Student's Handbook Guide to CMPUT 379

Operating System Concepts

A student's guide to the complex nature of Operating Systems

Learn about the definition of a process; process states and state transitions; process control block; operations on processes; interrupt processing; parallel processing; resource allocation; shared and unshared allocation; critical sections; semaphores; deadlock; deadlock prevention, avoidance, detection, and recovery; memory management; memory allocation schemes; virtual memory; paging and segmentation; page replacement strategies; working sets; demand paging; job and processor scheduling; scheduling levels, objectives, and criteria; various scheduling algorithms; multi-processor considerations; file system functions; file organization; tree structured file systems; space allocation; file catalogs; file access control mechanisms; operating systems security.

By mastering the complex sub-field of Operating Systems will lead you to a heroic victory against the machination of the professor. By slaying his programming minions, defeating his midterm champion and ending the final themselves. The heroic legacy of your achievement will be remembered on your academic transcript.

Disclaimer: With the sole exception of the cover this book will crush your hopes and dreams as it does not contain prehistoric or high fantasy content like elves, dwarfs, dragons, or aarakocras.

Contents

1: Introduction and Overview

An operating system is a bunch of software layers separating your programs from the computer hardware.

Abstraction

The idea that some details are removed from the user. It removes details that the user does not need to worry about. This however can lead to performance loss as the user loses some of the control over the hardware.

Example

> Files are an abstraction of information that are stored on persistence storage, i.e. non-volatile.

To maintain or improve performance, the operating system needs to control the hardware optimally. The operating system is responsible to make the user experience fast and seamless even with limiting hardware. Moreover, the user doesn't know this process is happening.

Compiling, Linking, Loading

Compiling

When compiling the source code the GCC compiler is converting the human-readable code into a **object file**. This object file or **modules** is a tentative executable, it is missing certain things like functions from external files, etc. To represent these missing information, the object file contains symbolic information in the form of a symbol table. It also contains procedures to fix the object file into a fully functioning executable.
The object file is in a specific format called ELF or Executable and Linking Format.

Linking

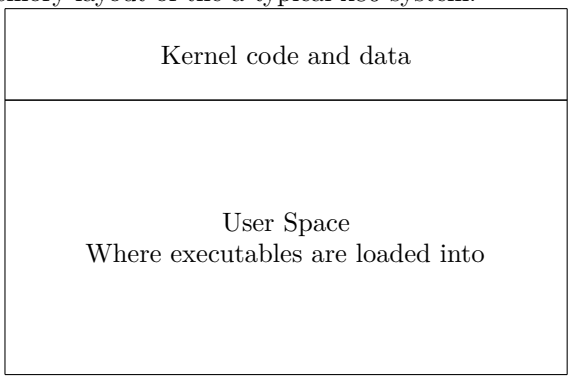
Linking takes the objects file(s), and using the information from the object file(s) fix and link the files together to create a fully functioning executable that can be loaded into memory.

Loading

Loading takes the executable and loads the information into memory. The actual location of the executable is flexible but not arbitrary. Each platform/OS use specific conventions and format where an executable will go. The loader uses information from the previous steps to indicate where and how to place (or initialize) the various memory areas of the process-to-be. Some libraries are not loaded until needed by the program, these are dynamically loaded libraries.

Intro to x86 Memory Layout

In the figure below shows the memory layout of the a typical x86 system.



The Kernel space is protected and have privileged access to certain services while the user space does not have privileged access or is un-privileged.
To access the kernel code and use the kernel's services a system call is called by the program where it runs a subroutine in the kernel space protected from the user. A system call is not the same as a function call as a system call calls a subroutine in the operating system while a function call will call a subroutine in the program. Moreover, a system call will interrupt the current process for the system call process making the program a bit slower then its program's equivalent.

Basic Storage classes in C

Here is a sample code describing the storage classes in C

```
int read_something (void);
int do_something (int);
void write_something (const char*);

int some_global_variable;
static int some_local_variable;

main () {
    int some_stack_variable;
    some_stack_variable = read_something ();
    some_global_variable = do_something (some_stack_variable);
    write_something ("I am done");
}
```

```
int some_global_variable;
```

This variable is globally visible by all modules (common, bss)

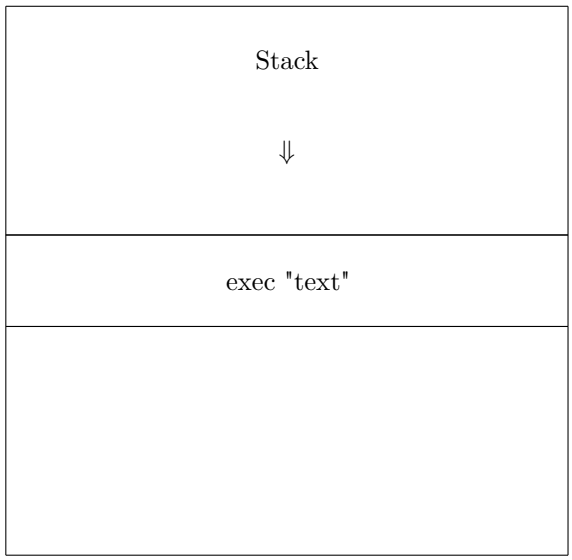
```
static int some_local_variable;
```

This variable is globally visible by all function within the current module, but not outside the module (data)

```
int some_stack_variable;
```

This variable is allocated on stack, visible only within the block (called auto for automatic)

Memory layout in the User Space



The stack grows downward for this system.

2: ELF: Executable and Linking Format

Introduction

This describes the format for object files compiled in GCC.

Relationship with other Elves

Note: These ELFs are not related to the Dungeons and Dragon's variety of elf, nor are they related to Game Workshops' varieties of Aeldari from Warhammer 40k, or the Asur, Druchii, or Asrai from Warhammer Fantasy.

DWARF

Is a common debugger format used to debug ELF.

Like the relationship status of the ELF these DWARFs are not related to the Dungeon and Dragon's variety of dwarf, nor are they relate to Game Workshops' varieties of Squats from Warhammer 40k, or the many dwarf species from Warhammer Fantasy.

The three main types of an object file

1. **Relocatable file**: holds code and data suitable for linking with other object files to create an executable or a shared object file.
2. **Executable file**: holds a program suitable for execution.
3. **Shared object file**: holds code and data suitable for linking in two contexts. First, the link editor may process it with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

Created by the assembler and link editor, object files are binary representations of programs intended to execute directly on a processor. Programs that require other abstract machines are excluded.

File Format

Because this format is used in the linking (building a program) and execution (running a program). The object file format provides parallel views of a file's contents, reflecting the different needs of these activities.

(a) Linking View

ELF Header
Program Header Table *Optional*
Section 1
...
Section n
...
...
Section Header Table

(b) Execution View

ELF Header
Program Header Table
Segment 1
Segment 2
...
Section Header Table *Optional*

EFL Header

An *ELF header* resides at the beginning and holds a "road map" describing the file's organization, it also contains the number of sections in the file. Sections hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, etc.

Program Header Table

A *Program header table*, if present, tells the system how to create a process image (execute a program). Files used to build a process image must have a program header table; relocatable files do not need one.

Section Header Table

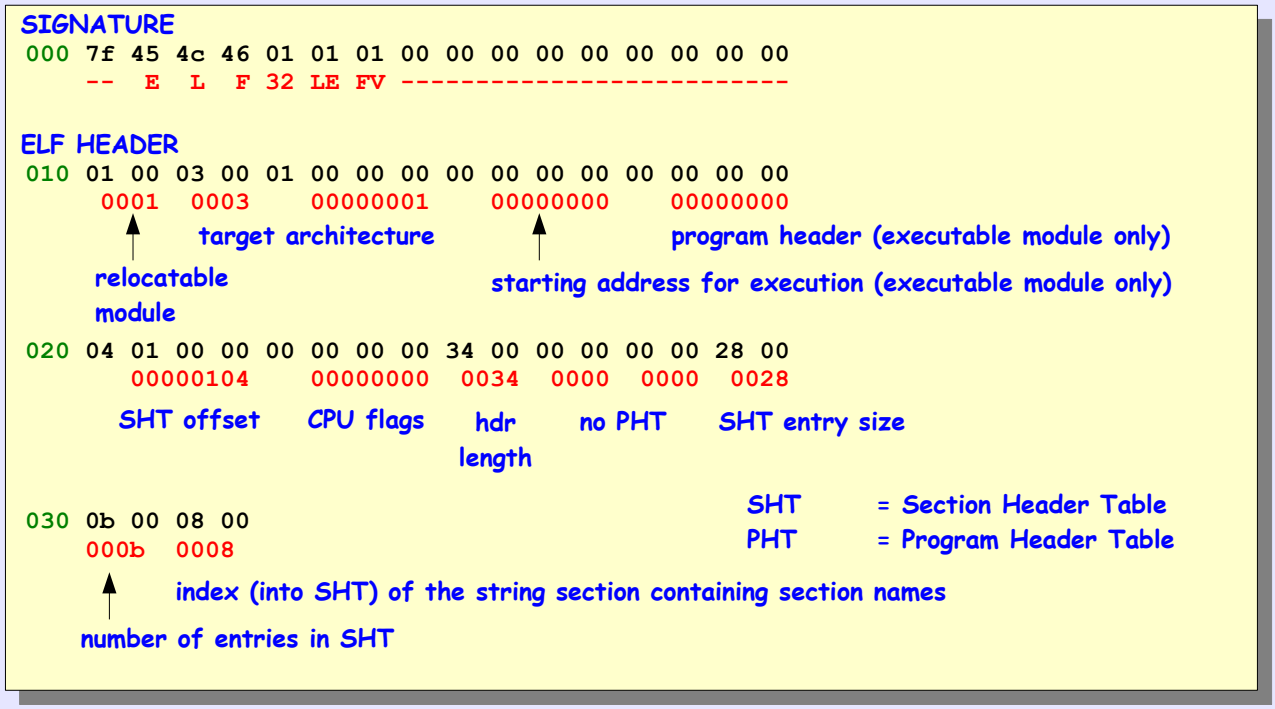
A *section header table* contains information describing the file's sections. Every section has an entry in the table, each entry gives information such as the section name, location, size, and so on. Files used during linking must have a section header table, other object files may or may not have one.

Example of an ELF

Here is an example of an ELF format object file for this source code below:

```
1    int read_something (void);
2    int do_something (int);
3    void write_something (const char*);
4
5    int some_global_variable;
6    static int some_local_variable;
7
8    main () {
9        int some_stack_variable;
10       some_stack_variable = read_something ();
11       some_global_variable = do_something (some_stack_variable);
12       write_something ("I am done");
13   }
```

ELF format



What does these mean?

- **Signature**: Tells the linker that this is a ELF formatted object file and defines whether to use 32 or 64 bit addresses. In this case it uses a 32-bit address.
- **SHT offset**: Is the relative offset from the start of the file to the start of the Section Header Table.
- **SHT entry size**: Tells the size of each entry in the Section Header Table. To deal with variable length sections, each entry will have a pointer to the section as well as it size. There are other information as well but that will be described later.
- **# of entries in SHT**: Tells the number of entries in the Section Header Table.
- **index (into SHT) of the string section containing section names**: This is the index of the section in the Section Header Table that contains the location, size, and other information regarding the string section containing section names. This section is called **.shstrtab** or Section Header STRing TABle.

ELF contents: SHT (0-1)

```
Section 0 (NULL)
104             00 00 00 00 00 00 00 00 00 00 00 00
110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
120 00 00 00 00 00 00 00 00 00 00 00 00

                                                    Entry size = 28; 104 + 28 = 12c
                                                    28 bytes = a words

Section 1 (.text)
12c 1f 00 00 00 = 0000001f   name pointer (index into a string section)
    01 00 00 00 = 00000001   section type (PROGBITS = belongs to the program)
    06 00 00 00 = 00000006   flags: in program's memory + executable
    00 00 00 00 = 00000000   address in memory (unknown because module is relocatable)
    34 00 00 00 = 00000034   offset to section in ELF file
    38 00 00 00 = 00000038   size 34 + 38 = 6c
    00 00 00 00 = 00000000   extra link (none - typically points to a related section)
    00 00 00 00 = 00000000   extra info (none)
    04 00 00 00 = 00000004   address alignment (word boundary)
    00 00 00 00 = 00000000   entry size (if the section consists of fixed-size entries)
```


What does these mean?

- **name pointer**: Also called sh_name, is an offset to a null-terminated character sequence in the section name (.shstrtab) section that provides the name for this section.
- **section type**: Also called sh_type, describes the type of the section, in this case 0x01 means SH_PROGBITS or Program data.
- **flags**: Also called sh_flags, identities the attributes of the section, in this case 06 means in program's memory and executable.
- **address in memory**: Also called sh_addr, is the virtual address of the section in memory, for sections that are loaded. In this case it is unknown (NULL) because the module is relocatable.
- **offset to section in ELF file**: Also called sh_offset, is the offset of the section in the ELF file. The offset is relative from the head of the ELF file.
- **size**: Also called sh_size, is the size in bytes of the section in the ELF file. May be 0 due to compiler optimization, and other things.
- **extra link**: Also called sh_link, contains the section index of an associated section. This field is used for several purposes, depending on the type of the section.
- **extra info**: Also called sh_info, contains extra information about the section. This field is used for several purposes, depending on the type of the section.
- **address alignment**: Also called sh_addralign, contains the required alignment of the section. This field must be a power of 2.
- **entry size**: Also called sh_entsize, contains the size, in bytes, of each entry, for sections that contain fixed-sized entries. Otherwise, this field is zero.

ELF contents: section body 8, 5, 1

Section 8 (STRTAB .shstrtab)

```
0b1     00 2e 73 79 6d 74 61 62 00 2e 73 74 72 74 61
        --  .  s  y  m  t  a  b --  .  s  t  r  t  a
0c0 62 00 2e 73 68 73 74 72 74 61 62 00 2e 72 65 6c
     b --  .  s  h  s  t  r  t  a  b --  .  r  e  l
0d0 2e 74 65 78 74 00 2e 64 61 74 61 00 2e 62 73 73
     .  t  e  x  t --  .  d  a  t  a --  .  b  s  s
0e0 00 2e 72 6f 64 61 74 61 00 2e 6e 6f 74 65 2e 47
    --  .  r  o  d  a  t  a --  .  n  o  t  e  .  G
0f0 4e 55 2d 73 74 61 63 6b 00 2e 63 6f 6d 6d 65 6e
     N  U  -  s  t  a  c  k --  .  c  o  m  m  e  n
100 74 00
     t --
```

Section 5 (PROGBITS .rodata)

```
074             49 20 61 6d 20 64 6f 6e 65 00
                 I     a  m     d  o  n  e --
```

Section 1 (PROGBITS .text)

```
034             55 89 e5 83 ec 08 e8 fc ff ff ff 89
040 c0 89 45 fc 83 ec 0c ff 75 fc e8 fc ff ff ff 83
050 c4 10 89 c0 a3 00 00 00 00 83 ec 0c 68 00 00 00
060 00 e8 fc ff ff ff 83 c4 10 c9 c3 90
```

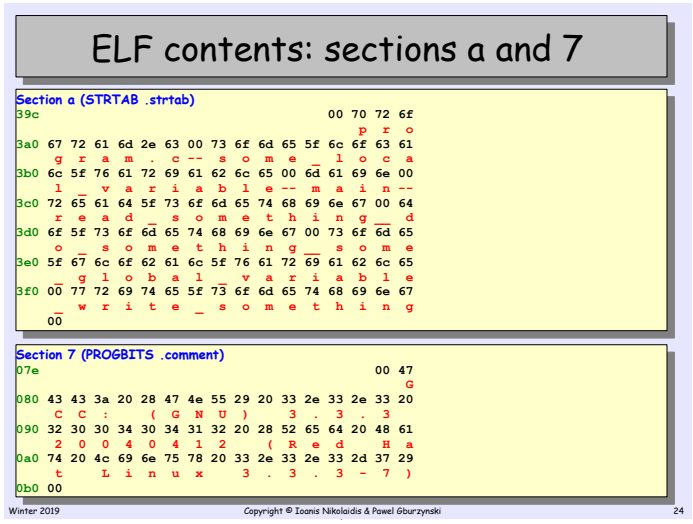
Winter 2019 23

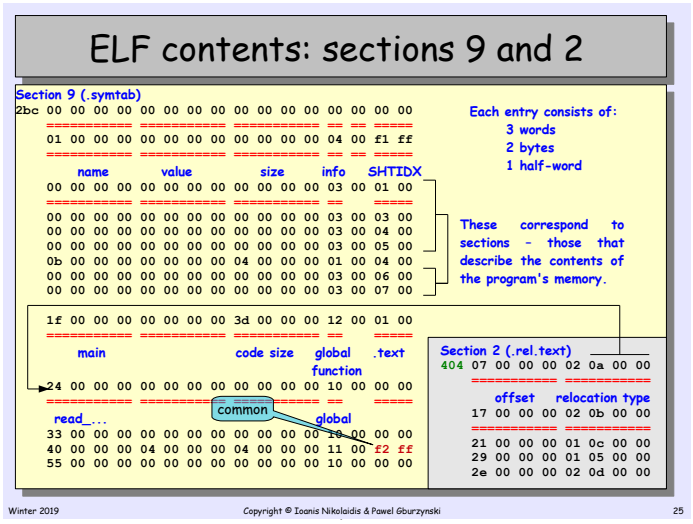
- As you can see being 0x1f from Section 8 points to the beginning of .text\0. The section name for Section 1.
- In section 5, it's only "I am done\0", .rodata means read-only data as the string that is being used is in a function where its only parameter takes in a constant char array. Therefore the string is immutable.
- Section 1 contains the code segment or the executable code for the program.
- Section 8 is a string table (SH_STRTAB) that contains the name for each section.

Data Segment

Components in the data segments

Text	Is where a portion of an object file or the corresponding section of the program's virtual address space contains the executable instructions is stored and is generally read-only and fixed in size.
Data	contains any global or static variables which have a pre-defined or initialized value and can be modified.
BSS	Uninitialized data.
Heap	Heap space.
Stack	Call stack.





- Section a is a string table that contains null-terminated character sequences that contain the names for the symbol table.
- Section 7 is part of the program and contains comment about the compiler used. - Section 9 is the symbol table for the EFL file.
- Section 2 is .rel.text or relocatable .text

Symbol Table

Each entry in the symbol table is made up with 3 words (32-bits in this case), 2 bytes, and 1 half-word (16-bits in this case). These represent the name, value, size, info and the Section Header Table index respectively.
- name is an offset from the head of the .strtab section that provides the name for each entry, NULL values are possible.
- value is the value for that symbol if its a variable.
- size is the size in bytes for that symbol if its a piece of code.
- info gives information about the symbol, is it a global function, global variable, etc.
- SHTIDX is the index in the section header table to the section that the symbol is related to.

