains a **data pointer** to the next block (which takes up space)

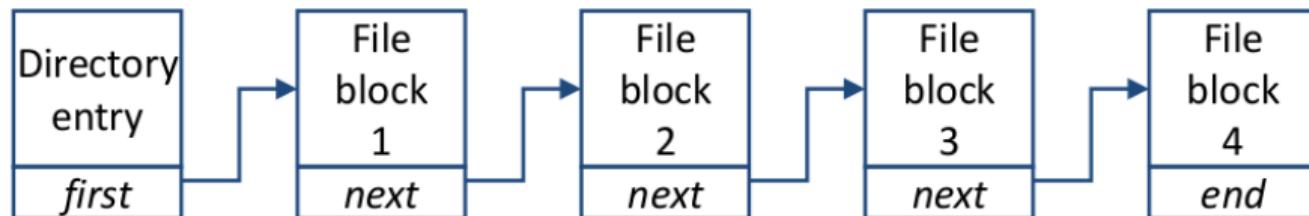


Figure: Linked List File Storage

Linked Lists

Advantages

- **Easy to maintain:** only the first block (address) has to be maintained in the directory entry
- Files can **grow dynamically:** new blocks/sectors are appended at the end
- Similar to paging, there is **no external fragmentation**
- **Sequential access** is straightforward, although **more seek operations** may be required (non-contiguous)

Linked Lists

Disadvantages

- **Random access** is very **slow**
- **Internal fragmentation:** the last half of the block is unused (on average)
 - Internal fragmentation will reduce for **smaller block sizes**
- May result in **random (slow) disk access**
 - **Larger blocks** (containing multiple sectors) will improve speed (but increase internal fragmentation)

Linked Lists

Disadvantages (Cont'ed)

- The data within a block is **no longer a power of 2**
- **Reduced reliability:** if one block is corrupt/lost, access to the rest of the file is lost

File Allocation Tables

Key Concept

- Store the linked-list pointers in a **separate index table**, called a **File Allocation Table (FAT)** (loaded in memory)

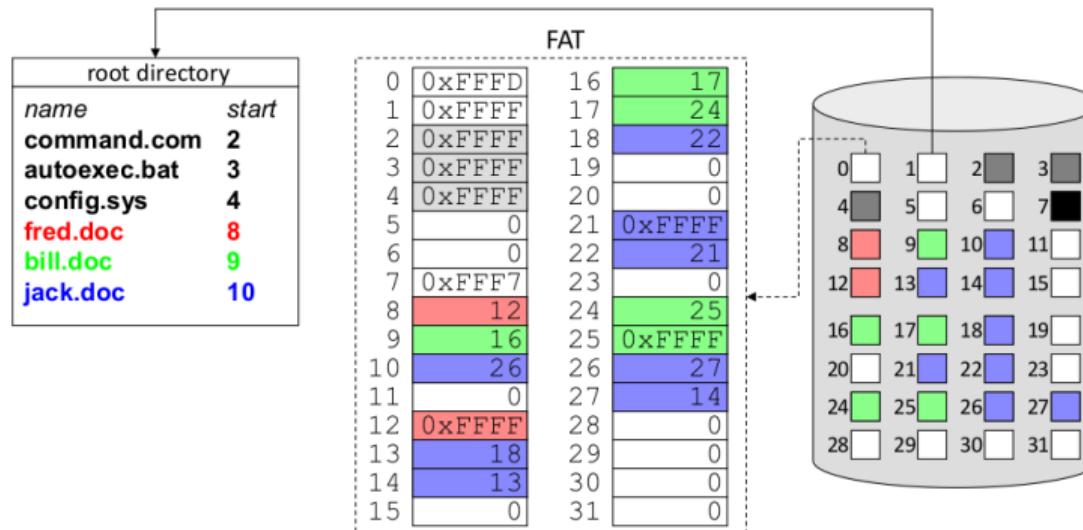


Figure: File Allocation Tables

File Allocation Tables

Advantages and disadvantages

- Advantages:
 - **Block size remains power of 2** (no space is lost in block due to the pointer)
 - **Index table** can be **kept in memory** allowing fast non-sequential/random access (table access itself remains sequential)
- Disadvantages:
 - The **size** of the **file allocation table** grows with the number of blocks/disk size
 - **200 million entries** are required for a **200GB disk** with a **1KB block size** (**800MB** at 4 bytes per entry)

i-nodes

Concept

- Each file has a small data structure (on disk) called **i-node** (index-node) that contains its attributes and block pointers.
 - In contrast to FAT, an i-node is **only loaded when the file is open** (stored in system wide open file table)
 - If every i-node consists of n bytes, and at most k files can be open at any point in time, at most $n \times k$ bytes of main memory are required
- i-nodes are composed of **direct block pointers** (usually 10), **indirect block pointers**, or a combination thereof (e.g., similar to **multi-level page tables**)