An Example

The symbol for main is called main as the offset of 0x1f in the .strtab is .main, it has no value as its not a variable, it has a size of 61 byte, its a global function, and its related to the .text section.

```
0: 55                   push   %ebp
1: 89 e5                mov    %esp,%ebp
3: 83 ec 08             sub    $0x8,%esp
6: e8 fc ff ff ff       call   7 <main+0x7>
b: 89 c0                mov    %eax,%eax
d: 89 45 fc             mov    %eax,0xfffffffc(%ebp)
10: 83 ec 0c            sub    $0xc,%esp
13: ff 75 fc            pushl  0xfffffffc(%ebp)
16: e8 fc ff ff ff      call   17 <main+0x17>
1b: 83 c4 10            add    $0x10,%esp
1e: 89 c0               mov    %eax,%eax
20: a3 00 00 00 00      mov    %eax,0x0
25: 83 ec 0c            sub    $0xc,%esp
28: 68 00 00 00 00      push   $0x0
2d: e8 fc ff ff ff      call   2e <main+0x2e>
32: 83 c4 10            add    $0x10,%esp
35: c9                  leave
36: c3                  ret
```

This is the .text section disassembled back into x86 assembly. An address doesn't have to be located outside the current module to be essentially unknown. The exact location of the module in memory is only known when the **complete** program is loaded for execution. The benefits of DLL are many:

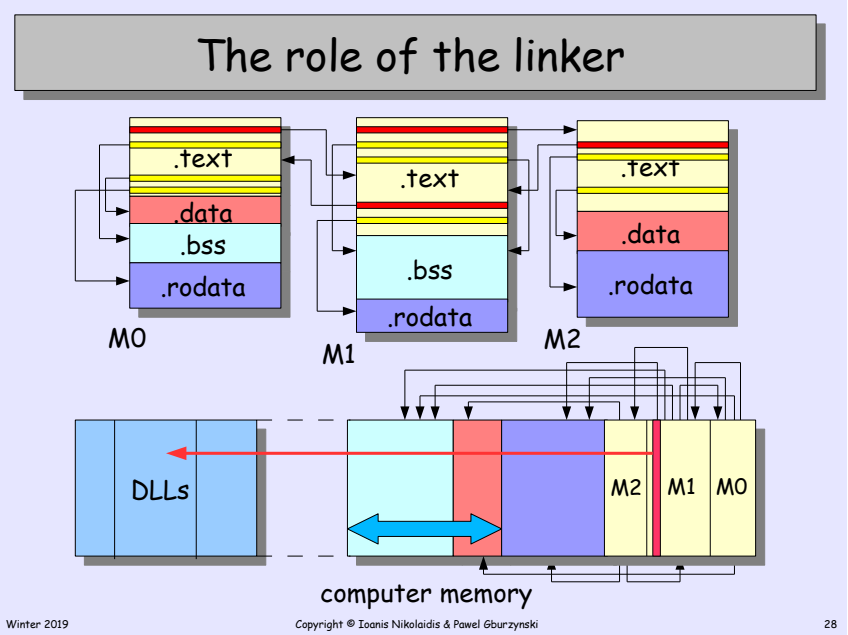


Figure 2: The role of the linker is it combine the modules and consolidate each data segment from the modules together, so for example all .data sections from the modules that have one will be grouped together and have the pointers in the sections updated to those new locations.

The Benefits of DLL

- **Speed of Loading/executable size**: Not everything needs to be loaded during execution.
- **Incremental Loading (as needed)**: Things are loaded during run time, therefore increasing the speed of loading
- **Debugging**: You don't have to recompile everything, just the modules that are being tested/debugged.

3: DLL

Terms

- **Absolute**: it can only be executed when loaded in a specific memory location.
- **Relocatable**: some relocatable information is left in the executable, such that when the program is loaded, it can still be relocated to the actual location.
- **Position-independent**: the program executes correctly, no matter where it is loaded.
- **Movable**: the program can be loaded anywhere, it can also be moved to a new location in the middle of its execution without impairing its integrity.

PIC: Position Independent Code

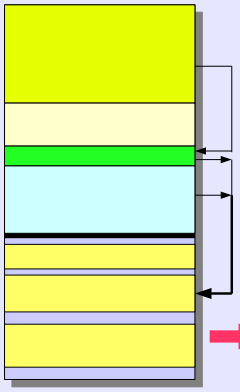
Most contemporary CPU architectures implement PC-relative addressing modes, which typically look like this:

operand = Mem (PC + offset)

```
...
addr:
...
loop:   load   r1, addr
        load   r2, r4
        call   funct
...
        jump   loop
...
       .data   addr
...
```

PIC has many uses in various scenarios where it is convenient to be able to load the code in any location without having to worry about relocation (DLLs, for one thing).

Illustration



Note that the dynamic code can operate on non-PIC data supplied by the static caller, e.g., via function parameters.

It may be convenient to be able to put those pieces wherever there is room available

The executable

```
SIGNATURE (same as before)
000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
    --  E  L  F 32 LE FV ------------------------

ELF HEADER
010 02 00 03 00 01 00 00 00 90 83 04 08 34 00 00 00
     0002  0003     00000001     08048390     00000034
       ↑ target architecture          ↑ program header pointer
   executable                  starting address for execution
   module
020 c8 2a 00 00 00 00 00 00 34 00 20 00 06 00 28 00
      00002ac8     00000000  0034  0020  0006  0028
    SHT offset   CPU flags   hdr   PHT   SHT entry size
                             length params
                                   size +
030 1f 00 1c 00                    #entries   SHT = Section Header Table
    001f  001c                               PHT = Program Header Table
      ↑ index (into SHT) of the string section containing section names
  number of entries in SHT
```


objdump

Once you know what to expect in an ELF file, there is a way to display its contents in a more friendly format – see man objdump. Also, you may want to try readelf.

objdump -h executable (list section headers)

```
executable:     file format elf32-i386
Sections:       (a bit confusing, numbering ignores the NULL section and is in decimal)
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  080480f4  080480f4  000000f4  2**0   /lib/ld-linux.so.2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
......
 11 .text         000001d0  08048390  08048390  00000390  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
......
 13 .rodata       0000001d  08048580  08048580  00000580  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .data         00000010  080495a0  080495a0  000005a0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
......
 21 .bss          00000020  080496b4  080496b4  000006b4  2**2
                  ALLOC
......
```


The essence

The executable includes many sections that at first sight appear exotic. The meaning of some of them may become clearer later in the course.

The obvious ones include:

.text	the program code
.rodata	constants, mostly character strings
.data	dynamic global initialized data
.bss	dynamic global uninitialized data

Some "exotic" sections relate to the idea of dynamic linking. More generally, a program may need specific DLLs and/or require assistance of "interpreters" to create its memory image, as its fragments become dynamically needed.

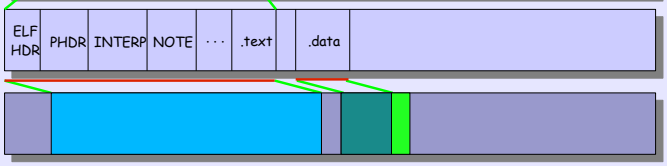
The .text section

```
Section b (.text)     = 2ac8 + c * 28
2ca8   80 00 00 00 = 00000080    name pointer
       01 00 00 00 = 00000001    section type (PROGBITS)
       06 00 00 00 = 00000006    flags: alloc + exec
       90 83 04 08 = 08048390    address in memory
       90 03 00 00 = 00000390    offset in ELF file
       d0 01 00 00 = 000001d0    size
       00 00 00 00 = 00000000    extra link (none)
       00 00 00 00 = 00000000    extra info (none)
       01 00 00 00 = 00000001    address alignment (byte)
       00 00 00 00 = 00000000    entry size
```

Sections representing memory chunks, like .text, .data, .rodata, are simply pieces to be written into the respective locations in memory to build the program's executable image.
Some allocatable sections, like .bss, have no image in the file: size and location are the only attributes that matter.

Things are a bit more complicated:

```
Program Header: (this is what in fact describes how the program is loaded)

    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000c0 memsz 0x000000c0 flags r-x
  INTERP off    0x000000f4 vaddr 0x080480f4 paddr 0x080480f4 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0000059d memsz 0x0000059d flags r-x
    LOAD off    0x000005a0 vaddr 0x080495a0 paddr 0x080495a0 align 2**12
         filesz 0x00000114 memsz 0x00000134 flags rw-
 DYNAMIC off    0x000005ec vaddr 0x080495ec paddr 0x080495ec align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000108 vaddr 0x08048108 paddr 0x08048108 align 2**2
         filesz 0x00000020 memsz 0x00000020 flags r--
```



When the loader is called to load the executable into memory, it will load the correct files into the logical memory in the most appropriate way, for example DLLs may not be loaded if they are not being used. Each file that are being loaded can have read, write, and execute permissions set depending on it's content. Certain files have to be aligned to the memory's page size.

How do we interface with the OS?

```
int read_something (void) {
    int res;
    scanf ("%d", &res);
    return res;
}
```

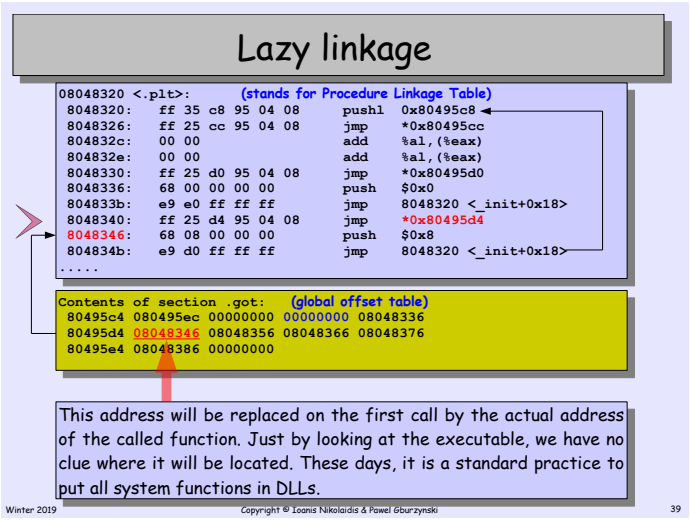
```
80484d0 <read_something>:
80484d0:    55                      push   %ebp
80484d1:    89 e5                   mov    %esp,%ebp
80484d3:    83 ec 08                sub    $0x8,%esp
80484d6:    83 ec 08                sub    $0x8,%esp
80484d9:    8d 45 fc                lea    0xfffffffc(%ebp),%eax
80484dc:    50                      push   %eax
80484dd:    68 92 85 04 08          push   $0x8048592
80484e2:    e8 59 fe ff ff          call   8048340 <_init+0x38>
80484e7:    83 c4 10                add    $0x10,%esp
80484ea:    8b 45 fc                mov    0xfffffffc(%ebp),%eax
80484ed:    c9                      leave
80484ee:    c3                      ret
```

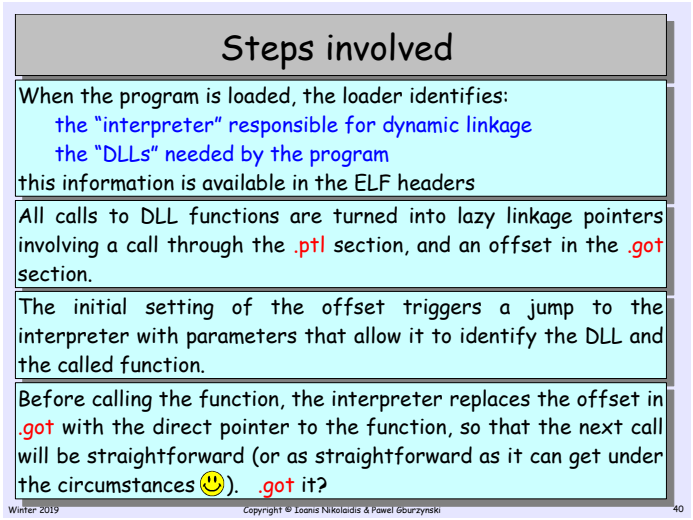
```
08048320 <.plt>:        (stands for Procedure Linkage Table)
 8048320:   ff 35 c8 95 04 08      pushl  0x80495c8
 8048326:   ff 25 cc 95 04 08      jmp    *0x80495cc
 804832c:   00 00                  add    %al,(%eax)
 804832e:   00 00                  add    %al,(%eax)
 8048330:   ff 25 d0 95 04 08      jmp    *0x80495d0
 8048336:   68 00 00 00 00         push   $0x0
 804833b:   e9 e0 ff ff ff         jmp    8048320 <_init+0x18>
 8048340:   ff 25 d4 95 04 08      jmp    *0x80495d4
 8048346:   68 08 00 00 00         push   $0x8
 804834b:   e9 d0 ff ff ff         jmp    8048320 <_init+0x18>
.....
```

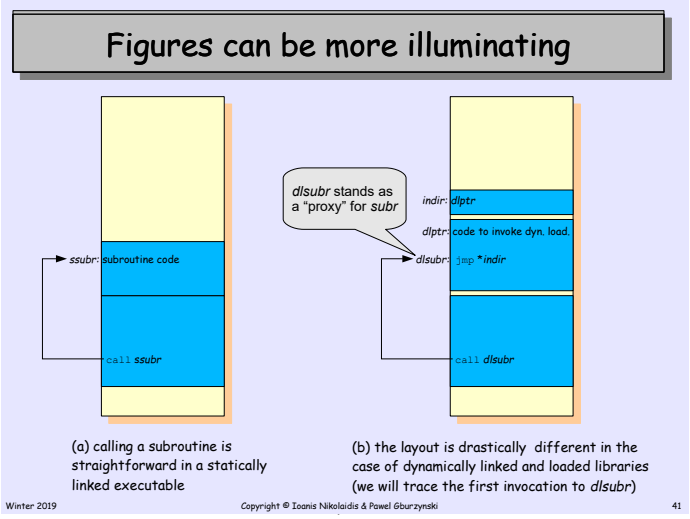

When scanf is called in the program, the PC does not directly go to the actual function. Because this function is at an unknown location (can be a DLL). The PC first goes to the procedure linkage table as a level of redirection where it then jumps to the entry point of scanf.

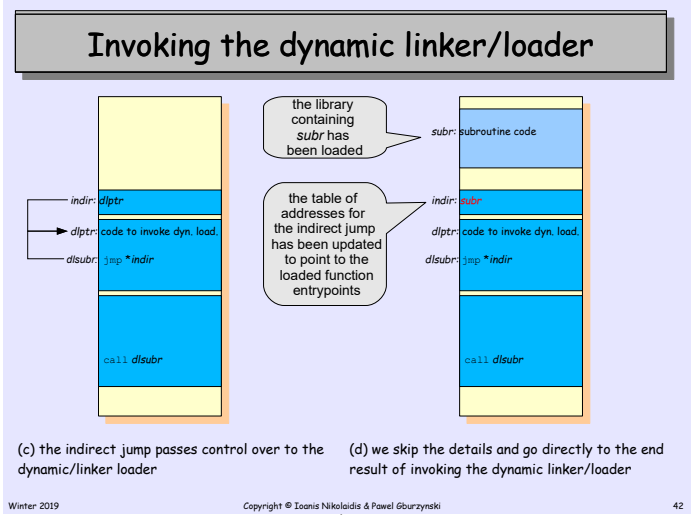
How are DDLs loaded

Libraries that are dynamically linked and loaded often have their own methods of being loaded into memory. One way is by lazy linkage.







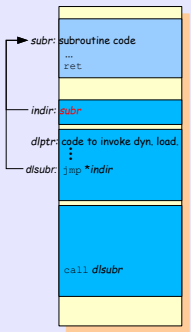


b) During the first function call of the procedure in the executable. The linker first adds a level of indirection to the procedure linkage table. The **Procedure Linkage Table or .plt** section contains code that will dynamically load the function into memory.

c) Because the new function can be placed really anywhere in memory the **Global Offset Table or .got** is used as another layer of indirection to point the jump call from before to the entry point of the function that will load the requested function into memory.
d) Once the function has been loaded into memory the .got is updated so that it points to the entry point of the function. Note: the return address has not been touch, it still points back to the main executable.

Following through the 1st invocation

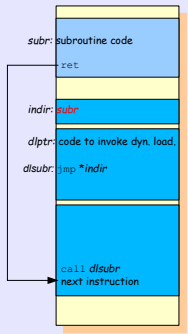
(e) for convenience let's assume that the execution after invoking the loader continues to the same indirect jump we had encountered earlier



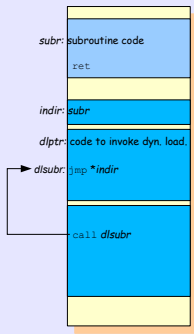
(f) upon (re-executing) the jump we branch to the address of the function that has been loaded

Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 43

The final touches



(g) this step looks obvious but it implies that the state of the stack when the `jmp` (figure (f)) was executed is as it was after the `call` (figure (b))

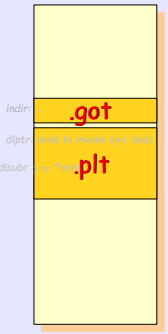


(h) the second and subsequent calls (from the same point or elsewhere in the code) to *subr* pay only the cost of the one additional indirect jump instruction

Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 44

e) Right after the code that loaded the function into memory is the same jump command that points to the same entry in the .got table, only this time it had been updated to point to the entry point of the function.
f) When we recall the same jump command instead of reloading the function again which is idiotic it redirect the PC to the entry point of the loaded function.
g-h) As this loading process only used pure jump commands the return address register was not effected. Therefore, calling return in the subroutine updates the PC back to the main executable. All subsequent calls only needs to pay one additional level of redirection to execute the subroutine.

The names for the two special areas



Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 45

Now for the easy way

```
gcc -o executable -static program.o extras.o
```

Executable size: dynamically linked = 14221 bytes
statically linked = 1691082 bytes

```
8048218 <read_something>:
8048218:   55                      push   %ebp
8048219:   89 e5                   mov    %esp,%ebp
804821b:   83 ec 08                sub    $0x8,%esp
804821e:   83 ec 08                sub    $0x8,%esp
8048221:   8d 45 fc                lea    0xfffffffc(%ebp),%eax
8048224:   50                      push   %eax
8048225:   68 f2 e4 08 08          push   $0x808e4f2
804822a:   e8 cd 04 00 00          call   80486fc <scanf>
804822f:   83 c4 10                add    $0x10,%esp
8048232:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
8048235:   c9                      leave
8048236:   c3                      ret
080486fc <scanf>:
80486fc:   55                      push   %ebp
.....
8048711:   e8 ba 34 01 00          call   805bbd0 <_IO_vfscanf>
.....
```

Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 46

You can statically linked all the libraries together to get a massive executable if you don't want to deal with this mess. It also increases portability (slightly) at the cost of size.

This is the function call for scanf. It looks complicated but the thing you need to know is the int $0x80 system call. The %eax identities which system call is invoked

```
080651c0 <__libc_read>:
80651c0:   53                      push   %ebx
80651c1:   8b 54 24 10             mov    0x10(%esp,1),%edx
80651c5:   8b 4c 24 0c             mov    0xc(%esp,1),%ecx
80651c9:   8b 5c 24 08             mov    0x8(%esp,1),%ebx
80651cd:   b8 03 00 00 00          mov    $0x3,%eax
80651d2:   cd 80                   int    $0x80
80651d4:   5b                      pop    %ebx
80651d5:   3d 01 f0 ff ff          cmp    $0xfffff001,%eax
80651da:   0f 83 70 da fe ff       jae    8052c50 <__syscall_error>
80651e0:   c3                      ret
```

When we want input from the user the function scanf uses a system call that will transfer control from the program or the user in user space to the kernel that will retrieve the information on your behalf. This is to improve security, and reliability as the kernel developer is responsible for this function. When the system call returns control is given back to the program.
Hardware can also call kernel functions by the means of an external software interrupt. This is usually handled by an interrupt handler.
When you create a new process, you fork the parent process so that the running process which can be completely different is a child of the parent process. The only exception to this idea is the first process which is created differently.

Blocking system calls

Because of the inability to proceed with the execution of the system call (see the last **read** call on the previous slide), the process that issued the system call needs to be (temporarily) blocked from continuing execution. Therefore we need:

- a mechanism to **switch** from the current process to another one that is not blocked (for productive and efficient use of the CPU)
- a mechanism to **unblock** the blocked processes when the reason (internal or external event) they were waiting for is satisfied (and to continue its execution in the CPU)
- genuinely **external events** are asynchronously delivered to the CPU via interrupts (rarely, via polling); **internal events** originate from the execution of other code (typically, other processes); inter-process communication (IPC) mechanisms provide some of those internal events

** The term "event" is overloaded. Here we use it informally.

4: Processes

A process and its children are formed into a tree.
In modern OSes Processes are running concurrently with a "seamless" experience.

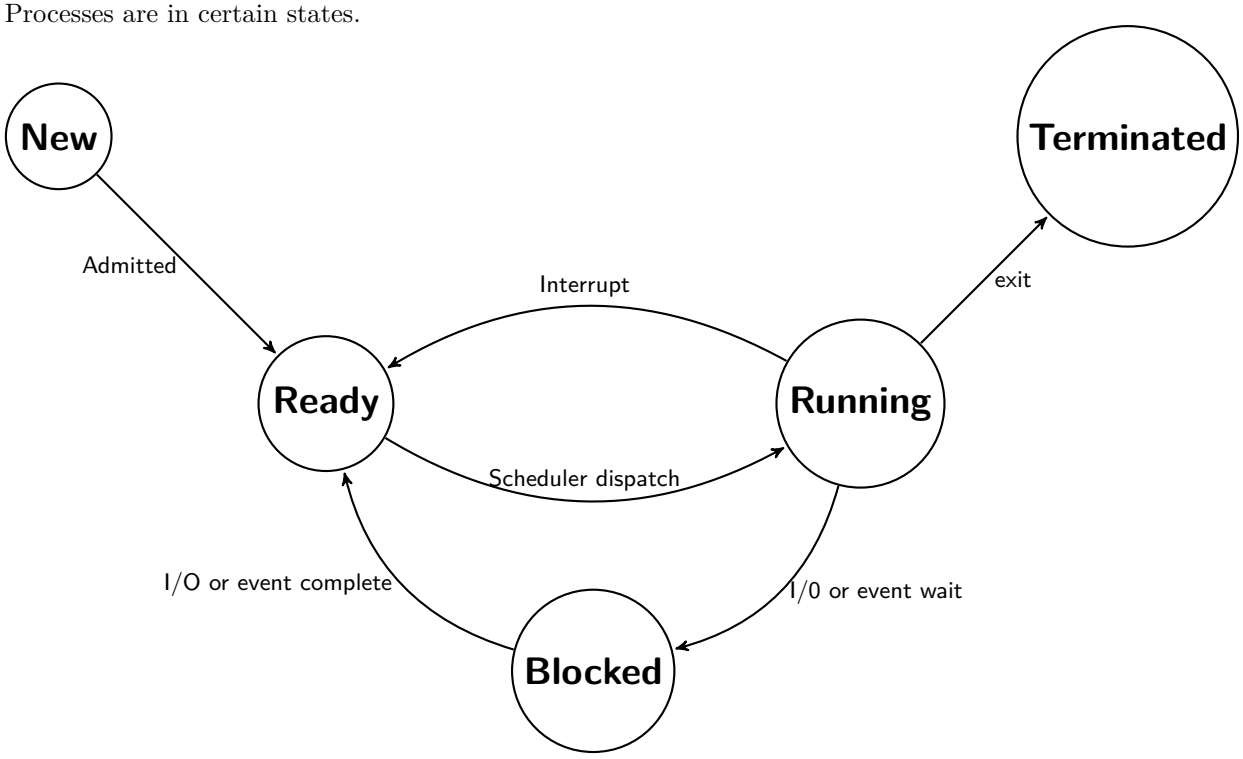
Definitions

Process: A program executing in some environment.

Program Executing: text, bss, heap, and stack segments in memory.
Context: Shell environment variables, user privilege access, etc.

Conceptual Framework

Processes are in certain states.



When a new process is created the OS will try to allocate the resources to that process.
Once the resources are allocated the process is ready to be ran by the scheduler. When the process is running its in the running state, when the process is waiting for I/O or events it goes to the Blocked state where it then goes to the ready state once the event or I/O is completed. When OS does not require the process it is terminated.

Process Control Block

Contains information required by the scheduler to work with multiple processes. Each block contains information about the process's state, pid, PC, register being used, memory limit, etc.

Process Queues

Different Types of Queues

Ready Queue: Are a queue used for processes in the ready state.
I/O queue: Are a queue used for processes in the waiting state.

Components of a Procedure

There are many components involved within a process. A process contains a **Memory image**, **The state of execution**, **Resources Allocated**, etc.

Context Switching

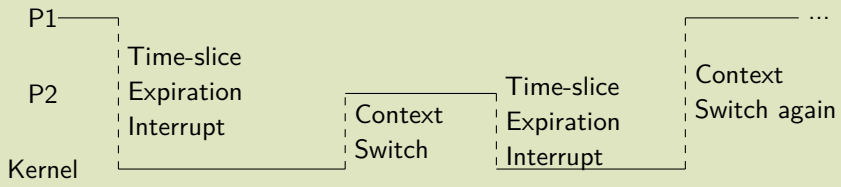
Context Switching

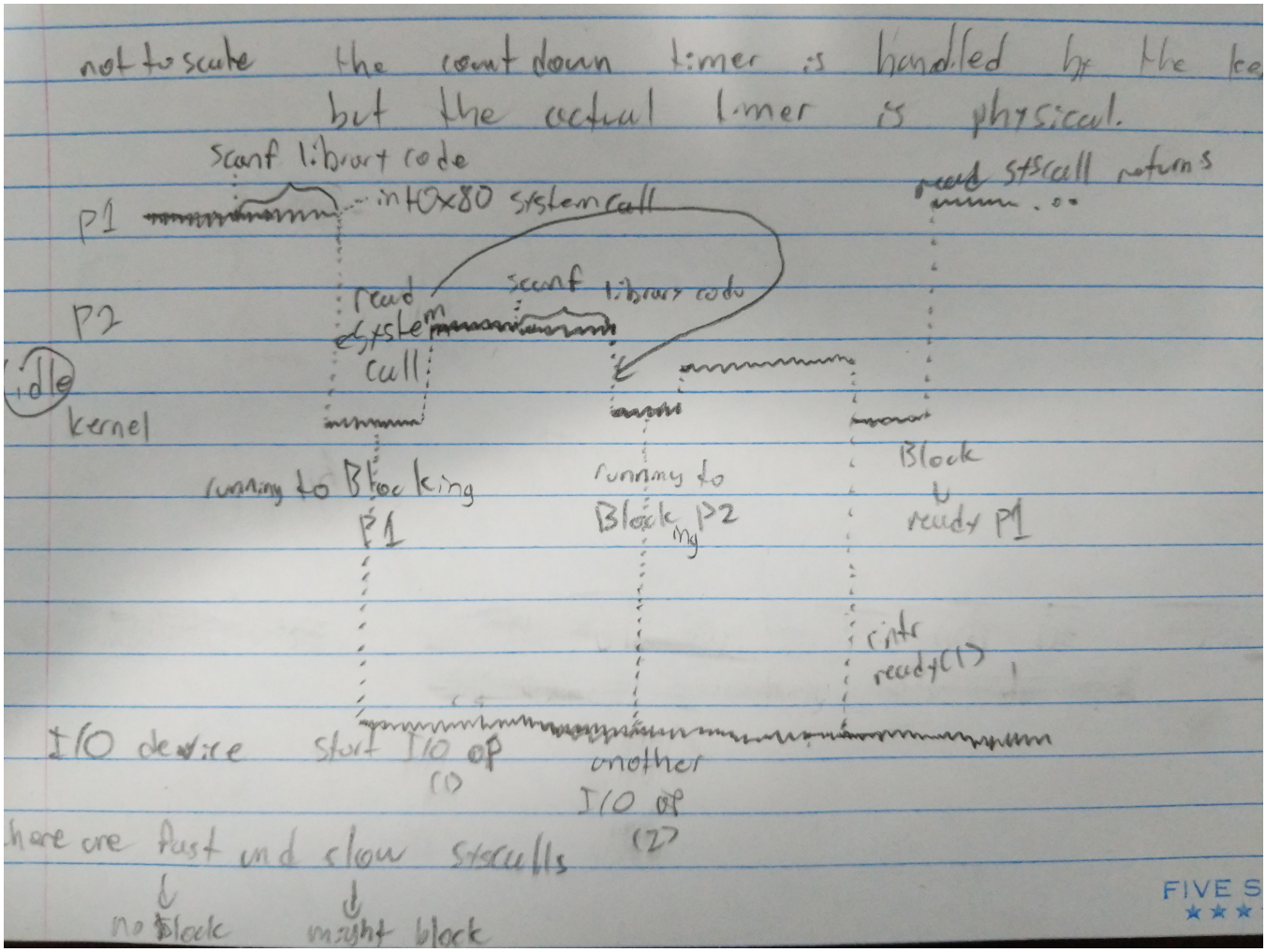
Context Switching: Is the transition from one process to another. The current process is **blocked** or paused while switching to the other process.

The timer used to tell the scheduler when to switch the process is done using hardware built into the processor. Once the timer has elapsed a certain amount of time it triggers a hardware trap that indicate to the scheduler that a new process should be processed.

Example

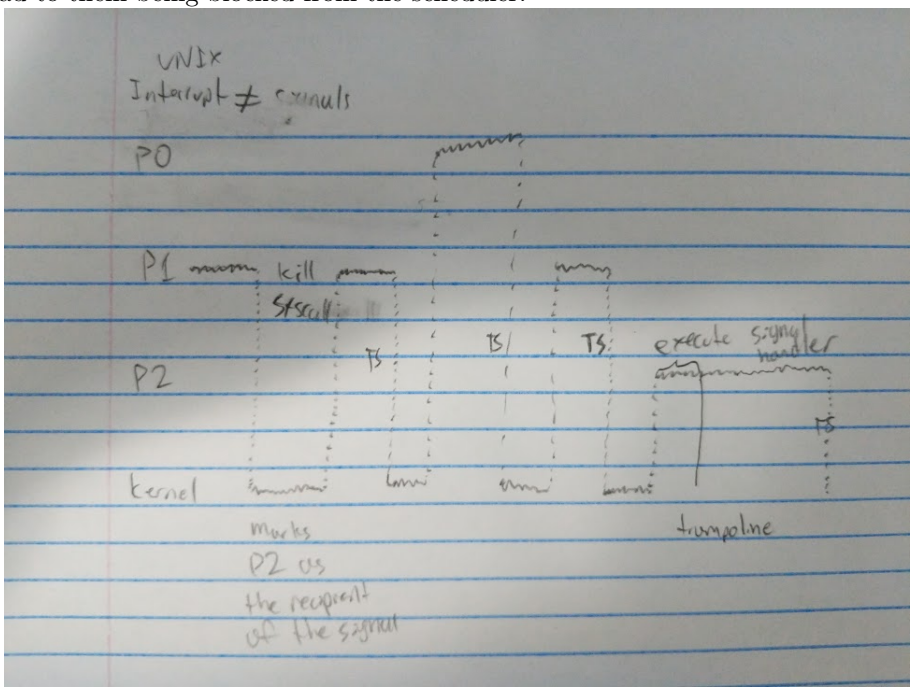
Here is an example of context switching between process P1, P2 and the kernel.

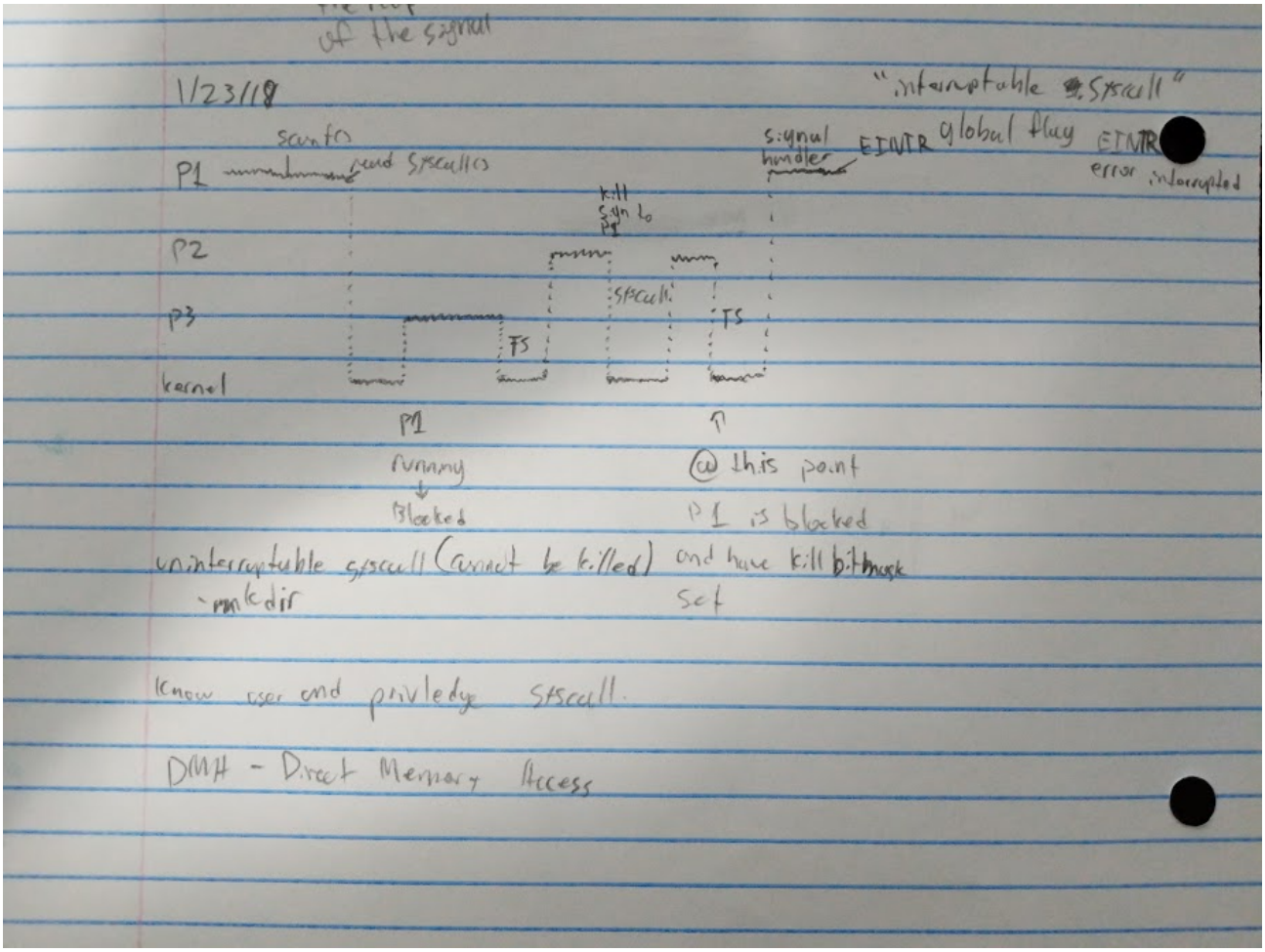




In the olden days there would be an idle process that the scheduler will jump into doing nothing if there were no processes in the ready state.

There are system calls that are fast and system calls that are slow. System calls that involve I/O operations are slow and might lead to them being blocked from the scheduler.





The diagram shown above shows a case where a process (P1) is blocked during which a kill signal is sent by process (P2). The default behaviour for most cases is that the signal handler in P1 will handle the signal and interrupt the read() system call but will have the EINTR (Error interrupt) flag set as the process is being interrupted.

Most system calls are interruptible, meaning the system call can be blocked if a signal is received. Some signals are not interruptible and therefore cannot be killed by standard means. For example mkdir is an uninterruptible system call.

CPU State

There are some instructions that are illegal for the user in **User Mode** to execute. These instructions are called privileged instructions and the CPU must switch to the **kernel, or supervisor mode**. In this mode the CPU can execute these instructions that have direct access to fundamental resources like peripheral devices, memory mapping, CPU state, etc.

To gain access to this mode the user must invoke a special machine instruction (system call) that simultaneously calls a special "function" and switch the CPU state into supervisory mode. The actual flag that set this mode is stored in the **PSW (Program Status Word)**.

We generally have two modes to improve security and reliability as different users will have different processes running and you don't want one program to kill the rest of programs.

Resource Management

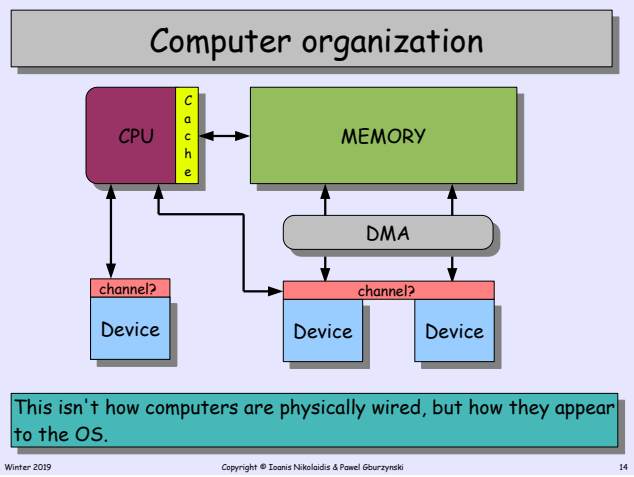
According to some point of view, operating systems are all about resource management. In the early days of computing, humans where the slowest links in the chain of computing as they took valuable time with loading and unloading of programs, debugging and so on.

Two ideas changed the world of computing

Batch processing: remove the clumsy human being from the computer room and have a queue of programs to run.

Multi-programming: find something to do for as many hardware components as possible. This only works if the components can operate **autonomously** and does not need to wait for other components to be finished.

Computer Organization



Component Defintions

Channels are special "processors" associated with an independent peripheral device. They connect the device to other components in the computer. Some channels can have multiple devices connected to a bus.

DMA or Direct Memory Access is a components that will load information between the computer's memory and its devices. This is beneficial as there could be a lot of information from the devices that will bog down the CPU if the CPU have to do loading and storing of everything.

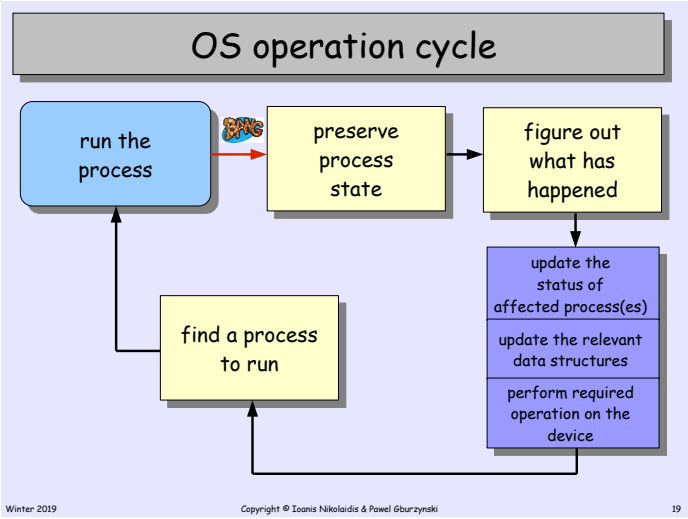
Activities

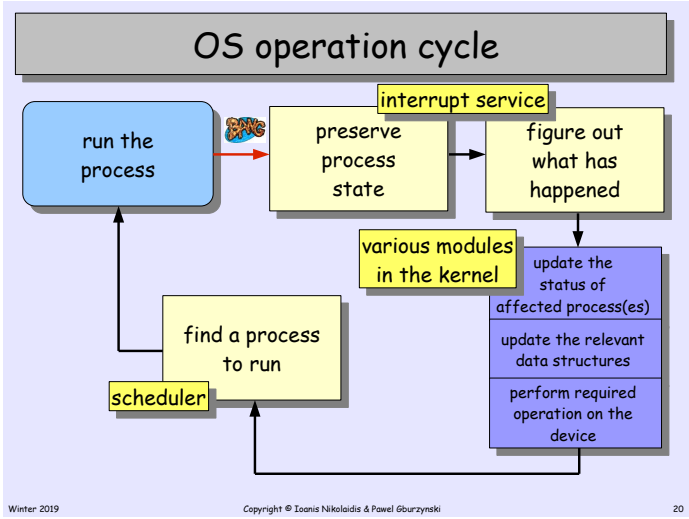
Typically, during its lifetime, your program can be **preempted** and then **resumed**, possibly many times. The system can handle other activities while running your program. There are computers today with multiple cores that can handle multiple processes, but in general we can consider the CPU as a "single" resource that the processors are competeting for. On general OSes are designed such that they provide **fair** and **efficient** stragtegies for handling competeting for resources. These competitors are called **processes**.