i-nodes

Concept

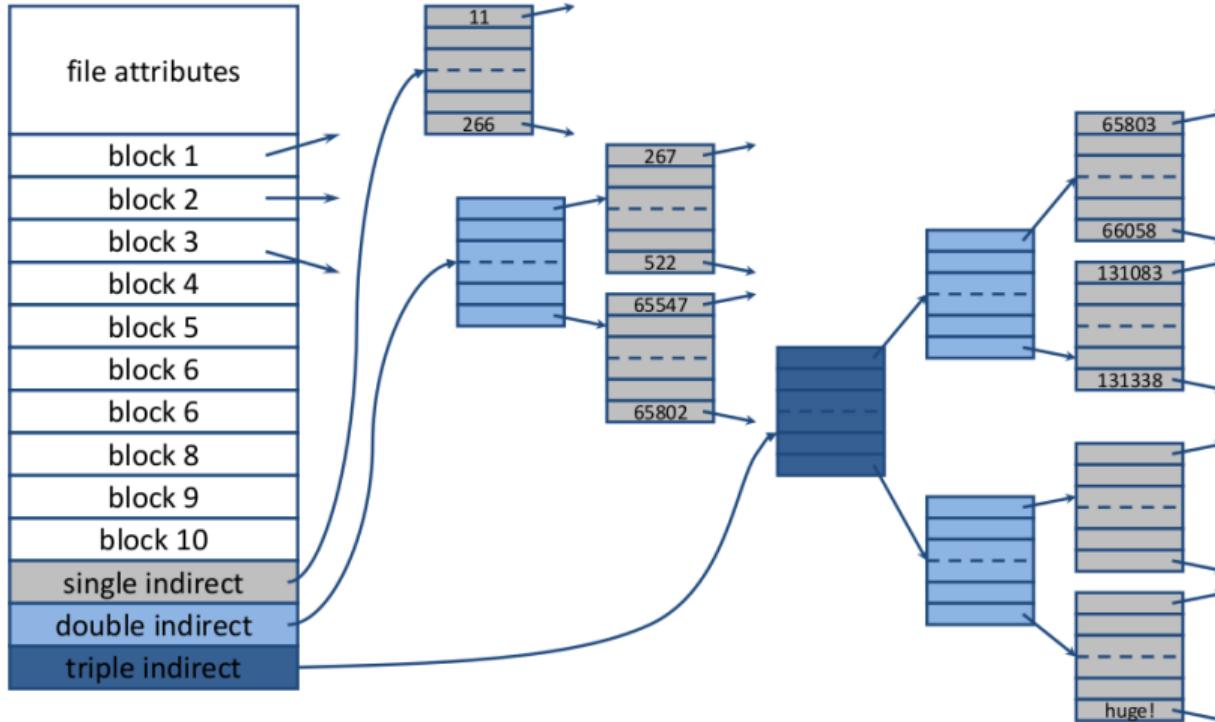


Figure: File Storage Using I-Nodes

i-nodes

Concept

- Assuming a block size of 1KB, and 32-bits disk address space
- With **only direct block pointers** the maximum file size is $1KB \times \text{NUMBER_OF_DIRECT_BLOCK_POINTERS}$
- With a **single indirect block** the maximum file size is $(10 + 256) \times 1KB = 266KB$
- With a **double indirect block** 256 blocks containing 256 pointers each
 - The maximum file size is $(10 + 256 + 256^2) \times 1KB = 65802KB$
- If we need files larger than this, we will need a triple indirect blocks for a maximum file size of $(10 + 256 + 256^2 + 256^3) \times 1KB$

Directories

Implementation with i-nodes

- In UNIX/Linux/MacOS:
 - All **metadata** for a file (type, size, date, owner, and block pointers) is stored in an **i-node**
 - **Directories** are very simple data structures composed of file name and a pointer to the i-node
- Directories are no more than a special kind of **file**, so they have their own i-node

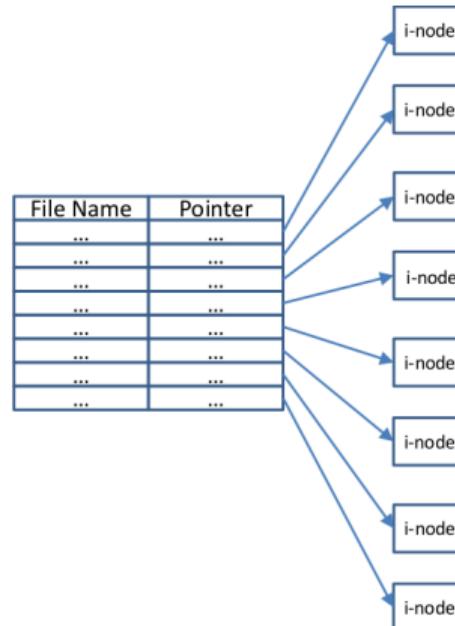


Figure: i-node Directory Structure

File System Comparison

Contiguous vs. Linked vs. Indexed

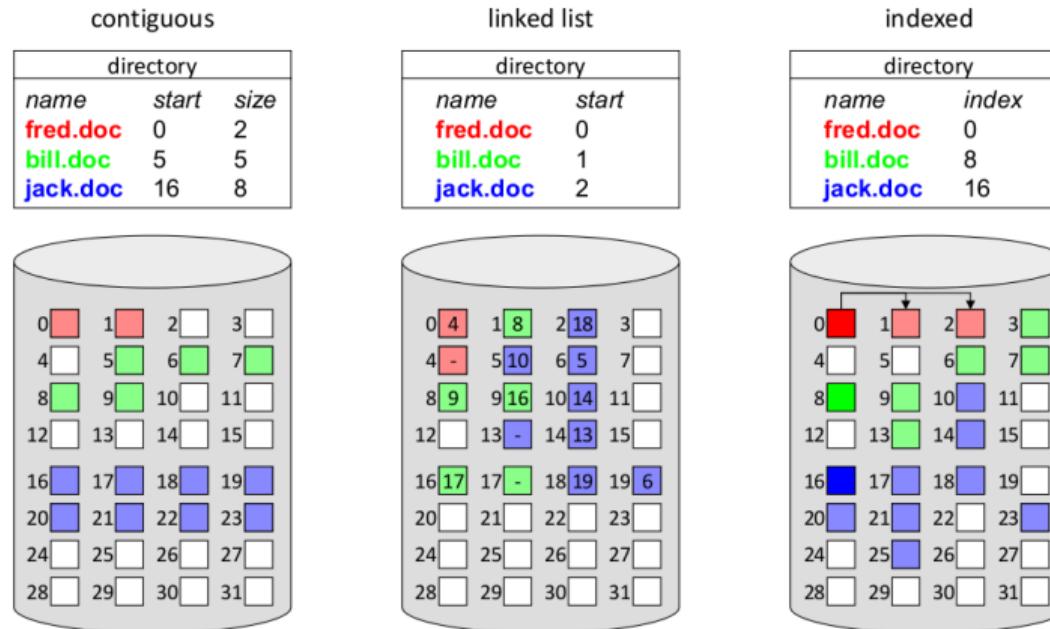


Figure: Contiguous vs. Linked List (or FAT) vs. i-nodes (or indexed)

i-nodes

Lookups

- Opening a file requires the disk blocks to be located
 - Absolute file names** are located relative to the root directory
 - Relative file names** are located based on the current working directory
- E.g. Try to locate /usr/gdm/mbox