Terms

- **Processes**: Identifiable activities that can be stopped and resumed. They are described by special data structures and characterized by well-known sets of attributes. One such attribute is the process status
- **Interrupt service routines + syscall**: Activities to handle events in the system. Everything that happens in the system that may affect the status of a process boils down to the execution of an interrupt service routine
- **Others?**: Nope! The entire kernel is in fact a bunch of interrupt service routines, sometimes ina minor disguise.

OS Operating Cycle





Process Context

Informally, (process) context is the answer the following question:

> What needs to be saved such that a process can be suspended/frozen at an arbitrary point of its execution, such that it can be resumed at the exact same point later "as if nothing happened". Other than side-effects due to the passing of natural time, e.g., the current time advancing, the process is unable to determine on its own that this suspension/resumption cycle ever happened? (An "amnesia" of sorts.)

The answer must include:

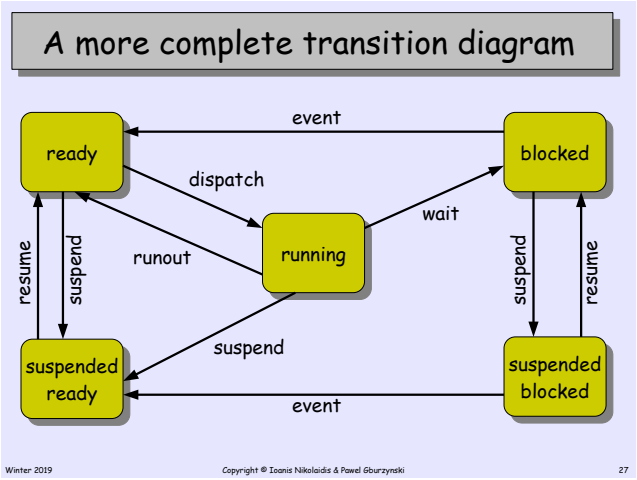
- The state of the address space (including heap, stack, etc.)
- The state of all the general purpose registers used
- The state of special purpose registers (PC, SP, etc.)
- The state of the process used resources (file ptr, etc.)

The transition between one context and another is called **Context Switch** and it generally considered an "expensive" operation. When terminating a process someone (usually the parent) needs to capture its exit code.

If the exit code is not caught by anyone then the PCB is not removed from the process queue and is considered a zombie process as it should be dead but the process is still alive by the OS

Suspended State

Sometimes we want to remove a process from the scheduler entirely but still want to keep its instance or state. Processes in suspended states are not considered by the scheduler to being operations so it is separated into its own separate pool. The process is then brought back to the scheduler pool once it resume back from suspension.



Threads vs Processes

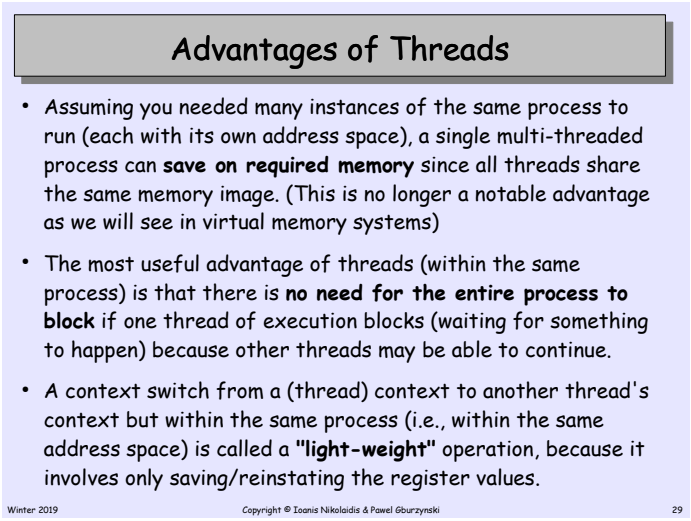
Difference between Threads and Processes

A **process** expresses a single execution path over the address space. **Threads** are (multiple) independent execution paths that share the same address space.

Each thread has, at a minimum:
- its own stack,
- its own register (incl. Special purpose one) values

Of course there are other luxuries like thread-private data and resources, but strictly speaking, they are not needed to have a multi-threaded system.

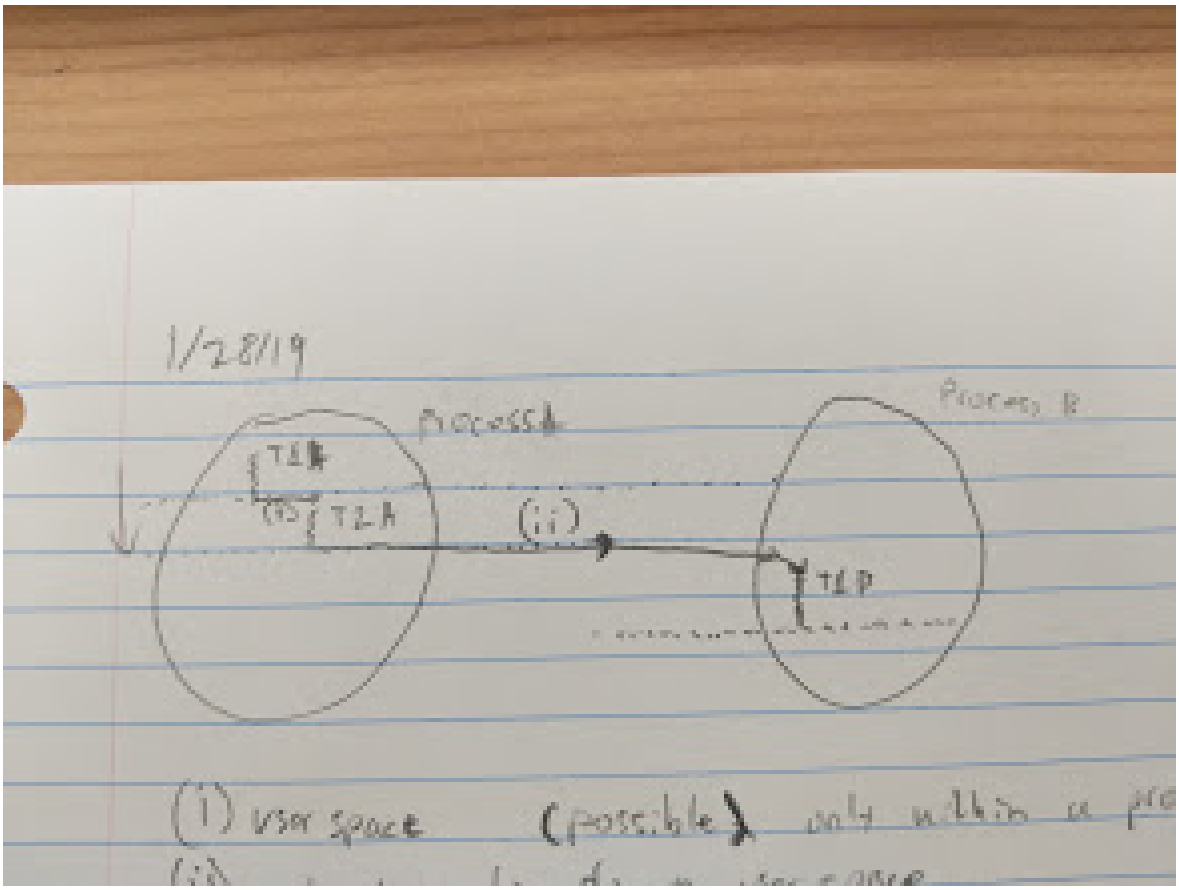
There are different types of threads. One performed by user code alone are called **user space threads**, threads performed by the OS are called **OS threads**, or you can have a hybrid type of (OS/User thread support).



Threads can be switched in user space if the threads are within the same process. Note the kernel can also switch thread within the same process if no code is provided for the switch.
Threads in different processes however, will need to call the kernel version of the thread switching instruction.

Generally the kernel operation of thread switching is done is two ways:

1. Time-slice Expiration (if no kernel thread support) OR
2. Kernel is aware of the threads within the process and it automatically.



Linux

Linux has, essentially, only threads, and hence each "PCB" corresponds to, and describes, a thread.

But Linux threads can share resources among them.

Hence, multiple threads sharing the same address space can be thought conceptually as being threads of a single process.

The PCB in Linux is **`task_struct`**

(in **`include/linux/sched.h`**)

The context switch is performed by **`__switch_to_asm`**

(in **`arch/x86/entry/entry_64.S`** for x86_64)

Digression: the term "Context"

- It means, in general, any state information that has to be maintained and managed carefully in order for correct, meaningful, computation (and not just by processes) to take place (or to save it and continue with it later).
- We will encounter use of the same term for interrupt handler execution (*interrupt context*) or execution within the kernel (*kernel context*). In each case there is a corresponding technicality of what is the scope of the context (boils down to at least the registers and stack involved). It is recommended to mention the type (process, thread, interrupt, kernel) corresponding to the context concept you are trying to convey. Do not let the meaning of context depend on context :-)

5: Synchronization

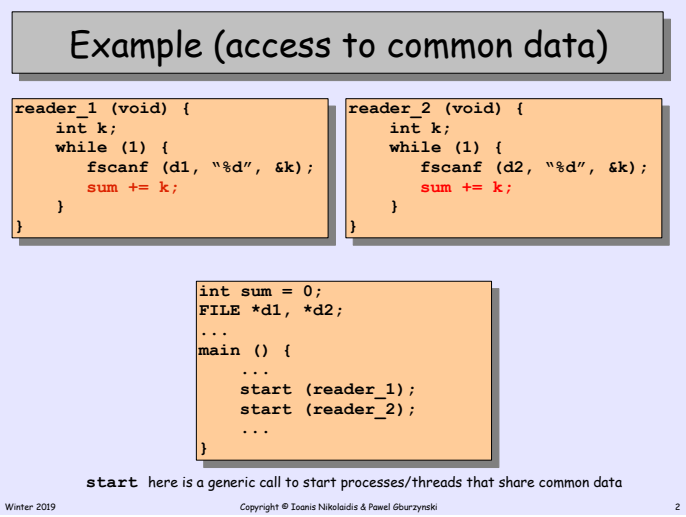
The synchronization of processes and threads are important to maintain the semantics of the program.

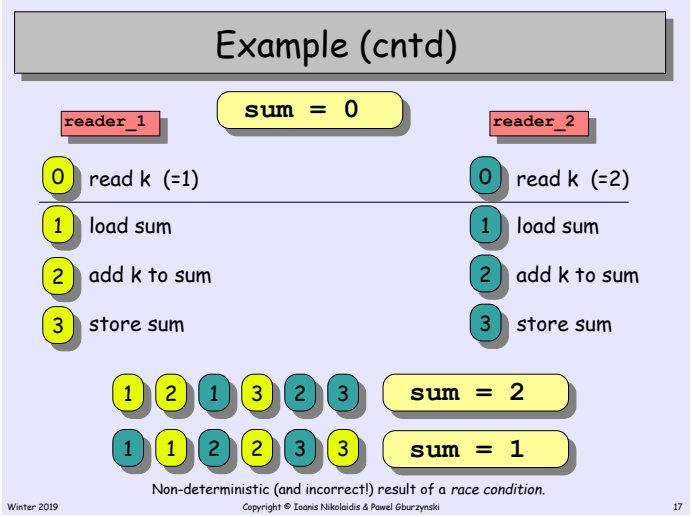
Suppose that we have a number of concurrent processes/threads

They could be interleaving where the CPU (single core) switches from one process to another over time. They could overlap where (in a multicore CPU) each core works on a different process.

Generally, unless we employ some mechanism, we have no way of telling in which way the instructions of the two processes will be interwoven in time. Sometimes it makes a difference.

Example of Thread Concurrency





The correct behaviour of this program is to add up all the integers between two files concurrently.
Bad ordering between the processes will lead to a race condition and incorrect behaviour.

We can use mutual exclusion to solve this issue.

Mutual exclusion

Where one thread of execution never enters its **critical section** at the same time that another concurrent thread of execution enter its own critical section. These critical sections may have operations on global variable where improper order of operations can lead to incorrect behaviour.

The critical sections in the code can be different from each other. The only reason why it's a critical section is becuase they have access to shared data and the concurrency of that data must be stable or sematically correct.

Note: There could be many independent critical sections so it's best to parameterize the operations.

What is required for concurrency

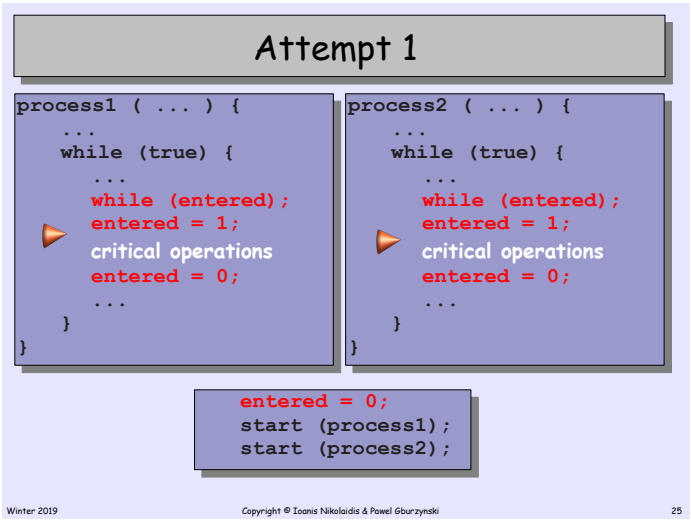
1. Processes execute concurrently (it makes no difference whether they are interleaving or overlapping). We never know which one will go next and for how long.

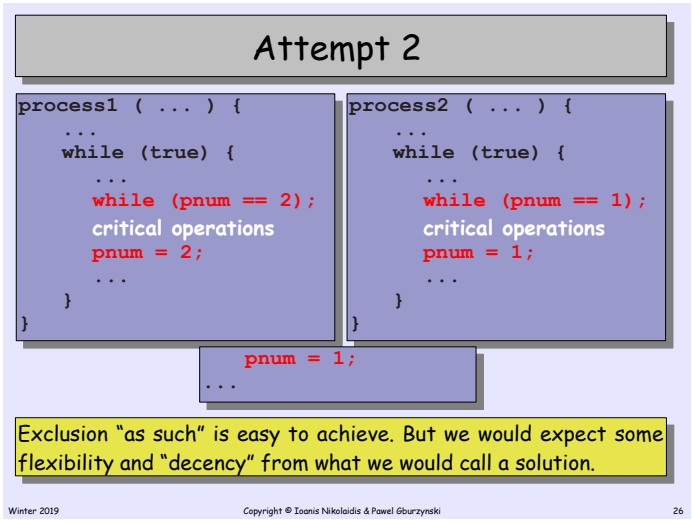
2. Consistency of basic operations is guaranteed in hardware. i.e., if two processes try to store something in the same location at the same time, the location is going to end up with one thing or the other thing but not with garbage.

Postulates

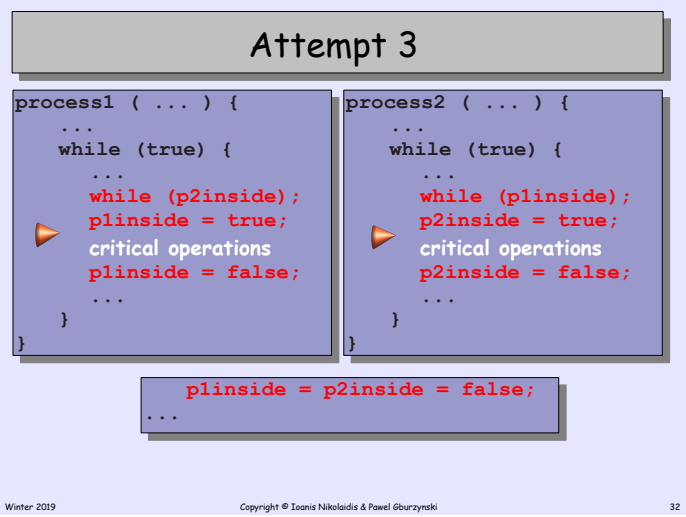
- **No "almost working" solutions**: If you think that some interleaving scenarios are not very likely, think again.
- **Simplicity is an asset**: Most critical sections are short, and the operations guarding them should not overshadow their cost.
- **No locked-in patterns of behavior**: Do not assume a particular pattern, e.g., that B will always follow A or anything like that
- **Determinism**:The amount of waiting time should not depend on circumstances other than for how long the section is going to be actually busy (no indefinite postponement).
- **Generalizability**: N processes should not be much more difficult to handle than 2.

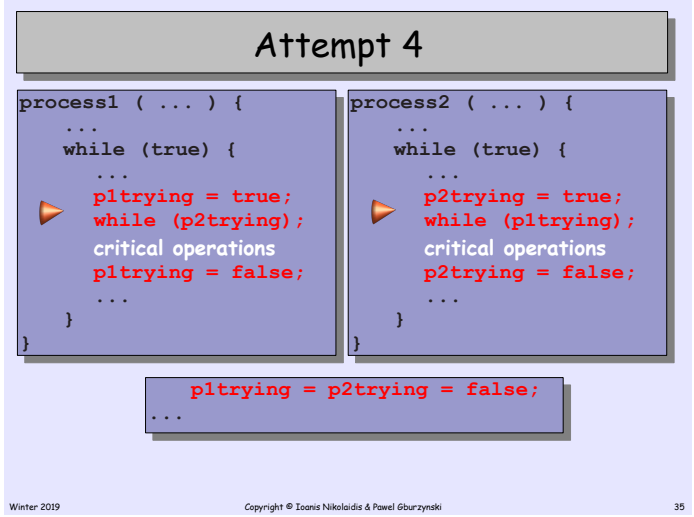
Example of mutual exclusion attempts



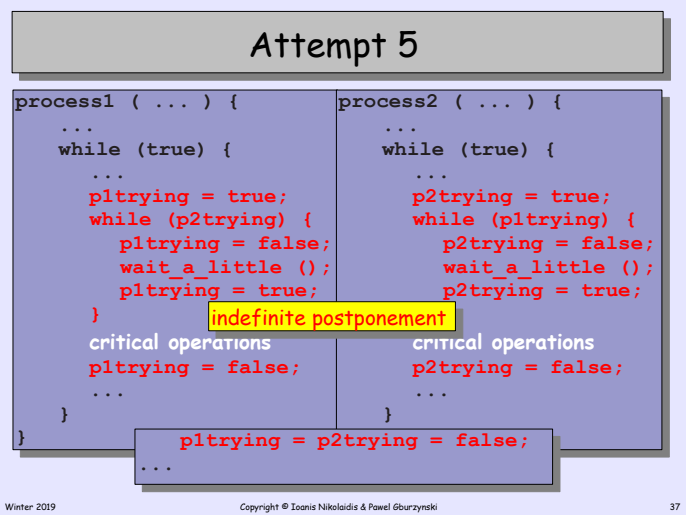


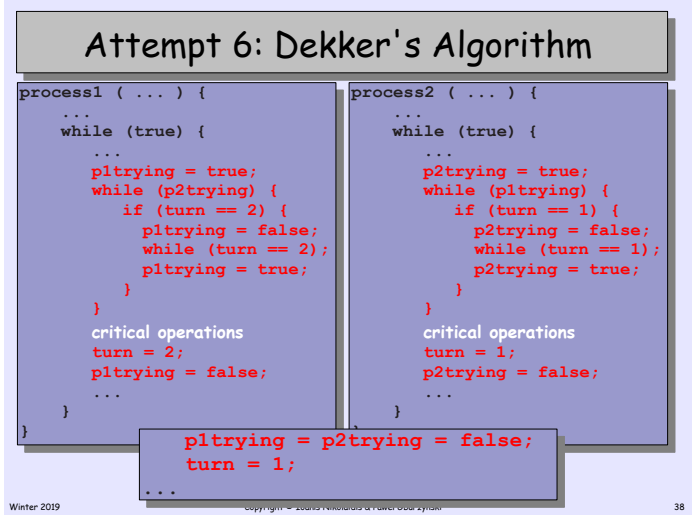
Attempt 1 is incorrect as the context switch can happen in between the while instruction and the entered = 1 instruction. Thus leading to the same problem as before. Attempt 2 is correct, but now we are setting the order and its basically the same as a sequential instruction. If one of the process takes too long then the other process will have to wait until that process is finished, wasting time and resoures.



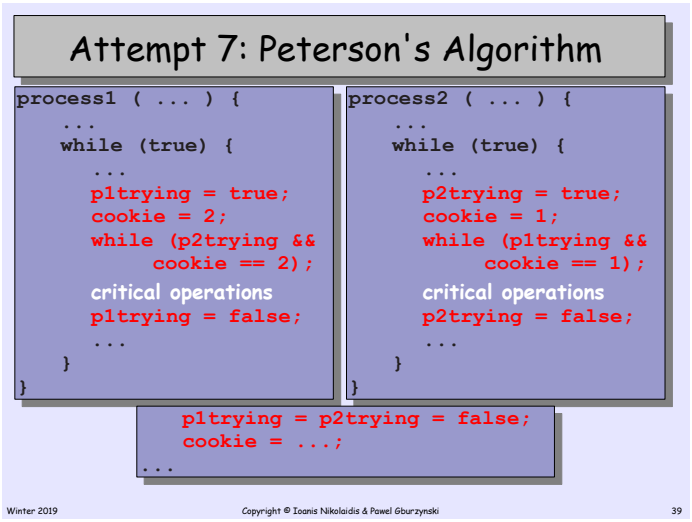


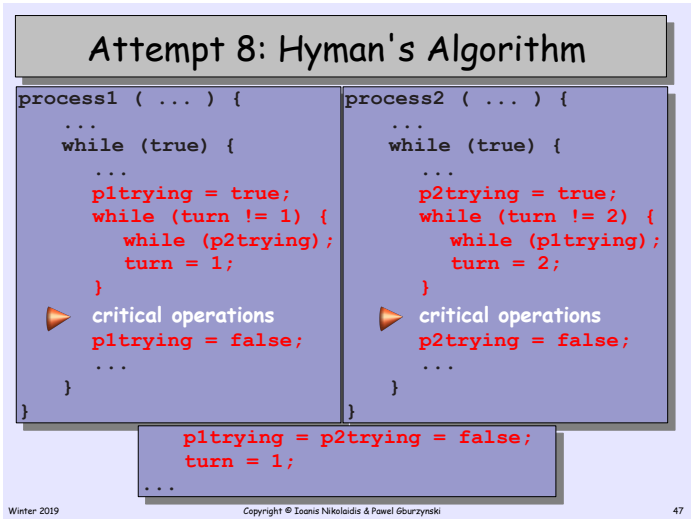
Attempt 3: Same problem as attempt number 1, you can break this mutual exclusion.
Attempt 4: Deadlock if you context switch in between the intent to enter and the while loop.





Attempt 5: Indefinite postponement, meaning each process can be waiting too long for the other to do something.
Attempt 6: Something that works finally, this type of mutual exclusion prevents the other process of entering the critical section based on its intent to enter the critical section.





Attempt 7: This does the same thing as the previous exmaple but puts preferences to a specific process.
Attempt 8: A more elaborate problem from attempt 1 and 3. If the context switch happened in the while loop after the second while loop, while the $PC of the other is before the true p1trying stage.

A Better Way

The best algorithms are relatively simple, but they only work for two processes. Generalizations are possible but:

- N must be known prior to the execution
- the algorithm becomes awfully complicated

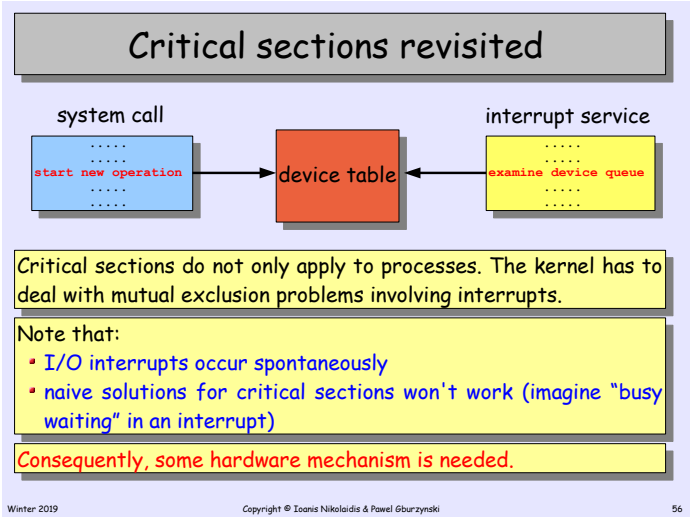
Real-life solutions are based on stronger prerequisites than variable sharing:

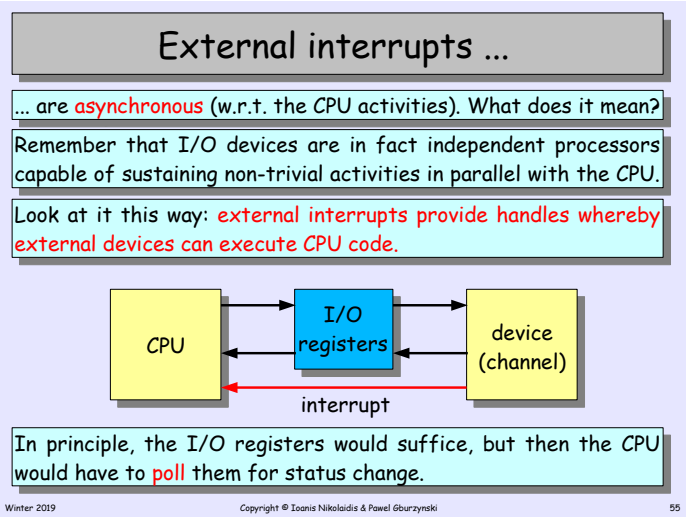
- the OS can provide requisite tools for processes (available as system calls
- some hardware tools can be used in desperate situations

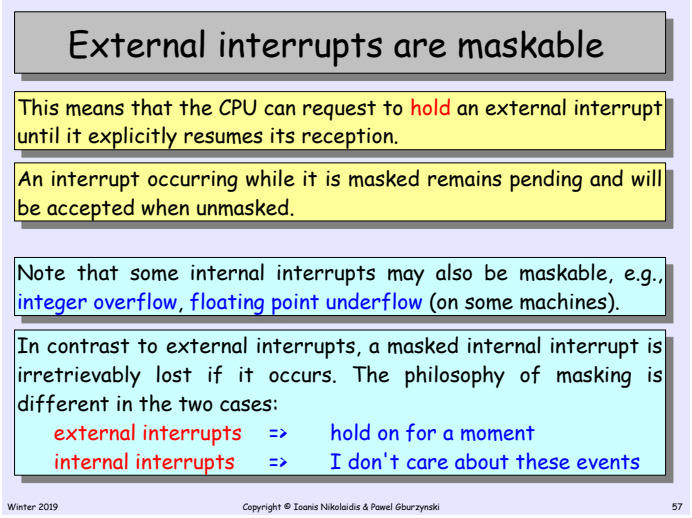
What we want is a function call that will prevent preemption while in the critical section but also parameterize the critical section so that multiple critical sections can exist within a program. We also don't want busy waiting, so we want the waiting process to give way to other processes.

Taxonomy of Interrupts

- **Internal Interrupts**: Sometimes called traps or exceptions. They are caused by events occurring **within** the CPU. Things like:
  - syscall
  - division by zero
  - illegal memory reference
- **External Interrupts**: They are caused by event occurring **outside** the CPU. Things like:
  - I/O notifications
  - timers going offset
  - reset button







Each interrupt have a priority level. Meaning the interrupting interrupt must have a priority higher then the current interrupt in order for the interrupt to take place. Otherwise, current interrupt is executed.

Intel x86 interrupts

All of Intel's interrupts are maskable except for one just in case. Also unlike the MC 68000, the default behavior of Intel's interrupt handling is that only one interrupt is handled at a time. There is no "priority" of interruption.

Another view of mutual exclusion

```
process1 ( ... ) {
    ...
    enter_section (A) ;
    critical operations
    exit_section (A) ;
    ...
}
```

```
process2 ( ... ) {
    ...
    enter_section (A) ;
    critical operations
    exit_section (A) ;
    ...
}
```

```
enter_section (A)
```

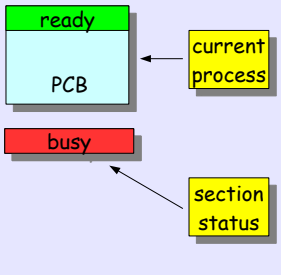
```
some_process ( ... ) {
    ...
    wait_for_event (A) ;
    ...
}
```

```
interrupt_hndlr ( ... ) {
    ...
    trigger_event (A) ;
    ...
}
```

This looks like a wait/wakeup mechanism, i.e., event passing.

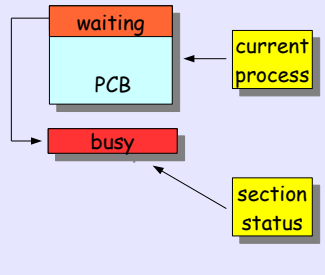
How most systems would do it

enter system call
do not preempt
check section status
set section busy
preempting OK
return from syscall



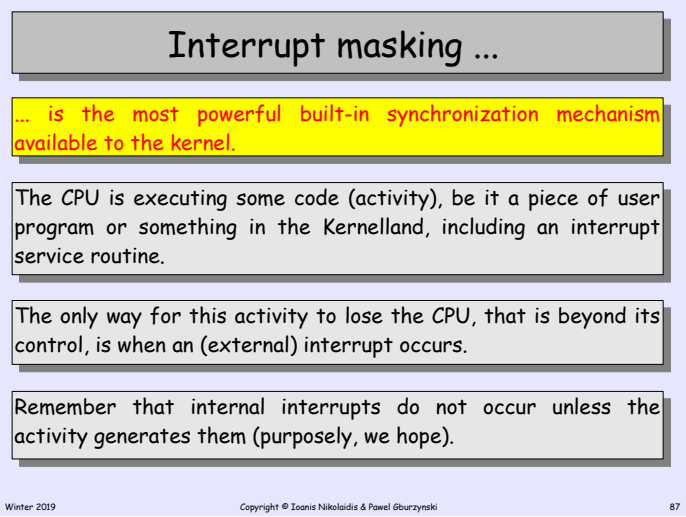
How most systems would do it

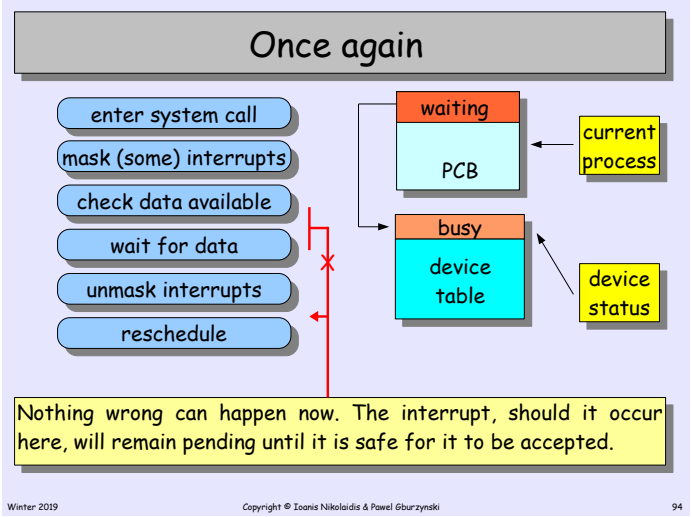
enter system call
do not preempt
check section status
mark as waiting
preempting OK
reschedule



Implementing "do not preempt" is easy: the kernel may simply make sure that no rescheduling will occur for a while. To make their lives easier, many systems do not allow rescheduling to occur at all while a process is in the Kernelland.

As you can see in this example the current PCB is now waiting as someone else is currently accessing shared data. The same idea works for devices.





Masking interrupts are important as some data may show up while the process is checking for data. The device status may change by an interrupt service routine so the process is waiting even though the device is ready to transmit but can't as the process is not checking for available data.

6: Synchronization Abstraction

Semaphores

Counting Semaphores

```
wait(S):
    if(S > 0) S--;
    else wait until allowed to proceed;

signal(S):
    if(processes wait on S)
        allow one to proceed;
    else S++;

{ in main code }
repeat
...
wait(S);
critical section
signal(S);
...
until false;
```

What does P and V mean

P means "to test"

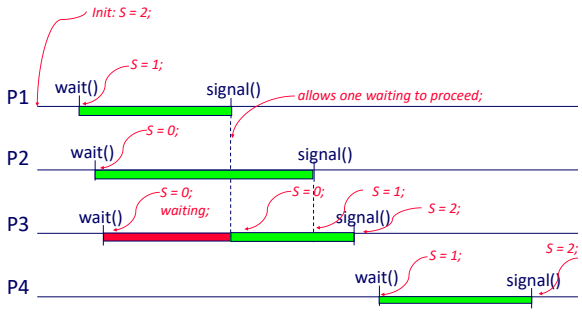
V means "to increment"

Where: S is the number of process(es) that can access the critical section. Note this looks very similar to the waiting algorithm in the previous chapter. If S is initialized to 1, then it is a binary semaphore.

This works well if you programmed it correctly, if programmed incorrectly like flipping the order of wait and signal then the semantics of the algorithm is lost.

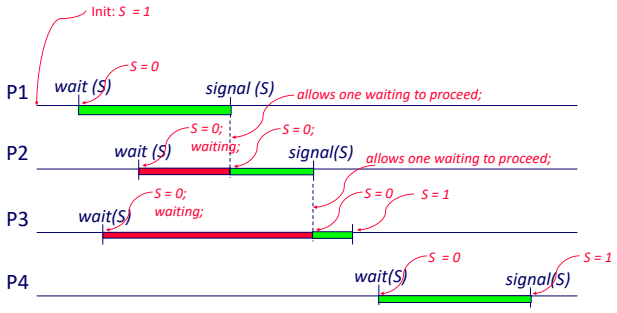
Example of a Counting Semaphore

A counting semaphore example



CMPUT 379, Winter 2019 Synchronization Abstractions 3

A binary semaphore example



Note: we did not define (so far) any order in which multiple processes waiting on a semaphore are to resume execution.

CMPUT 379, Winter 2019 Synchronization Abstractions 4

Atomicity & semaphore implementation

Here is a basic implementation of a semaphore.
Note the two functions must be atomic, they cannot be blocked. Moreover, S can only be modified by one process at a time.

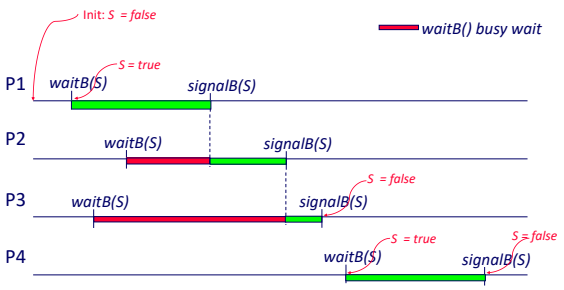
```
1   wait(S){
2       while (S <= 0); // busy wait
3       S--;
4   }
5
6   signal(S){
7       S++;
8   }
```

Because the standard implementation is inefficient as it can lead to busy waiting. The implementation below is better.

```
1   typedef struct {
2       int value;
3       struct process* list;
4   } semaphore;
5
6   wait(semaphore* S){
7       S->value--;
8       if(S->value < 0){
9           S->list.push_back(this->process)
10          block();
11      }
12  }
13
14  signal(semaphore* S){
15      S->value++;
16      if(S->value <= 0){
17          process = S->list.pop_front();
18          wakeup(process);
19      }
20  }
```

```
{ Test and Set version of
a binary semaphore }
waitB(S):
    while Test_and_Set(S) do no-op;
signalB(S):
    S = false;
```

A binary semaphore example (T&S)



Note: with the busy wait implementation we have also given up any hope we will control the order of resumption.

This a test and set semaphore that sacrifices efficiency for simplicity of implementation. This code is similar to waiting until time slice expiration.

Note

This is a valid solution for multi-core computers where multiple cores are accessing the same critical section at the same time.

Implementation of Binary & Counting Semaphores

Binary semaphores (T&S implementation)

```
S.guard = false;
S.flag = true;
S.queue = EMPTY;
```

```
waitB(S):
    while Test-and-Set(S.guard) do no-op;
    if (S.flag) do {
        S.flag = false;
        S.guard = false;
    } elsedo {
        enqueue(S.queue);
        … and S.guard = false; …
    }

signalB(S):
    while Test-and-Set(S.guard) do no-op;
    if (waiting(S.queue)) do
        makeready(S.queue);
    elsedo
        S.flag := true;
    S.guard = false;
```

Taking matters under control, we will build a binary semaphore with T&S but *without* busy waiting (well almost).

Implementing counting semaphores

(Using Binary Semaphores)

```
wait(S):
    waitB(S.S1);
    S.C := S.C - 1;
    if S.C < 0 then
        begin
            signalB(S.S1);
            waitB(S.S2);
        end
    signalB(S.S1);
```

```
signal(S):
    waitB(S.S1);
    S.C := S.C + 1;
if S.C ≤ 0 then signalB(S.S2);
    else signalB(S.S1);
```


Adding some extra code can help with the busy waiting.
You can use two binary semaphores to implement a counting semaphore. One semaphore is used to protect the counter (value) and the other is a place to wait.

During initialization let:
int n;
semaphore mutux = 1;
semaphore full = 0;
semaphore empty = n;
Where n is the size of the pool of buffers.

Producer-consumer (bounded-buffer) revisited

```
repeat
        ...
        produce item in nextp
        ...
        wait(empty);
        wait(mutex);
        ...
        add nextp to buffer
        ...
        signal(mutex);
        signal(full);
until false
```

```
repeat
        wait(full);
        wait(mutex);
        ...
        remove item from buffer to nextc
        ...
        signal(mutex);
        signal(empty);
        ...
        consume item in nextc
        ...
until false
```


This is a circular buffer semaphore where "producers" add items into a finite buffer and "consumers" take items from the an array of buffer.

The mutex semaphore is used to prevent access to the pool of buffers when the pool is being modified.
The empty and full semaphores are interconnected with each other.

The producer uses the empty semaphore to check if the pool is not full of filled buffers. If not then the producer tries to get access to the buffer and modifies it, decrementing the empty counter and signaling the full semaphore to increment their counter by one as the one of the buffers became full.

The consumer uses the full semaphore to check if the pool is not full of empty buffers. If not then the consumer tries to get access to the buffer and modifies it, decrementing the full counter and signaling the empty semaphore to increment their counter by one as one of the buffers became empty.

Readers and Writer Problem

```
// During initialization let:
int read_count = n;
semaphore rw_mutex = 1;
semaphore mutex = 1;

// writer process
do{
    wait(rw_mutex);
    /* writing is performed */
    signal(rw_mutex);
}while (true);

// reader process
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    /* reading is performed */
    wait(mutex);
    read_count--;
    if(read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while(true);
```

This is the readers and writers problem where writers cannot update the data when there are readers reading the data. In this is a unique case as the preference of mutual exclusion is given to the reader as you can have multiple readers looking at the data without the worry of race conditions. Writers however, must wait until no process is reading the data in order for it to update.

Limitation of classic semaphores

Lack of enforcement schemes.
User processes can call wait and/or signal in arbitrary order:

- mutual exclusion is possibly violated, and/or,
- deadlock may occur

Processes must be willing to block when invoking wait().
The resumption strategy of signal is set and not controlled by the user.