/ inode	/ contents	/usr inode	/usr contents	/usr/gdm inode	/usr/gdm contents
1	2	6	132	26	406
size mode times	1 . 1 .. 4 bin 7 dev 14 lib 9 etc 6 usr 8 tmp	size mode times	6 . 1 .. 19 fred 30 bill 51 jack 26 gdm 45 cfi	size mode times	26 . 6 .. 64 research 92 teaching 60 mbox 17 grants
2		132		406	

Figure: Locating a File

i-nodes

Lookups

Locate the root directory of the file system

- Its i-node sits on a fixed location at the disk
(the directory itself can sit anywhere)

Locate the directory entries specified in the path:

- Locate the i-node number for the first component
(directory) of the path that is provided
- Use the i-node number to index the i-node table
and retrieve the directory file
- Look up the remaining path directories by
repeating the two steps above

Once the file's directories have been located, locate
the file's i-node and cache it into memory

i-nodes: Sharing files between directories

Hard and Soft Links

- There are two approaches to **share a file**, e.g. between directory B and C, where C is the 'real' owner:
 - **Hard links:** maintain two (or multiple) references to the same i-node in B and C
 - the i-node link reference counter will be set to 2
 - **Symbolic links:**
 - The owner maintains a reference to the i-node in, e.g., directory C
 - The "referencer" maintains a small file (that has its own i-node) that contains the location and name of the shared file in directory C
- What is the **best approach?** ⇒ both have advantages and disadvantages

i-nodes: Sharing files between directories

Hard Links

- Hard links are the **fastest way of linking files!**
- Disadvantages of hard links:
 - Assume that the owner of the file **deletes** it:
 - If the i-node is also deleted, any hard link will, in the best case, **point to an invalid i-node**
 - If the i-node gets **deleted** and “**recycled**” to point to an other file, the hard links will **point to the wrong file!**
 - The only solution is to **delete the file**, and **leave the i-node intact** if the “**reference count**” is larger than 0 (the original owner of the file still gets “charged” for the space)

i-nodes: Sharing files between directories

Hard Links

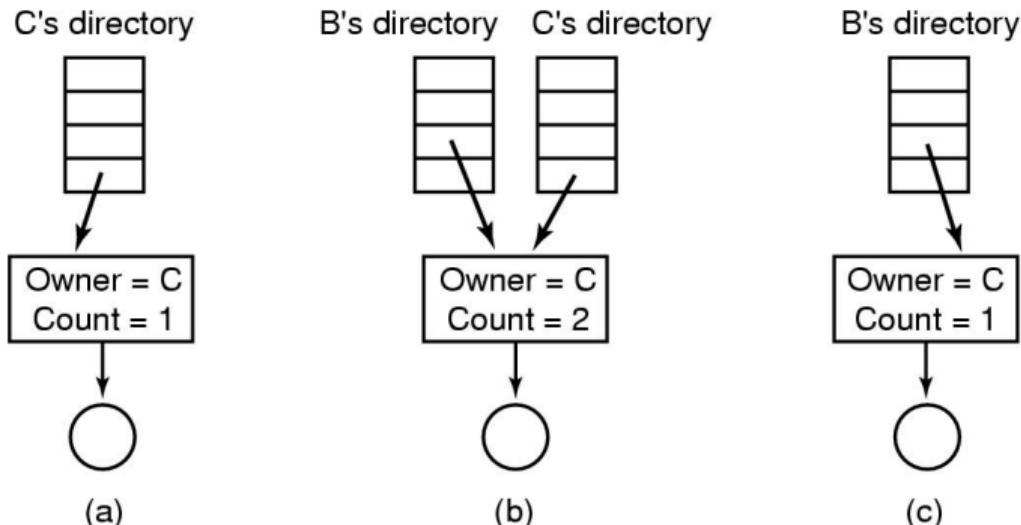


Figure: (a) Before creating a link. (b) After creating a link. (c) after the original owner removes the file
(Tanenbaum)

i-nodes

Soft Links

- Disadvantages of soft links:
 - They result in an **extra file lookup** (once the link file has been found, the original file needs to be found as well)
 - They require an **extra i-node** for the link file
- Advantages of symbolic links:
 - There are **no problems with deleting** the original file ⇒ the file simply does not exist any more
 - They can **cross the boundaries of machines**, i.e. the linked file can be located on a different machine

i-nodes

Hard and Soft Links in Unix

```
[pszgd@severn ~]$ pwd  
/home/pszgd  
[pszgd@severn ~]$ ls -i labs/  
3250013 req1b.c  
[pszgd@severn ~]$ ln labs/req1b.c hardLink  
[pszgd@severn ~]$ ln -s labs/req1b.c softLink  
[pszgd@severn ~]$ ls -ali hardLink softLink  
3250013 hardLink  
3250021 softLink -> labs/req1b.c  
[pszgd@severn ~]$ rm labs/req1b.c
```

File System Examples

Unix vs. Windows

- The Unix V7 File System:
 - **Tree structured** file system with links
 - Directories contain **file names** and **i-node numbers**
 - i-nodes contain **user and system attributes** (e.g. count variable)
 - One single, double, and triple **indirect blocks** can be used
- More sophisticated File Systems were developed later (e.g. ext3/4)
- Windows:
 - Up to XP used FAT-16 and FAT-32
 - From XP moved to NTFS (64 bits) because of file size limitations
 - NTFS uses File Tables, with bigger i-nodes that can also contain small files and directories
 - More recently uses ReFS

Recap

Take-Home Message

- Contiguous, linked list, FAT and i-nodes as file system implementations.
- Lookups with i-nodes.
- Hard and Soft Links

Test Your Understanding

File Systems

Exercises

With i-nodes, the maximum file size that we can have depends on the block size and the number of indirections.

- Assuming a 32-bit disk address space, what is the maximum (theoretical) file size for a FAT file system with a drive of 500GB and a block size of 1KB? (without accounting for directory metadata)
- The most used implementation of FAT is known as FAT-32. Investigate why there is a theoretical limitation of 4GB per file (and sometimes even less than 2GB).