- No additional control arguments.
- Process selection in signal should prohibit starvation,
- but fairness (proportional to "speed" of a process) is non-trivial.

Monitors

Monitors encapsulate the shared data into an object. Processes can access the object using procedure entry points or through specific synchronization calls (related to condition variables).
Conditional Variables are:

- Are not an ordinary variable. Similar to a structure.
- you have multiple variables
- signify "things" you can wait once
- They are not conditions (conditional expressions)
- Really they are just user-defined synchronization schemes

Many condition variables can be defined in a monitor.
The operation x.wait() means that the process invoking this operation is suspended until another process invokes x.signal(). x.signal() resume exactly one suspended process. If no process is suspended, then the signal() operation has no effect, i.e., the state of x didn't change.

Monitor Semantics

If a process executes x.signal(), should it be allowed to execute beyond the x.signal() if a process was already waiting in x.wait() and was resumed as a result of x.signal()?

1. The process that issued the x.signal() waits until the process that was waiting in x.wait() either exits the monitor or waits on another condition. (Hoare)
2. The process that was waiting on the x.wait() waits until the process that issued the x.signal() leaves the monitor or waits on a condition.
3. The process that was waiting on the x.wait()resumes immediately but the one that called x.signal() must leave the monitor (Concurrent C)

Monitor Implementation of a Binary Semaphore

You can use monitors to implement a binary semaphore.

```
var busy: Boolean, x: condition;

procedure wait():
    if busy then x.wait;
    busy = true;

procedure signal():
    busy = false;
    x.signal;

init:
    busy = false;
```

One key difference between this implementation and the "classic" implementation is that the signal becomes a no-op if they isn't anyone waiting.

Semaphore implementation of monitors (Hoare)

The complication is that a process that signals on a condition variable must, itself get suspended from executing (hence has to wait) because the signaled process will now resume execution within the monitor. This is a different "kind" of waiting than the waiting to (originally) enter the monitor when a monitor method was called. We introduce one new semaphore (next) to deal with this.

1. You have processes waiting to enter the monitor (wait on mutex)
2. You have processes waiting on the "next" semaphore
3. You have processes waiting on the condition variable (x.sem)

Semaphore	
mutex	serialize access to operations in monitor (= 1)
nextsem	suspend processes after executing SIGNAL (= 0)
x.sem	suspend a process executing WAIT on condition x(= 0)
int	
next	number of processes waiting due to signal (= 0)
x.count	number of processes waiting due to waiting on x_i

```
function_j(){
    wait(mutex);
    <operation body>
    if (next > 0)
        signal(nextsem);
    else
        signal(mutex);
}
```

```
x.wait(){
    x.count++;
    if(next > 0)
        signal(nextsem);
    else
        signal(mutex);
    // process waits here
    wait(x.sem);
    x.count--;
}
```

```
x.signal(){
    if(x.count > 0){
        next++;
        signal(x.sem);
        wait(nextsem);
        next--;
    }
}
```

Java Example

Objects provided mechanism that could be used to implement monitors. A monitor class needs to use only private data fields, and use synchronized methods.

Each object possesses a lock and a wait queue. The lock is access by the synchronized (obj) ... block. In this form the argument to synchronized defines the object whose lock we are interested to access.
A method can be defined with asynchronized attribute. Really a shorthand for synchronized (this) ... of the outermost block (containing the entire code of the function).

Each object contains a single anonymous condition-like structure. The condition is accessed via calls to wait() / notify()
wait() causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

wait(), notify() or notifyAll() can be called on an object only by a thread that currently holds the lock to the object.

wait() the thread calling it is placed in the wait queue of the object. The lock to the object is released.

notify() if a thread is waiting at the object's wait queue, it is made ready. The thread waits to re-obtain the lock on the object. (Could be blocked from acquiring it because the thread that executed the notify() has not yet released it, or because another thread got it in the meantime.) After the lock is obtained, the thread resumes at the point following the wait(). The logical condition that was true when the notify() was invoked is not necessarily true when the thread that was waiting resumes running.

notifyAll() is like notify() but all threads in the wait queue of the object are made ready. Still, each of the ready threads needs to re-acquire the lock before continuing after the wait().

Semaphore

```
private class Semaphore {
  private int S;
  public Semaphore(int initial)
  {
    S = initial
  }
  public synchronized void waitSema()
  {
    S--;
    while(S < 0) wait();
  }
  public synchronized void signalSema()
  {
    S++;
    if(S <= 0) notify();
  }
}
```

Bounded Buffer and Producer-Consumer

```
Class BBuffer
{
  final int size = 10;
  int[] thebuffer = new int[size];
  int indexPut = 0, indexGet = 0;
  int count = 0;

  public synchronized void put(int item)
  {
    while (count == size-1) wait();
    buffer[indexPut] = item;
    indexPut = (indexPut+1)%size;
    count++;
    if (count == 1) notifyAll();
  }

  public synchronized int get()
  {
    int item;
    while (count == 0) wait();
    item = buffer[indexGet];
    indexGet = (indexGet+1)%size;
    count--;
    if (count == size-1) notifyAll();
    return item;
  }
}
```

Deadlock

```
Class Swapper {
  private int value;
  public synchronized int getValue()
  {
    return value;
  }
  public synchronized void setValue (int i)
  {
    value = i;
  }
  public synchronized void swapValue
  (Swapper s)
  {
    int temp = value;
    value = s.getValue();
    s.setValue(temp);
  }
}
```

Limitation of Monitors

- Unlike semaphores, signal()s in monitors do not accumulate.
- Condition variables are not linked directly to logical (Boolean) expressions.
- Evaluation of the logical condition (if any) is left up to the programmer.
- A monitor's state (that led to a particular "condition") may change (depending on the monitor semantics) either
  - between signal() and resumption of a wait()-ing thread or
  - between signal()and resumption of the thread that signal()-ed.
- Users are free to call into a monitor in any (not always "correct") order.
- Nesting of calls into monitors needs to be addressed.
- Visibility of the monitor internal variables needs to be addressed

Atomic Transaction

Atomic transactions

Gain access to resources...
...
account[i] := account[i] - sum;
...
CRASH!!!!
...
account[j] := account[j] + sum;
...
Release allocated resources...

CMPUT 379, Winter 2019 Synchronization Abstractions 27

Change should not be committed before the end of the transactions. An "all or nothing" property.
In the event of a crash, recovery is possible:

- log-based recovery
- checkpoints.

Concurrency introduces new challenges.

Serializability

Serializability
The outcome of the concurrent execution of atomic transactions should be equivalent to the sequential execution of the same transactions in an arbitrary order.

Serializability example

(serializable execution)

T0	T1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

(an equivalent sequential execution)

T0	T1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

CMPUT 379, Winter 2019 Synchronization Abstractions 30

Conflicting operations

- Write/write (overwriting uncommitted data)

  Example: transaction T0 overwrites value A with A'. Transaction T1 overwrites value B with B'. Then T0 overwrites value B' with B". Finally, T1 overwrites A' with A". The result is non-serializable. (Note all updates are without previous reads, i.e., "blind")

- Write/read (reading modified, but uncommitted, data)

  Example: Transaction T0 reads A and writes A'. Transaction T1 reads A' and writes A". Then transaction T0 does something else, but does not commit (aborts). The state is rolled back to A. Now it is as if T1 has read a value that never existed!

- read/write (unrepeatable reads)

  Transaction T0 reads value A. Transaction T1 updates A to A' and commits it. T0 then re-reads the value and gets A'.

Attaining concurrency and serializability

Associate each data item with a lock.

- must posses lock to access the data item
- Lock Modes:
  - Shared: allows reading only, others can also lock as shared (to prevent read/write conflicts)
  - Exclusive: allows reading and writing, others cannot hold the lock (to prevent write/read and write/write conflicts)
- Locks are not enough, you must know when to release them. The simplistic approach: obtain all at the beginning release all at end.
- A better approach: Two phase locking: First phase: growing phase (may obtain but not release), Second phase: shrinking phase (may release but not obtain).

General Problem of Synchronization

Our first priority is correctness, i.e., concurrent operations should leave the system in a consistent, i.e. "correct", state, but there are more things we need to address:
Livelock, or starvation: the situation in which some processes are making progress toward completion but some others are locked out of the resource(s). (could manifest as unfairness)
Deadlock: the situation in which two or more processes are locked out of the resource(s) that are held by each other. Progress is then impossible among the deadlocked processes.

7: Deadlocks

Deadlock
A process is said to be in a the state of deadlock when it is waiting for something that will probably never happen.

Resource Allocation

Traditionally, the deadlock problem is discussed in terms of resource allocation: a process holds some resources and needs more to complete. A serious problem occurs if there is no sensible (non-destructive) way to remove the resources from the process. By stretching our idea of a resource, we can define all (non-trivial) kinds of deadlock as resource allocation problems. Remember: critical sections are resources.

How to avoid deadlock?

There are many ways to avoid a deadlock

1. provide enough resources for everybody, such that they will never get into problems
2. teach the "processes" to be courteous and yield
3. teach them how to fight if other parties are not courteous
4. impose a protocol (restrictions) on the resource allocation scheme. Like a traffic signal.

Resource Types

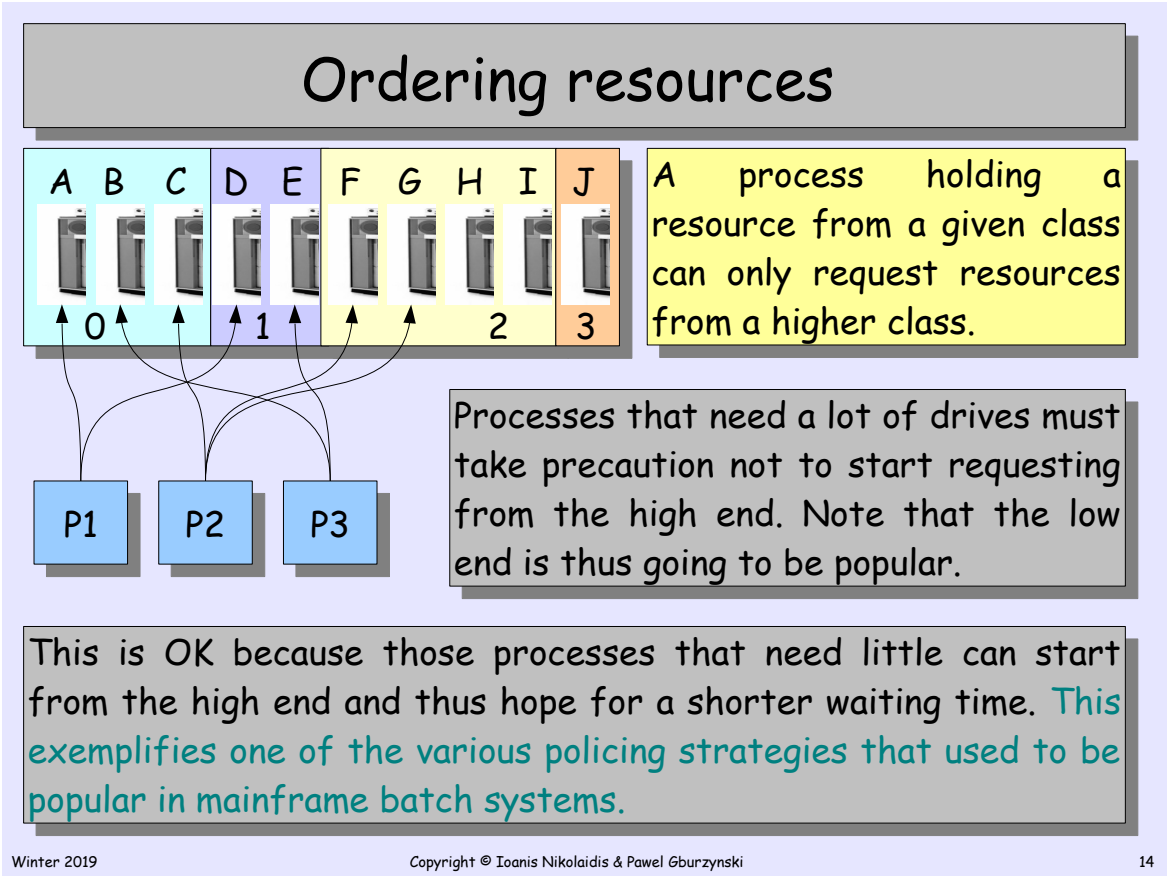
Types of resources
Preemptible: can be taken from the process and later returned without affecting the process's integrity. For example, CPUs are generally preemptible.
Non-preemptible: once a process acquires the resource, it must remain allocated to the process until explicitly released. For example, Tape drives, USB storage keys, are non-preemptible resources.

Necessary conditions for a deadlock

Deadlock conditions and their denials

- **Mutual Exclusion:** Resources are not shared. Processes holding them require mutual exclusion (exclusive access).
- **Hold + wait:** Processes are allowed to hold resources while waiting for additional resources. **Prevention** Processes cannot request resources incrementally. You must receive everything you are ever going to need at the beginning. If your request cannot be fulfilled immediately, you have to release everything, and re-request everything back together with the new item.
- **No Preemption:** Resources cannot be removed from processes and later returned unless the process releases them explicitly. **Prevention** If your request cannot be fulfilled immediately, you have to release everything, and re-request everything back together with the new item.
- **Circle:** One can identify a circular chain of processes and resources, with each process holding a resource needed by the previous process and requesting a resource held by the next process. **Prevention** You must request resources in a certain order. This order is followed by everybody.

Ordering Resources



You can have different ordering depending on the resource requirement of the process. Higher requirements of resources can get their requirement from the low end while lower requirement of resources can get their requirement from the high end.

Banker's Algorithm

Suppose that there is only one resource type with M units. These are the rules:

Rules

1. Each process (job) declares in advance the maximum number of units that it will be needing at any given time. This declaration cannot be more than M.
2. The system aborts a process that violates its declaration.
3. A process may request an arbitrary number of units at once, as long as the request does not violate its declaration. The process may have to wait, but the system guarantees that as long as processes eventually complete (and thus release what they've been holding), all processes will happily execute to successful completion.

Safe vs Unsafe State

Safe vs Unsafe State

A system is in a safe state if and only if the system can fulfill the maximum possible amount of resources of the "smallest" process ready to be executed.

If it cannot then the state is unsafe and a deadlock can occur.

General Case and Algorithm

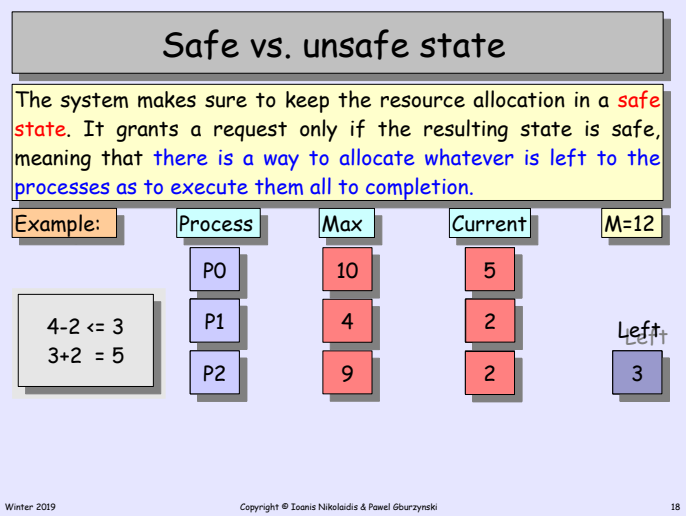
n	the # of processes
m	the # of resource Types
av[j]	the # of resource units of type j still available
Max[i,j]	the max demand of process i for resource j
Al[i,j]	current allocation of resource j to process i
N[i,j]	remaining need for process i for resource j
N[i,j]	= Max[i,j] - AL[i,j]
Av[j]	= T[j] - $\sum_i$ Al[i,j]

Where T[j] is the total number of units of resource j available in the idle system.

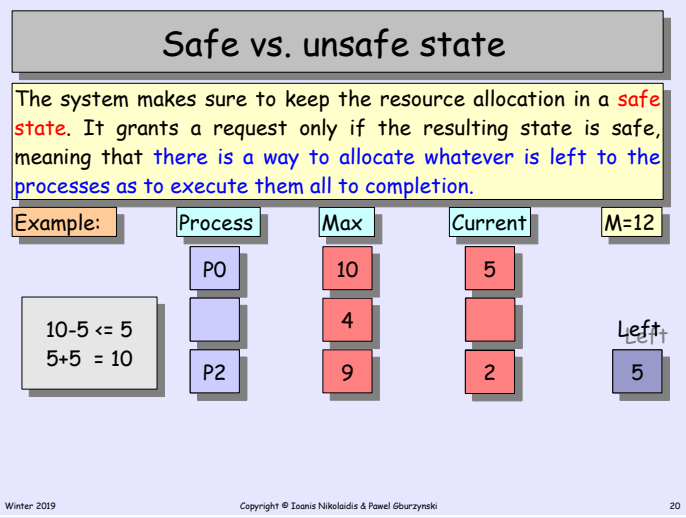
```
void allocate (int j, int k) {
  if (k > N[current,j])
    abort (current);
  while (1) { // lock
    if (k <= Av[j]) {
      Av[j] -= k;
      Al[current,j] += k;
      N[current,j] -= k;
      if (safe ()) {
        grant (j, k); // unlock
        return;
      }
      Av[j] += k;
      Al[current,j] -= k;
      N[current,j] += k;
    }
    wait (RESOURCE_RELEASE);
    schedule (); // unlock
  }
};
```

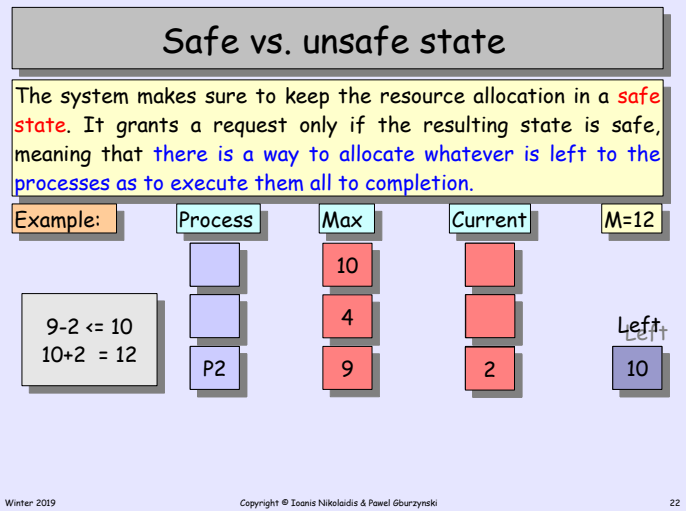
Safety Check

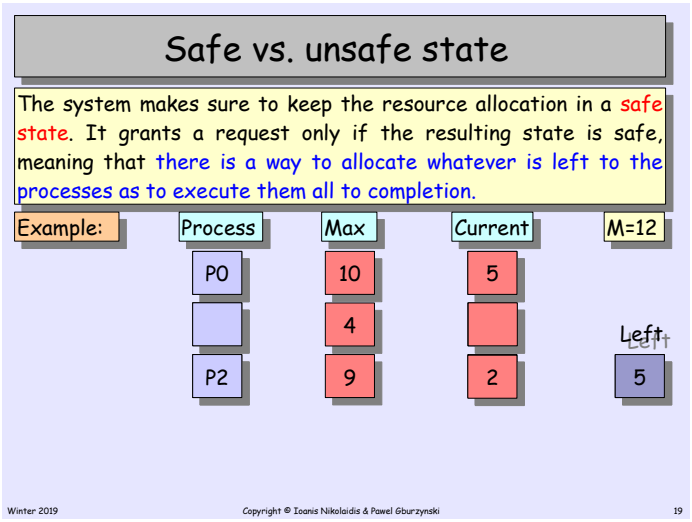
```
bool safe () {
  int W[m], i, j; bool F[n], alldone;
  for (j=0; j<m; j++) W[j]=Av[j];
  for (i=0; i<n; i++) F[i]=false;
Restart:
  alldone = true;
    for (i=0; i<n; i++) {
      if (!F[i]){
        for (j=0; j<m; j++)
          if (N[i,j] > W[j]) break;
        if (j == m) {
          for (j=0; j < m; j++) W[j] += Al[i,j];
          F[i] = true;
          goto Restart;
        }
      alldone = false;
      }
    }
  return alldone;
}
```

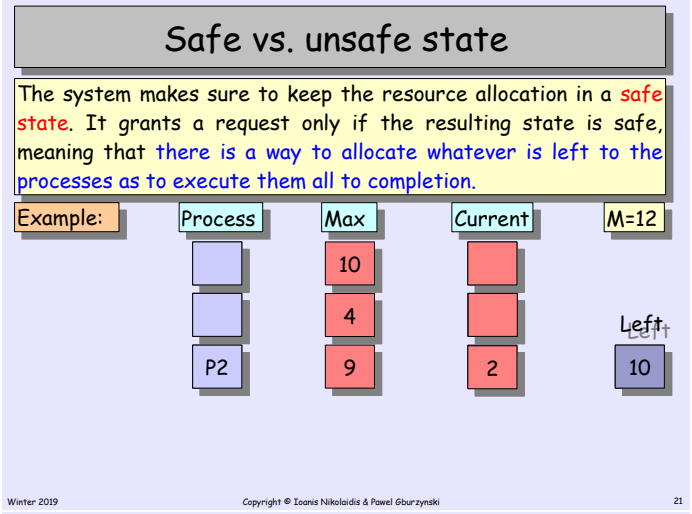


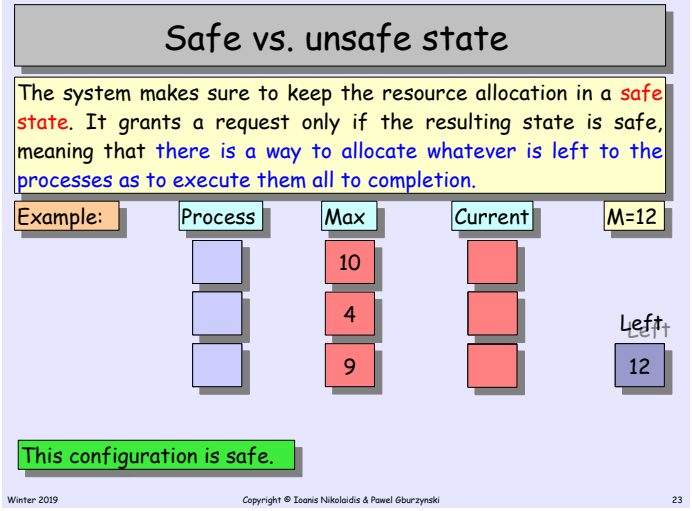
Process 1 can be processed as it only requires 2 more "resources" to reach its max. It is consumed and its resources released back into the pool. This repeats for process 0 then 2.











Safe vs. unsafe state

The system makes sure to keep the resource allocation in a safe state. It grants a request only if the resulting state is safe, meaning that there is a way to allocate whatever is left to the processes as to execute them all to completion.

Example:

Process	Max	Current	M=12
P0	10	5	
P1	4	2	
P2	9	3	
		Left	2

How about this one?

Safe vs. unsafe state

The system makes sure to keep the resource allocation in a safe state. It grants a request only if the resulting state is safe, meaning that there is a way to allocate whatever is left to the processes as to execute them all to completion.

Example:

Process	Max	Current	M=12
P0	10	5	
	4		
P2	9	3	
		Left	4

Safe vs. unsafe state

The system makes sure to keep the resource allocation in a safe state. It grants a request only if the resulting state is safe, meaning that there is a way to allocate whatever is left to the processes as to execute them all to completion.

Example:

?

Process	Max	Current	M=12
P0	10	5	
	4		
P2	9	3	
		Left	4

Note that unsafe state does not imply deadlock! The process's Max declaration may have applied to some moment in the past, and the process may not request that many items any more.

Safe vs. unsafe state

The system makes sure to keep the resource allocation in a safe state. It grants a request only if the resulting state is safe, meaning that there is a way to allocate whatever is left to the processes as to execute them all to completion.

Example:

4-2 <= 2
2+2 = 4

Process	Max	Current	M=12
P0	10	5	
P1	4	2	
P2	9	3	
		Left	2

Safe vs. unsafe state

The system makes sure to keep the resource allocation in a safe state. It grants a request only if the resulting state is safe, meaning that there is a way to allocate whatever is left to the processes as to execute them all to completion.

Example:

?

Process	Max	Current	M=12
P0	10	5	
	4		
P2	9	3	
		Left	4

Deadlock Detection

You see a stale configuration of processing, each of them waiting for some resource, and each of them holding some resource. You ask yourself "are they deadlocked or just temporarily stuck on something"?

n	# of processes that appear stuck
m	# of resource types
Av[j]	# of resource units of type j still available
Al[i,j]	current allocation of resource j to process i
Rq[i,j]	# of units of resource j requested by i

Difference between this and Banker's

While this looks similar to Banker's algorithm there are subtle differences that allows this this to simply check if the stuck processes can make any progress at all. This also doesn't require the processes to declare anything in advance. We just want to check if the stuck process can make progress at all.

Deadlock Detection Algorithm

```
bool deadlock () {
  int W[m], i, j; bool F[n], alldone;
  for (j=0; j<m; j++) W[j]=Av[j];
  for (i=0; i<n; i++) F[i]=false;
Restart:
  alldone = true;
    for (i=0; i<n; i++) {
      if (!F[i]){
        for (j=0; j<m; j++)
          // replace N[i,j] with Rq[i,j]
          if (Rq[i,j] > W[j]) break;
        if (j == m) {
          for (j=0; j < m; j++) W[j] += Al[i,j
    ];
          F[i] = true;
          goto Restart;
        }
      alldone = false;
      }
    }
  return !alldone; // negate this from safe
}
```

Is deadlock detection important?

The popular realistic cases of deadlock result practically exclusively from synchronization, i.e., acquiring multiple locks in the wrong order. Remember: always acquire multiple locks in the same order in all processes (or threads).
Sometimes you don't know whether the other processes follow your protocol. What then? You can force enforcement I don't know.

Another useful operation

```
// S->P(); + S->V(); S->T(); Try the semaphore
    , return immediately (true or false)

void getTwoLocks (semaphore *sem1, semaphore *
    sem2) {
  while (1) {
    sem1->P();
    if (sem2->T()) return;
    sem1->V();
    sem2->P();
    if (sem1->T()) return;
    sem2->V();
  }
}
```

8: Scheduling

In this chapter we will talk about different scheduling algorithms, how they work, their pros and cons.

An analogy can be made where the customers are the processes and the server is the entity that is tasked to handle or process the customer. This can be abstracted to CPU bursts which is handled by the CPU scheduler to be processed by the CPU, etc.

Basic Performance Criteria

Turnaround time: How much time it takes on average to handle a customer.
Throughput: How many customers can I handle per unit of time?
Customer satisfaction: Maximize the number of customers that are happy with my service.
There is a careful balance between turnaround time and throughput.

Short Term Scheduling

CPU bursts are the customers. We care about the turnaround time of CPU bursts.
SJF says that processes with short CPU bursts (i.e., I/O bound processes) should be given priority in acquiring the CPU.
Easy to say. How can we know what the duration of a process's next CPU burst is going to be?

Guessing the CPU burst

Exponential Moving Average

A standard way to calculate a snapshot dynamic average of (potentially) infinitely many samples, such that old samples are aged out is through this formula:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \tag{1}$$

Where:

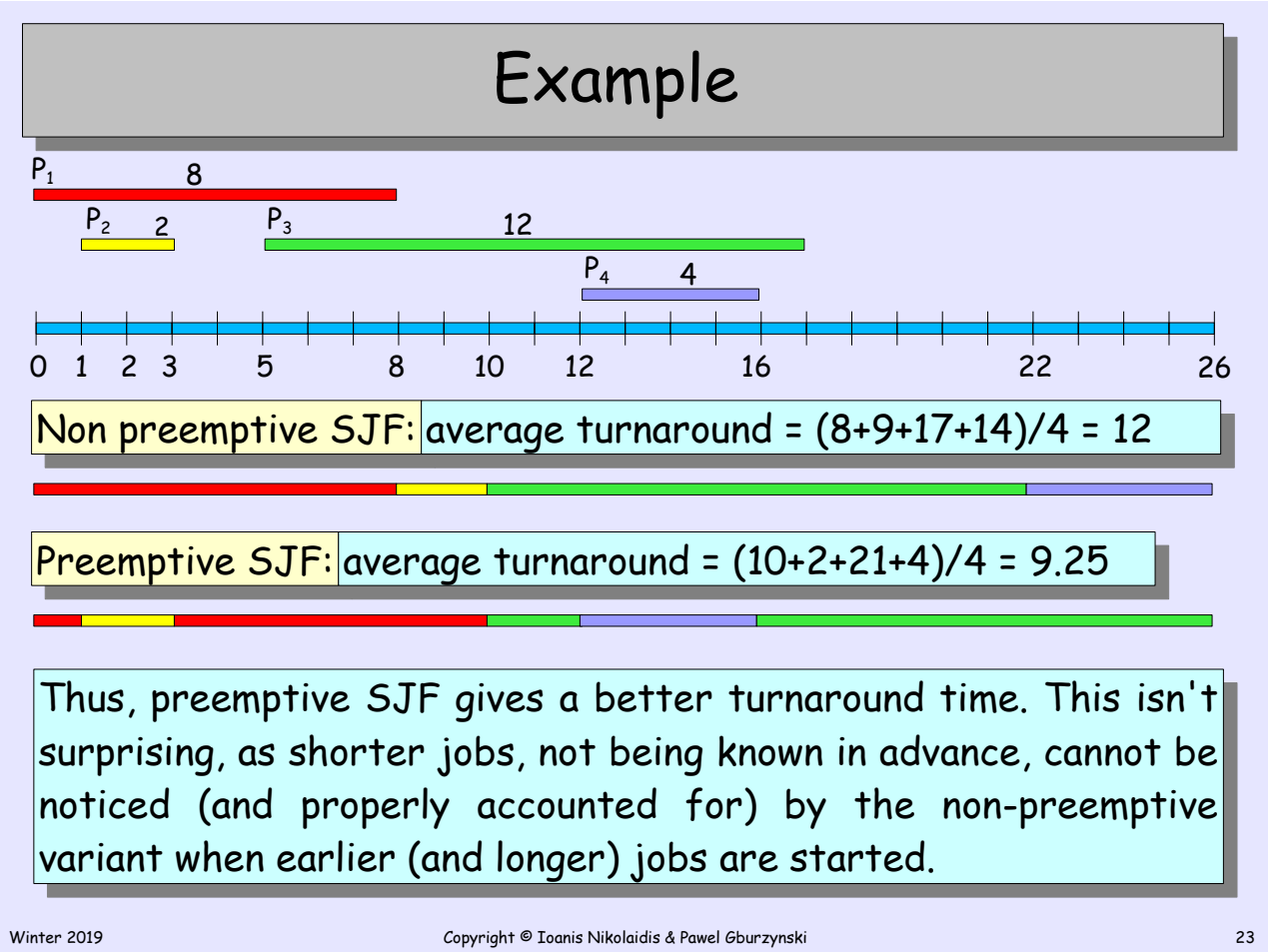
t_k is the k-th sample
τ_k is the k-th average
α is the recency coefficient (0-1] where a higher value means history matters very little and a lower value means history matters a lot.

SJF: Shortest Job First

Shortest Job First

Give priority to the customers with the shortest processing time. This policy guarantees the shortest possible average turnaround time.

Non-Preemptive vs Preemptive



The time taken for each process in both the non preemptive and preemptive SJF algorithm
$T(process)$ = time it actually finished − time it was suppose to start

Priority

Generally, scheduling schemes may be based on the notion of priority associated with a process.

Priority for SJF

$$P = \frac{1}{\tau} \tag{2}$$

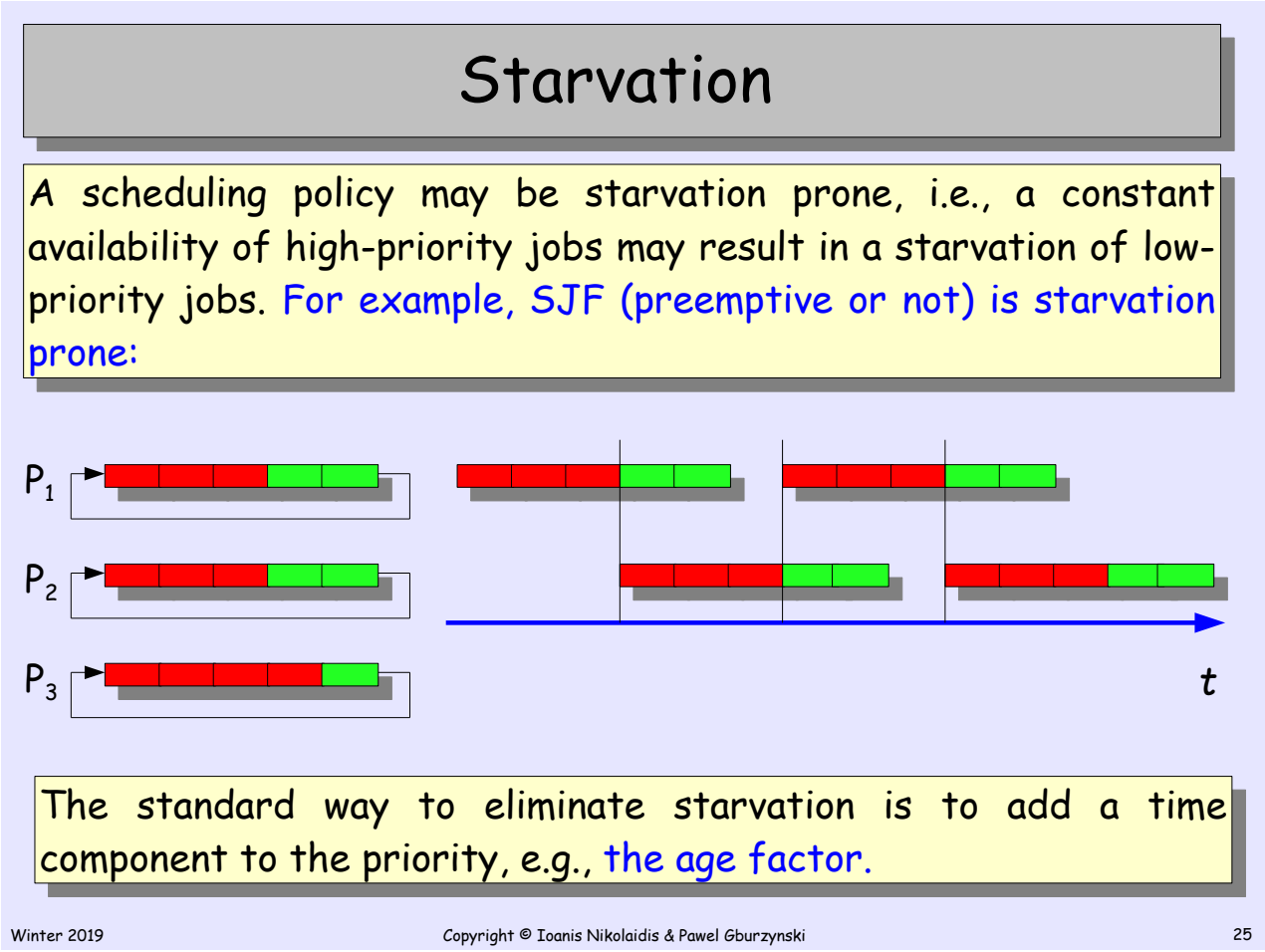
Where τ is the predicted duration of the next CPU burst.

Other factors

There could be other factors that affect the priority of the process such as internal factors like CPU burst length, memory size, objective importance of the process, etc or external factors who's attributes are assigned by humans.

Starvation

The problem with both SJF algorithms is that they are starvation prone. The current iteration of the algorithm does not allow other "low" priority processes to be executed.



In the diagram the red sections are CPU operations while the green sections are I/O operations which are blocking the process.

Round Robin

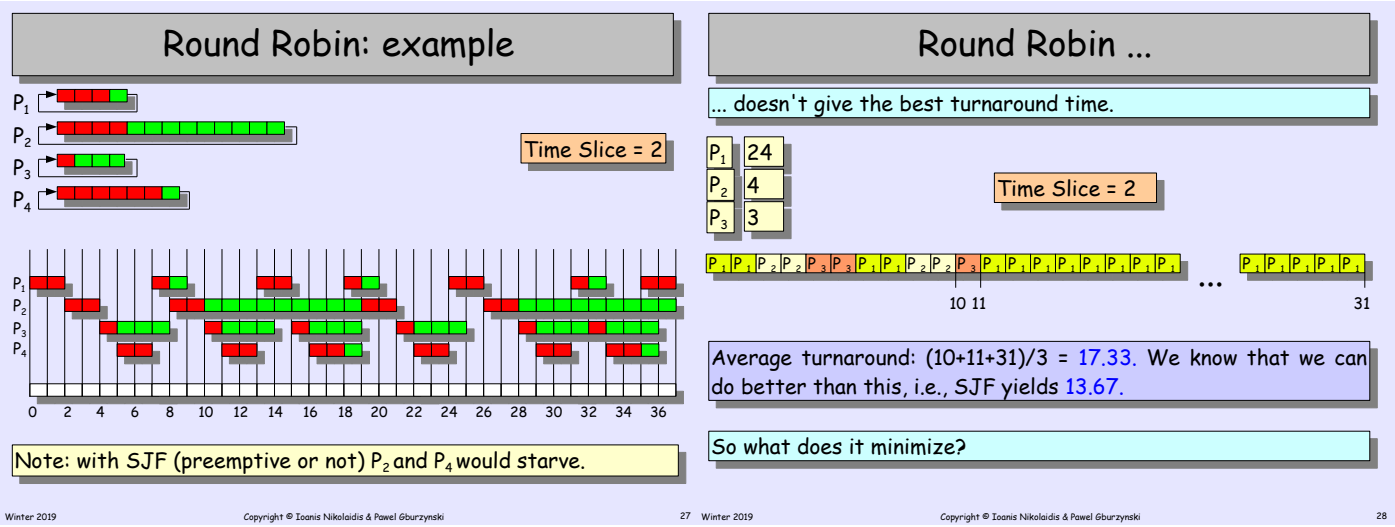
Average turnaround time is not the only performance criterion of interest. For an interactive job (with a human user sitting in front of a terminal), the primary criterion is the (expected) waiting time for the program's reaction.

Time slice

A process can lose the CPU in two ways:

- It hits an I/O burst, i.e., its CPU burst runs out naturally.
- Its time slice expires.

The time slice can be the same for all; also, some processes may be getting a larger share of the CPU âĂŞ reflecting their higher importance



Round robins guarantees that if a process "has something to say," it won't have to wait for more than $(N-1) \times S$ where N is the number of processes and S is the time slice. Theoretically, the best response time is achieved when S is the shortest possible, i.e., single instruction time. Practically, the context switch overhead imposes a limit on how small S can be. Typically, S is between a millisecond and a tenth a second.