Operating Systems and Concurrency

File Systems 4
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture: file system implementations

- **Contiguous implementations** are easy, fast, but result in external fragmentation
- **Linked lists** are sequential, and have block sizes $\neq 2^n$ (page sizes are 2^n)
- **FAT** have block sizes $= 2^n$, but the table becomes prohibitively large
- **I-nodes** are only loaded when the file is open, contain attributes and multiple block levels

Goals for Today

Overview

- File system **paradigms** (which use an underlying file system)
 - Log structured file systems
 - Journaling file systems
- The Unix/Linux **virtual file system**

Log Structured File System

Context

- Consider the creation of a **new file** on a Unix system:
 - 1 Allocate, initialise and write the i-node** for the file
 - i-nodes are usually located at the start of the disk
 - 2 Update and write the directory entry** for the file
 - 3 Write the data** to the disk
- The corresponding blocks are not necessarily in **adjacent locations**
- Also in **linked lists/FAT** file systems blocks can be **distributed across the disk**



Log Structured File System

Context

- Device characteristics:
 - Due to seek and rotational delays, **hard disks are slow**
 - **SSDs** suffer from **write amplification** (block must be erased), **write disturbance**, and **wear out**
- A **log structured file system** copes better with inherent **device characteristics**:
 - Aims to improve speed of a file system on a traditional hard disk by **minimising head movements** and **rotational delays**
 - Reduces **write amplification**, **disturbance** and **wear out**

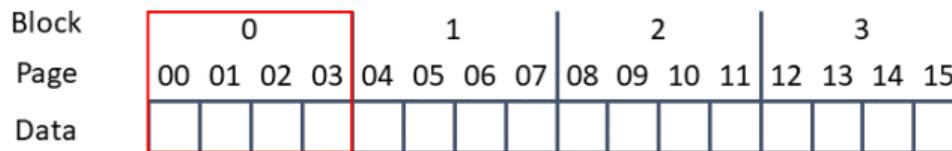


Figure: SSD Write Operation

Log Structured File System

Context

- Device characteristics:
 - Due to seek and rotational delays, **hard disks are slow**
 - **SSDs** suffer from **write amplification** (block must be erased), **write disturbance**, and **wear out**
- A **log structured file system** copes better with inherent **device characteristics**:
 - Aims to improve speed of a file system on a traditional hard disk by **minimising head movements** and **rotational delays**
 - Reduces **write amplification**, **disturbance** and **wear out**

Block	0			
Page	00	01	10	11
Data	00101001	10100101	11101011	00001010

Figure: SSD Write Operation

Log Structured File System

Context

- Device characteristics:
 - Due to seek and rotational delays, **hard disks are slow**
 - **SSDs** suffer from **write amplification** (block must be erased), **write disturbance**, and **wear out**
- A **log structured file system** copes better with inherent **device characteristics**:
 - Aims to improve speed of a file system on a traditional hard disk by **minimising head movements** and **rotational delays**
 - Reduces **write amplification**, **disturbance** and **wear out**

Block	0			
Page	00	01	10	11
Data	11111111	11111111	11111111	11111111

Figure: SSD Write Operation

Log Structured File System

Context

- Device characteristics:
 - Due to seek and rotational delays, **hard disks are slow**
 - **SSDs** suffer from **write amplification** (block must be erased), **write disturbance**, and **wear out**
- A **log structured file system** copes better with inherent **device characteristics**:
 - Aims to improve speed of a file system on a traditional hard disk by **minimising head movements** and **rotational delays**
 - Reduces **write amplification**, **disturbance** and **wear out**

Block	0			
Page	00	01	10	11
Data	10101010	11111111	11111111	11111111

Figure: SSD Write Operation

Log Structured File System

Concept

- A **log** is a data structure that is **written to at the end** and treated as a **circular buffer**
- Log structured file systems **buffer read and write operations** (i-nodes, data, etc.) in memory
 - Enables to **write “larger volumes”**
- Once the buffer is full it is “flushed” to the disk and written as **one contiguous segment** at the end of “**a log**”
 - **i-nodes and data** are all written to the **same “segment”**
 - **Finding i-nodes** (traditionally located at the start of the partition) becomes more **difficult**
- An **i-node map** is maintained in memory to quickly find the address of i-nodes on the disk

Log Structured File System

Structure

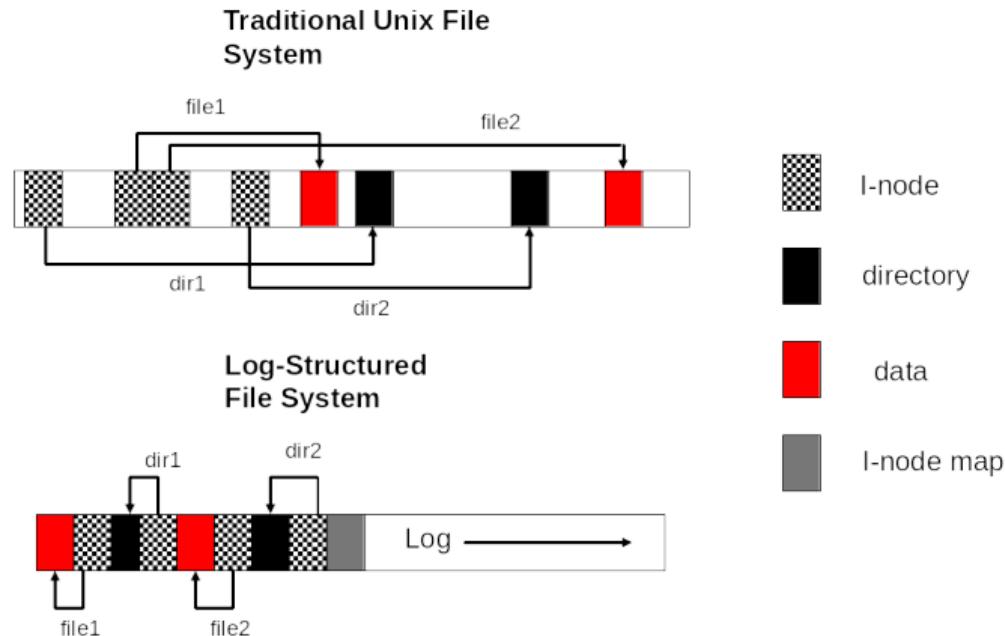


Figure: Blocks written to create two 1-block files: `dir1/file1` and `dir2/file2`

Log Structured File System

Concept: external fragmentation

- A **cleaner thread** (for deleted files) is running in the background and spends its time scanning the log circularly and **compacting** it
- **Deleted** files are marked as free segments and **files being used** right now are written at the end of the log

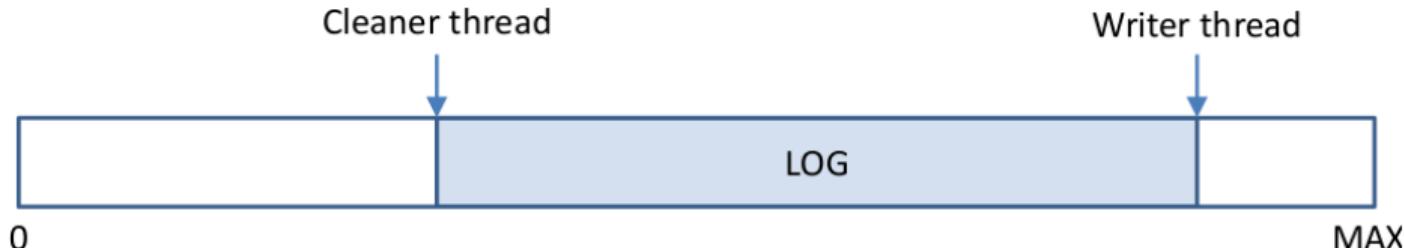


Figure: Log Structured File System

Log Structured File System

Advantages and Disadvantages

- It greatly **increases disk performance** on writes, file creates, deletes but the **cleaner thread** takes additional CPU time
- Writes are **more robust** as they are done as a single operation (multiple small writes are more likely to expose the file system to serious inconsistency)
- Less frequently used for **regular file systems** (because it is highly incompatible with existing file systems), more frequently used for **SSDs**

File System Implementations

Journaling File Systems: Example

- **Deleting a file** consists of the following actions:
 - 1 Remove the file's **directory** entry
 - 2 Add the file's **i-node** to the **pool of free i-nodes**
 - 3 Add the file's **disk blocks** to the **free list**
- It can **go wrong**, for instance:
 - **Directory** entry has been **deleted** and a crash occurs ⇒ i-nodes and disk blocks become inaccessible
 - The **directory** entry and **i-node** have been released and a crash occurs ⇒ disk blocks become inaccessible

File System Implementations

Journaling File Systems: Example

- **Changing the order** of the events does not necessarily resolve the issues
- Journaling file systems aim at **increasing the resilience** of file systems against **crashes** by recording each update to the file system as a transaction

File System Implementations

Journaling File Systems: Concept

- The key idea behind a journaling file system is to **log all events** (transactions) before they take place
 - Write the actions that should be undertaken to a log file
 - Carry them out
 - Remove/commit the entries once completed
- If a **crash** happens in the middle of an action the **entry in the log file will remain present**
- The **log is examined** after the **crash** and used to restore the consistency
- **NTFS, ext3, and ext4** and are examples of journaling file systems

File System Implementations

Virtual File Systems: Concept

- **Multiple file systems** usually co-exist (e.g., NTFS and ISO9660 for a CD-ROM, NFS)
- File systems are **seamlessly integrated** by the operating system's **virtual file systems** (VFS)
- VFS relies on **standard object oriented** principles (or manual implementations thereof), e.g. polymorphism

File System Implementations

Virtual File Systems: Concept

- Consider some code that you are writing, **reading “data records”** (DataObject) from a file
- These records can be stored in **CSV file**, or **XML File**
- How would you make your code resilient against **changes in the underlying data structure?**

File System Implementations

Virtual File Systems: Concept

- Consider some code that you are writing, **reading “data records”** (DataObject) from a file
- These records can be stored in **CSV file**, or **XML File**
- How would you make your code resilient against **changes in the underlying data structure?**
 - You would hide the **implementation** behind **interfaces** using **Data Access Objects** (DAOs)

File System Implementations

Virtual File Systems: Concept

- We can define a **generic interface**, e.g. DataReader, containing a method

```
public DataObject readData();
```

 - In the case of file systems, this would be the **POSIX interface** containing reads, writes, close, etc.
- You would hide the CSV and XML code in **specific implementations** of the DataReader interface, e.g. CSVDataReader and XMLDataReader
 - In the case of file systems this would be the file system implementations

File System Implementations

Virtual File Systems: Concept (Cont'ed)

- You would rely on **polymorphism** to call the correct method

- ```
DataReader dr = new CSVDataReader()
dr.readData() // reads data from CSV
```
- ```
DataReader dr = new XMLDataReader()  
dr.readData() // reads data from XML
```

File System Implementations

Virtual File Systems: Concept

```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4
5 public class ClientApplication {
6     public static void main(String[] args) throws IOException {
7         System.out.println("Choose CSV or XML?");
8         BufferedReader br
9             = new BufferedReader(new InputStreamReader(System.in));
10        String type = br.readLine();
11        br.close();
12
13        DataReader reader = null;
14        if (type.equals("CSV")) {
15            reader = new CSVDataReader();
16        } else if (type.equals("XML")) {
17            reader = new XMLDataReader();
18        }
19
20        if (reader != null)
21            System.out.println(reader.readData());
22    }
23 }
```

File System Implementations

Virtual File Systems: Concept

- In a similar way, Unix and Linux **unify different file systems** and present them as a single hierarchy and hides away / abstracts the implementation specific details for the user
- The VFS presents a unified interface to the “outside”
- File system specific code is dealt with in an **implementation layer** that is **clearly separated from the interface**

File System Implementations

Virtual File Systems: Concept

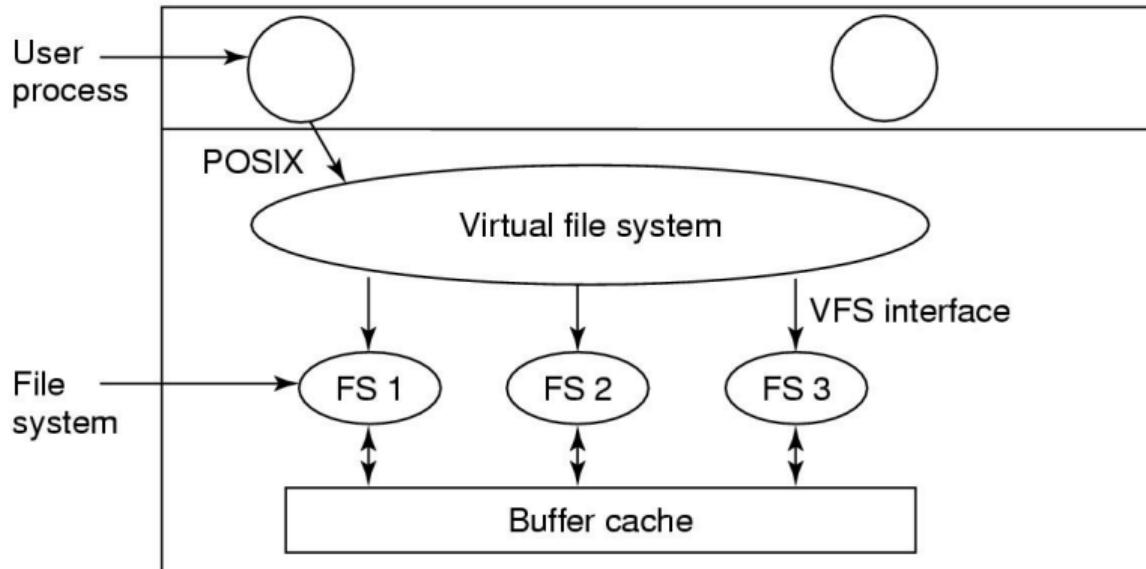


Figure: Virtual File System (Tanenbaum)

File System Implementations

Virtual File Systems: Concept

- The VFS interface commonly contains the **POSIX system calls** (open, close, read, write, ...)
- Each file system that meets the VFS requirements **provides an implementation** for the system calls contained in the interface
- Note that implementations can be for **remote file systems** (e.g. sshfs), i.e. the file can be stored on a different machine

File System Implementations

Virtual File Systems: In practice

- Every file system, including the root file system, is **registered with the VFS**
 - A list / table of addresses to the **VFS function calls** (i.e. function pointers) for the specific file system is provided
 - **Every VFS function call** corresponds to a specific **entry in the VFS function table** for the given file system
 - The **VFS maps / translates** the POSIX call onto the “native file system call”
- A virtual file system is essentially **good programming practice**

File System Implementations

Virtual File Systems: Example

```
$ df -Th
```

Filesystem	Type	Size	Used	Avail	Use%	Mounted on
devtmpfs	devtmpfs	32G	0	32G	0%	/dev
/dev/sda3	btrfs	215G	82G	132G	39%	/
tmpfs	tmpfs	32G	8.6M	32G	1%	/tmp
/dev/sda3	btrfs	215G	82G	132G	39%	/home
/dev/sda1	ext4	477M	166M	282M	38%	/boot
/dev/sdb	ext4	917G	165G	706G	19%	/data
pszit@bann:	fuse.sshfs	700M	282M	418M	41%	/mnt/bann

Summary

Take-Home Message

- **File system paradigms:**
 - Logs: store everything as close as possible
 - Journaling: apply the transaction principle
- VFS: apply good software design (polymorphism)

Test Your Understanding

File Systems

Exercises

Following up the previous question on FAT-32, we mentioned that FAT-32 has severe limitations (e.g. the file allocation table could be too big)

- Why do you think that this file system is still in use in most of flash drives, cameras or MP3 players?
- When would you consider formatting your flash drive with NTFS or ext3?
- If you format your flash drive with (Windows) NTFS, will it work (directly) in Unix systems?

Operating Systems and Concurrency

Lecture 23: File Systems V

COMP2007

Alexander Turner and Geert De Maere

{Alexander.Turner, Geert.DeMaere}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2022

Recap

- File systems implementations
 - Contiguous
 - Linked lists
 - FAT
 - i-nodes
- File systems paradigms (on top of implementations).
 - Log-structured file systems (improves efficiency)
 - Journaling (improves resiliency, robustness)
 - Virtual File Systems (improves flexibility, integration)

Goals for Today

Overview

- File system recovery
 - Scandisk
 - FSCK
- Defragmenting Disks
- File systems in Linux

File System Consistency

Checking Consistency

- Journaling heavily reduces the probability of having inconsistencies in a file system. In case of crash, the log stores what operations were not run.
- However, it can still be possible to get some inconsistencies (e.g. data blocks weren't flushed to the drive, typical case on USB drives!).
- This can be problematic, in particular for **structural blocks** such as i-nodes, directories, and free lists
- **System utilities** are available to restore file systems, e.g.:
 - Scandisk
 - FSCK
- There are two main consistency checks: block and directory.

File System Consistency

Checking Block Consistency

- Block consistency checks whether blocks are **assigned/used** the correct way
- Block consistency is checked by building **two tables**:
 - Table one counts how often a **block is present in a file** (based on the i-nodes)
 - Table two counts how often a **block is present in the free list**
- A consistent file system has a 1 in either of the tables for each block
- Typically, this is a **very slow process**, taking even hours (and running with the partition unmounted)

File System Consistency

Checking Block Consistency

Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15		
1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0	Blocks in use	
0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1		Free blocks
	(a)	(b)
	(c)	(d)

Figure 5–18. File system states. (a) Consistent.

Figure: Consistency checks (from Tanenbaum)

File System Consistency

Checking Block Consistency

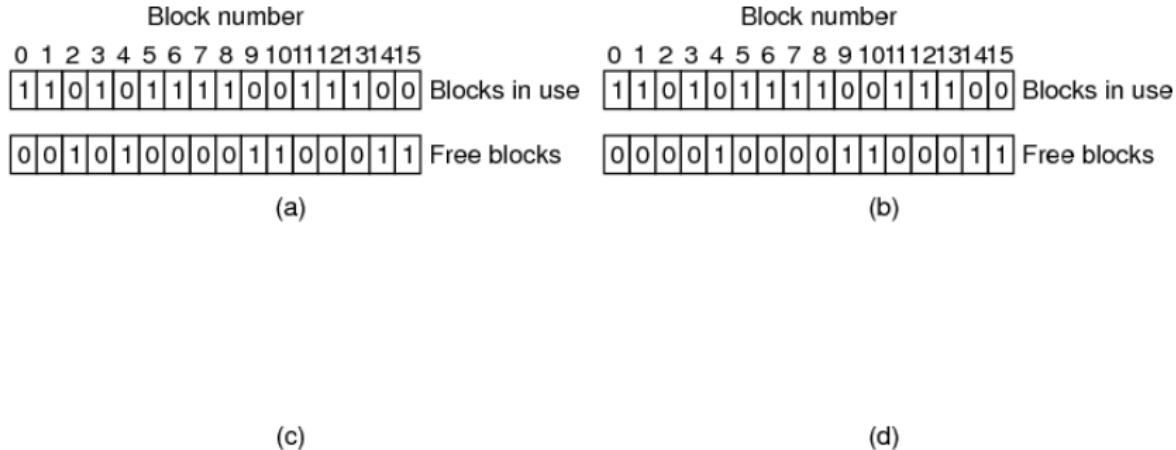


Figure 5–18. File system states. (a) Consistent. (b) Missing block.

Figure: Consistency checks (from Tanenbaum)

File System Consistency

Checking Block Consistency

Block number	Blocks in use	Block number	Blocks in use
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0]	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0]
	Free blocks		Free blocks
(a)		(b)	
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0]	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 1]
	Blocks in use		Blocks in use
	Free blocks		Free blocks
(c)		(d)	

Figure 5–18. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list.

Figure: Consistency checks (from Tanenbaum)

File System Consistency

Checking Block Consistency

Block number	Blocks in use	Block number	Blocks in use
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0]	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0]
0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1	Free blocks	0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 1	Free blocks
	(a)		(b)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0]	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[1 1 0 1 0 2 1 1 1 0 0 1 1 1 0 0]
0 0 1 0 2 0 0 0 0 1 1 0 0 0 1 1	Free blocks	0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1	Free blocks
	(c)		(d)

Figure 5–18. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data

Figure: Consistency checks (from Tanenbaum)

File System Consistency

Restoring Block Consistency

- A **missing block**: it does not exist in any of the tables ⇒ add it to the free list
- A block is **double counted** in the free list (“disaster” waiting to happen)
⇒ re-build the free list
- A block is present in **two or more files**
 - Removing one file results in the adding the block to the free list
 - Removing both files will result in a double entry in the free list
 - Solution: use new free block and copy the content (the file is still likely to be damaged)

File System Consistency

Restoring Block Consistency

FSCK Algorithm:

1. Iterate through all the i-nodes
 - retrieve the blocks
 - increment the counters
2. Iterate through the free list
 - increment counters for free blocks

File System Consistency

Restoring I-node Consistency

- Checking the directory system: are the **i-node counts correct?**
- Where can it go wrong?:
 - **I-node counter is higher** than the number of directories containing the file
 - Removing the file will reduce the i-node counter by 1
 - Since the counter will remain larger than 1, the i-node / disk space will not be released for future use
 - **I-node counter is less** than the number of directories containing the file
 - Removing the file will (eventually) set the i-node counter to 0 whilst the file is still referenced
 - The file / i-node will be released, even though the file was still in use

File System Consistency

Restoring I-node Consistency

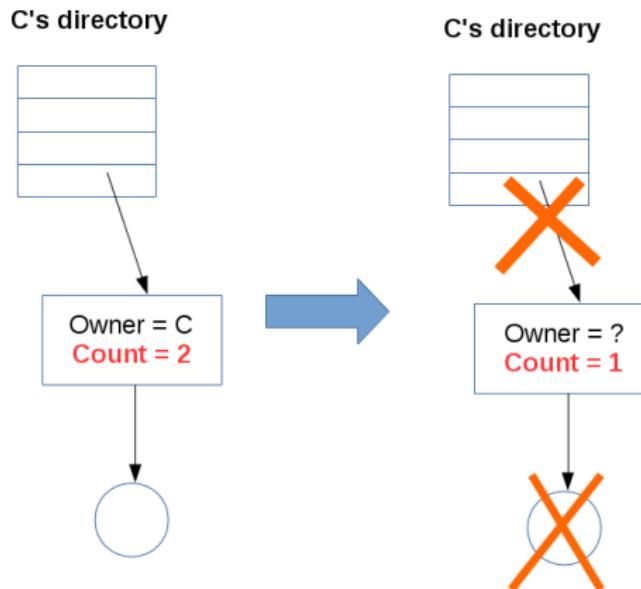


Figure: I-node counter is higher than the actual number of directories containing the file. Removing the file results in wasted memory.

File System Consistency

Restoring I-node Consistency

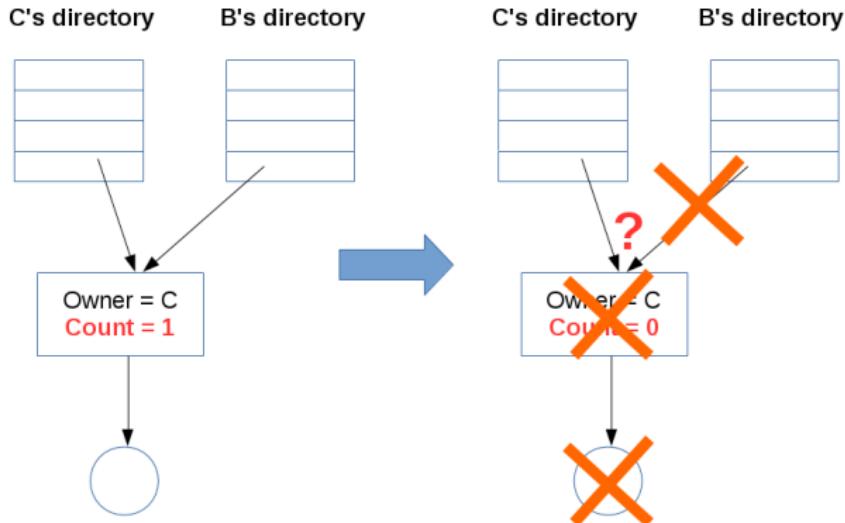


Figure: I-node counter is less than the actual number of directories containing the file.
Removing the file results in a missing file.

File System Consistency

Restoring I-node Consistency

- Recurse through the directory hierarchy
 - Check file specific counters
 - I.e. each file is associated with one counter
- One file may appear in multiple directories
 - Compare the file counters and i-node counters
 - Correct if necessary

File System Defragmentation

Compacting

- At the beginning, all free disk space is in a single contiguous unit.
- After a while, creating and removing files, a disk may end up badly fragmented (holes and file all over the place).
- **Defrag** utilities make file blocks contiguous (**very slow operation**), and free space in one or more **large contiguous regions** on the disk.
- Windows users should run this regularly, except on SSDs.
- **Linux (ext2/3) suffers less from fragmentation.**
- Defragmentating SSD is counter-productive (No gain in performance and SSDs wear out).

File System

History of the Linux file system

- **Minix file system:** the maximum file size was 64MB and file names were limited to 14 characters
- The “**extended file system**” (extfs): file names were 255 characters and the maximum file size was 2 GB
- The “**ext2**” file system: larger files, larger file names, better performance
- The “**ext3-4**” file system: journaling etc.

File System

The Extended 2 File System

- The second extended file system (**ext2**) is one of the most popular file systems in Linux.
- The main goals:
 - Improve the **performance** of MINIX and extfs file systems, distributing directories evenly over the disk.
 - Allow **greater file names and sizes**, improving directory implementation.

File System

Standard Unix file system vs. Extended 2 File System



Figure: Standard Unix Partition

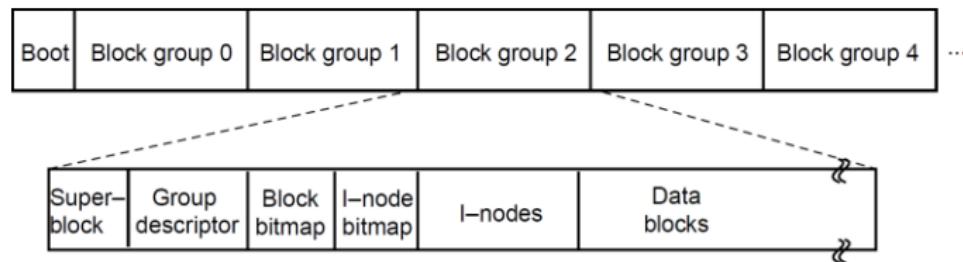


Figure: Ext2 Partition Layout (Tanenbaum)

File System

Directory Entries

- The **superblock** contains file system information (e.g. the number of i-nodes, disk blocks)
- The **group descriptor** contains bitmap locations, the number of free blocks, i-nodes and directories
- A **data block bitmap** and **i-node bitmap**, used to keep track of free disk blocks and i-nodes (Unix uses lists)
- A **table of i-nodes** containing file and disk block information
- **Data blocks** containing file and directory blocks

File System

The Extended 2 File System

- An ext2 partition is split into several **block groups** to:
 - **Reduce fragmentation** by storing i-nodes and files, and parent directories and files in the same block group if possible
 - Reduce **seek times** and improve performance
- All block groups have the same size and are stored sequentially (which allows direct indexing)

Exercises

- **Exercise 1:** Using the ext2 file system (i.e. 12 direct block addresses are contained in the i-node, and up to triple indirect), and assuming a block size of 4 kilobytes, and a 32-bits disk address space.
 - Could we store a file of 18 gigabytes?
 - How many disk blocks we spend for the i-node of a file of 16 megabytes?
- **Exercise 2:** In Linux, how many lookups are necessary to find (and load) the file: /opt/spark/bin/spark-shell?

Exercises

Could we store a file of 18 gigabytes?

- We have blocks of 4 kilobytes, and we need 32 bits (4 bytes) to represent a disk address ⇒ in one single block, we could store up to 1024 block pointers:
$$4\text{KB} = 4 \cdot 2^{10} / 4 \text{ bytes each} = 2^{10} = 1024 \text{ block pointers.}$$
- Using the 12 direct block pointers, we could have a file of with 12 blocks.
- Using the single indirect: 1024 extra block pointers.
- Using the double indirect: $1024 \cdot 1024$ block pointers (1048576).
- Using the triple indirect, $1024 \cdot 1024 \cdot 1024$ block pointers (1073741824).
If we aggregate all of them => 1,074,791,436 blocks of 4KBs which is approx 4TB.

Exercises

How many disk blocks we spend for the i-node of a file of 16 megabytes?

- For a file of 16 megabytes, we need: $16 \cdot 2^{20} / 4 \cdot 2^{10} = 2^{12} = 4096$ blocks pointers!
- We will use fully all direct block pointers (12) [**1 block**]
- With the single indirect we have 1024 extra block pointers [**1 block**]
- With the double indirect we can address 1048576 block pointer... so we won't go further than this level.
- With direct pointer and single indirect we have covered $1024 + 12$ block pointers... we still need 3060
- Each block will handle 1024 pointers. So, $3060 / 1024 = 2.9882$
- We need three blocks + the "first level" block.
- Total: 6 blocks of 4 kilobytes => 24KB

Exercises

How many disk blocks we spend for the i-node of a file of 16 megabytes? Differently written

- The 12 block pointers in the Inode = one disk block (for the inode really)
[1 block]
- A single indirect block pointer points to another disk block which holds 1024 block pointers (which is still not enough - we need 4096 block pointers which is 3060 more than we have : $4096 - (12 + 1024)$).
[1 block]
- So we use a double indirect pointer (to get another 3060 block pointers **[1 block]** which points to another disk block, which points to 1024 other disk blocks (each containing 1024 block pointers). We only need 3 **[3 blocks]** of these disk blocks to get out 3060 and take us over the 4096 block pointers needed to store the 32mb file.
[3 blocks]

Summary

Take-Home Message

- File system consistency
- Linux file systems