FPPS: Fixed Priority Preemptive Scheduling

What is the best scheduling policy for high priority, extremely I/O bound processes? Such processes are called "**reactive**"

FPPS

Processes are rigidly ordered in some fashion. The first process that is ready gets the CPU

We know for a fact that the CPU bursts are always short. There is no need to guess or estimate. Therefore, fixed (high) priority is OK. Also, the exact ordering of those processes (their actual priority) doesn't really matter, as long as they are ahead of everything else.
This type of scheduling is particularly important for kernel processes.
To perform actions that cannot be properly performed from an interrupt service routine. For example:

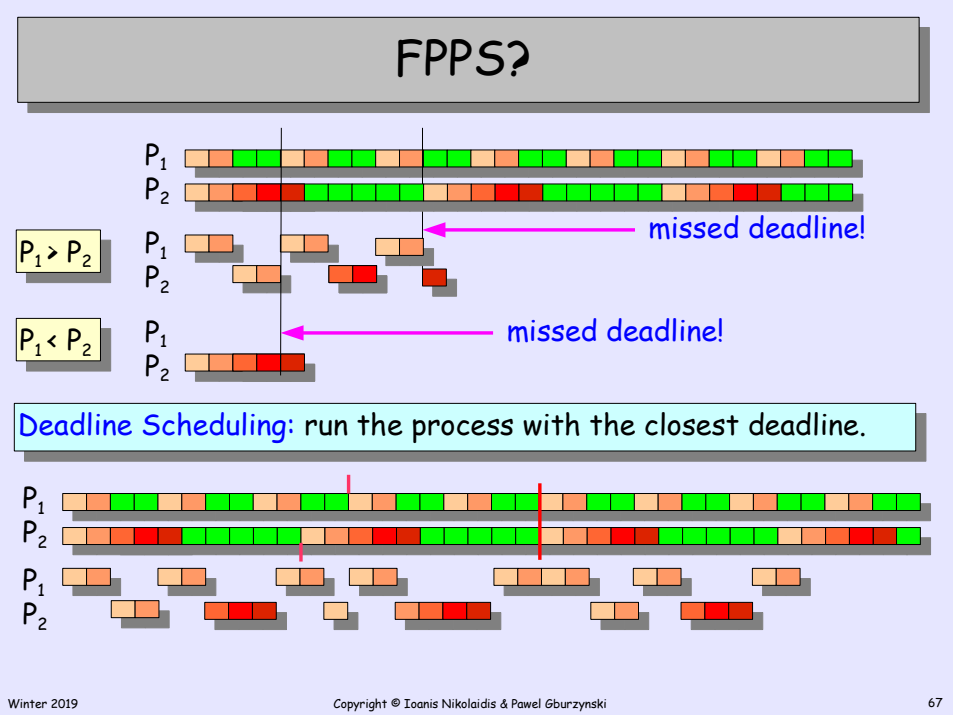
- something that does I/O on its own and looks like a thread (paging-related daemons, file-system components, especially for networked file systems)
- something that contains complicated actions of lower priority than interrupts; while those actions are triggered by interrupts, it makes better sense to carry them out elsewhere.

Note that interrupts may block other interrupts, and thus impair then system's responsiveness, if they are "too heavy".

In the latter case, if the action looks like a single burst, it need not be implemented as a single process. Therefore, during an interrupt service where it would return to User-land it will run the bottom halves of the lower priority actions.

Hard deadlines

In some (RT) applications, average performance is irrelevant. What matters is the obedience of hard deadlines. A process may have a certain amount of time to complete it work and then slack off, give the CPU to someone else. However, they must finish their work before its next ready time which is set at a certain time interval. Note different processes could have deadlines where they need to be completed by.



Regardless of priority both processes will miss their deadline for completion.
If P1 must be completed as soon as it becomes available then P2 will miss their deadline as the scheduler is giving P1 too much time. Likewise if P2 must be completed as soon as it becomes available then P1 will their deadline for the same reason.

To fix this issue the scheduler must complete the process that is closest to its deadline.
In this case instead of P1 starting its third round the scheduler allows P2 to finish as its deadline was closer then P1

Multi-mode systems/Multiple policies

Having multiple policies

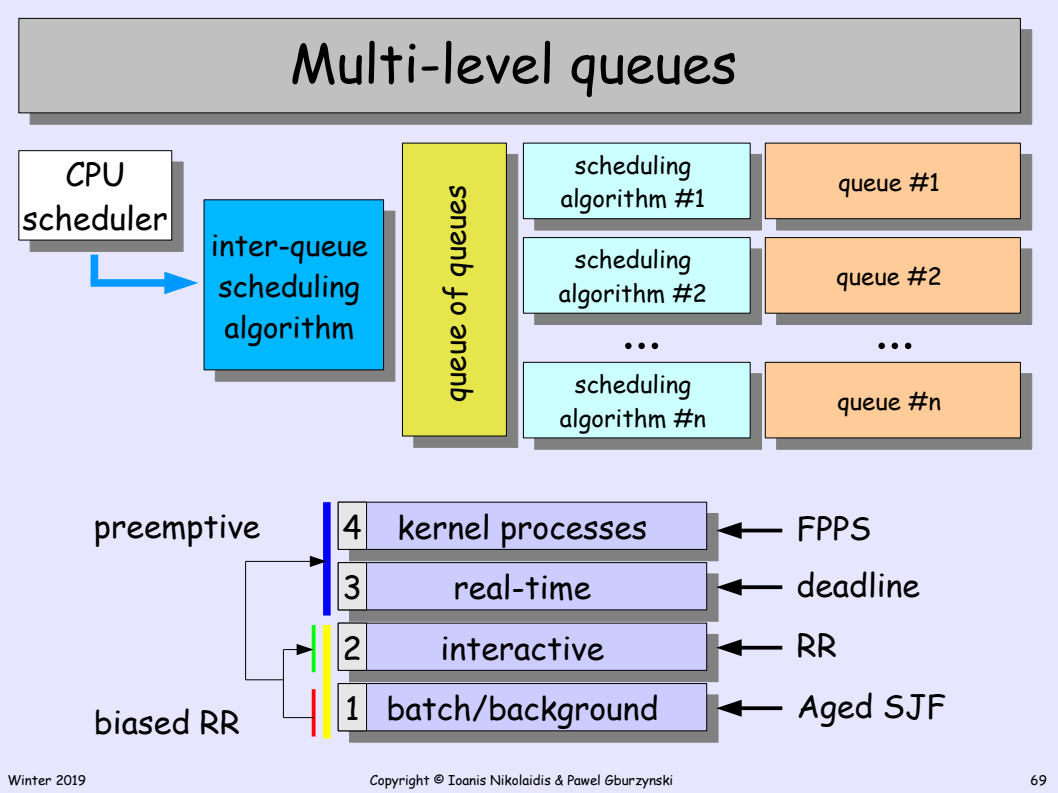
SJF: Batch (or background) jobs. Turnaround time is the measure that matters. Aging may be added to avoid starvation.

Round Robin: Interactive jobs. Response time is the objective.

FPPS: kernel jobs. They should have the highest priority, and their relative importance doesn;'t really matter.

Deadline Scheduling: Real-time jobs with hard constraints. Obedience of deadlines is the objective.

Multi-level queues

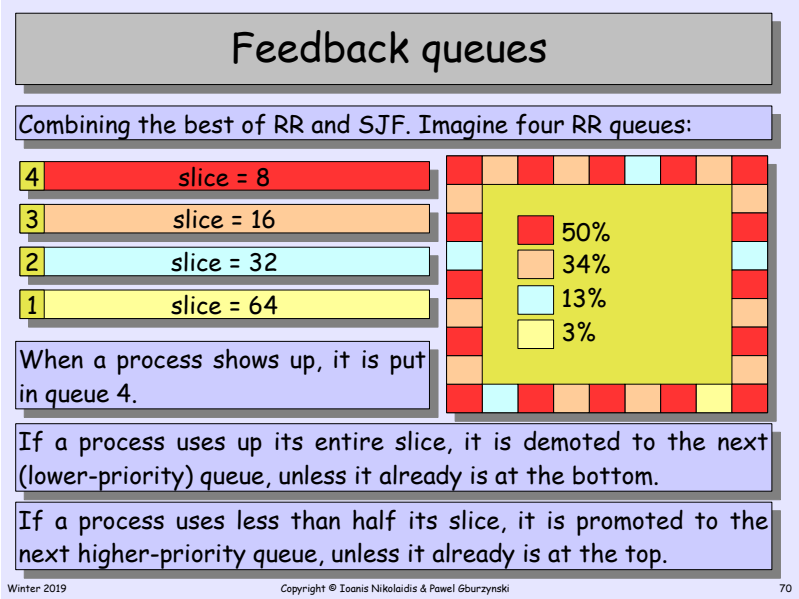


Different tasks based on priority have different scheduling policies.

Kernel processes have the highest priority, they get all the cpu time. Then real-time, followed by interactive, and finally batch/background processes have the lowest priority compared to the rest.

Some batch/background processes can be interactive depending on the workload.

Kernel processes cannot be controlled by the user. They are fixed by the developer. The rest can be controlled at some level.



A single policy that is a combination of two policies we discussed before.

Processes that uses less or equal to the smallest time slice will be called more often achieving the benefits of a SJF policy. But to prevent starvation there are round robin queues that have larger time slices but are called less often for processes that take up more cpu time. This trickle down to larger queues with smaller frequencies.

Real-time vs Standard System

Standard system	
Capacity	high throughput, statistically good utilization of all equipment and low overhead
Responsiveness	low average turnaround time, low average response time
Overload	fair performance degradation
Real-time system	
Capacity	"scheduleability," number of context switches per second, number of interrupts per second
Responsiveness	worst-case delay, low variance, deterministic behavior
Overload	stability, unaffected performance for critical tasks

Primary sources of problems

- **Scheduling policies**, e.g., aging introduces non-determinism, rigid priorities are starvation-prone, deadline scheduling requires extra specification and is not for everybody.
- **Context switch**, e.g., extra actions upon context switch may reduce capacity and increase non-determinism
- **Synchronization tools and paradigms**, e.g., critical sections in the kernel may impair scheduleability (context switches and interrupts).
- **Memory management**, e.g., sophisticated contemporary techniques of memory mapping introduce a lot of non-determinism.

Synchronization tools and paradigms are a problem in real-time systems as it loses determinism. The process that gets the lock are not known until it happens. Same with memory management, it abstracts memory management at the cost of having non-determinism.

Examples of Non Real-time paradigms

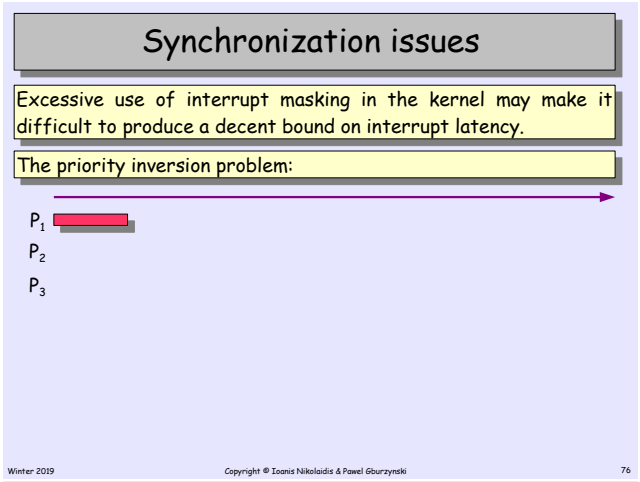
- **Non-preemptibility of processes in the Kernel-land (many standard UNIX systems work this way):** Processes cannot be rescheduled until they leave the Kernel-land. Usually, they spend very little time in there, but who can tell the bound?
- **Virtual memory, including lazy loading, also DLLs:** A program running happily at full speed may suddenly have to wait for I/O for no good reason
- **Disk inodes for representing devices (as on UNIX systems):** Access to an otherwise extremely fast device may stall because its inode has disappeared from the in-memory cache

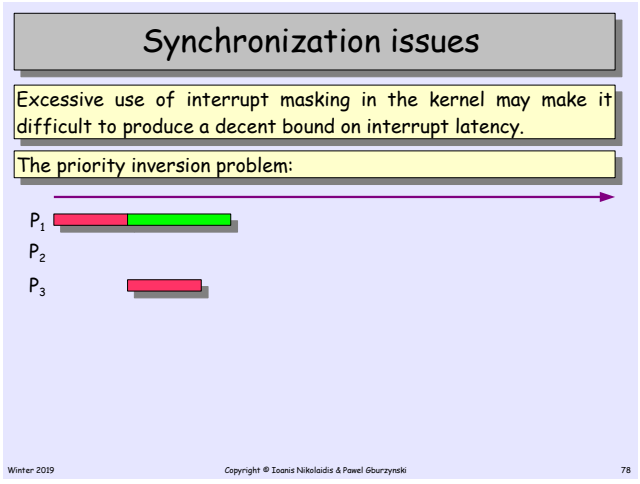
Sync Issues

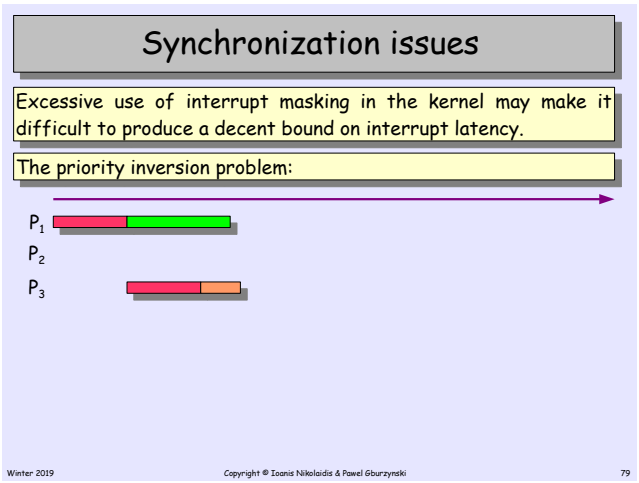
Priority Inversion

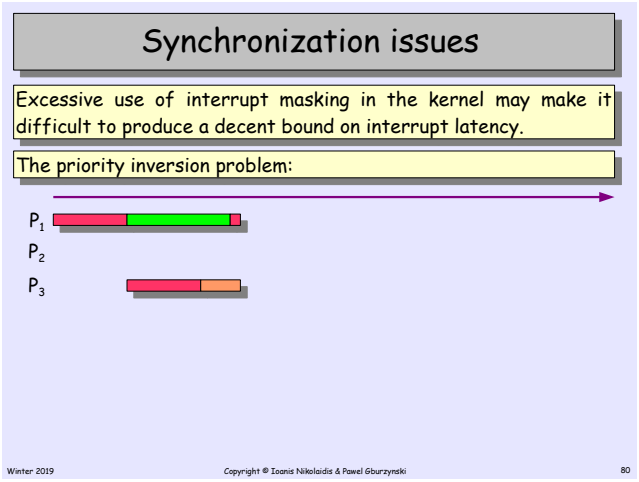
LEGEND

red - using the cpu (cpu burst)
orange - process in a critical section
green - I/O op or whatever.

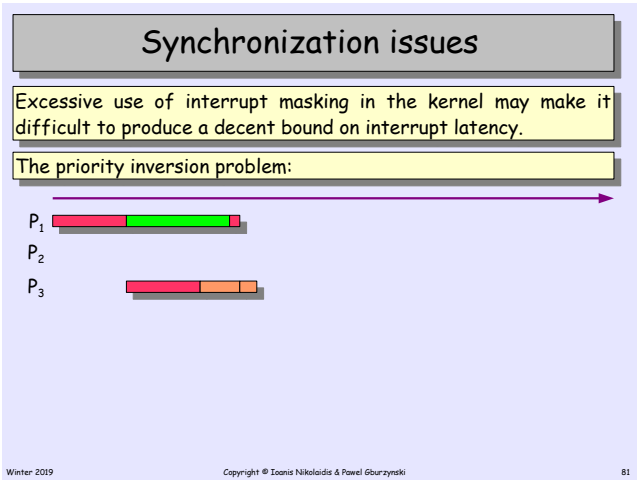




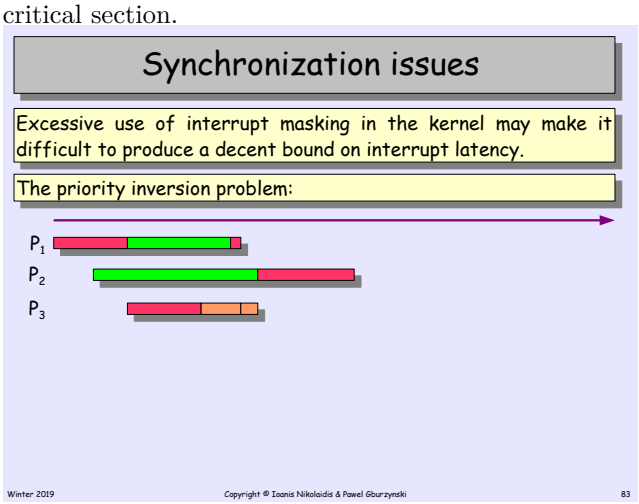




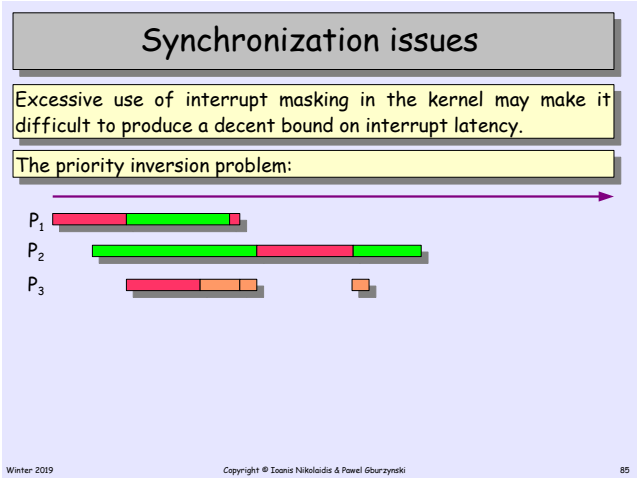
P1 has priority over P2 and P3 and tries to gain access to the critical section but can't as P3 has the lock.



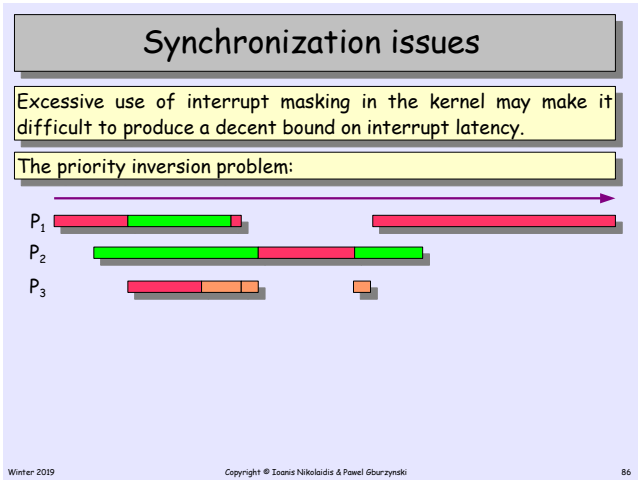
P3 continues with P1 waiting to gain access to the critical section.



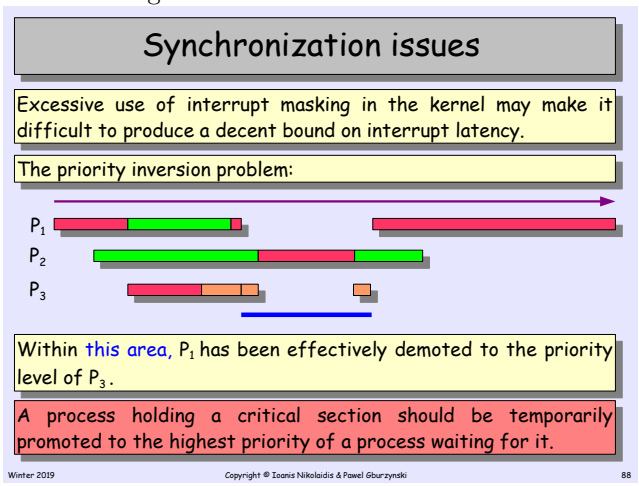
P2 is introduced. Since P2 has a higher priority then P3 it blocks P3 and uses its cpu burst.



P2 finishes its CPU burst and allows P3 to execute, but P3 only needed a little extra time to finish its ownership of the critical section.



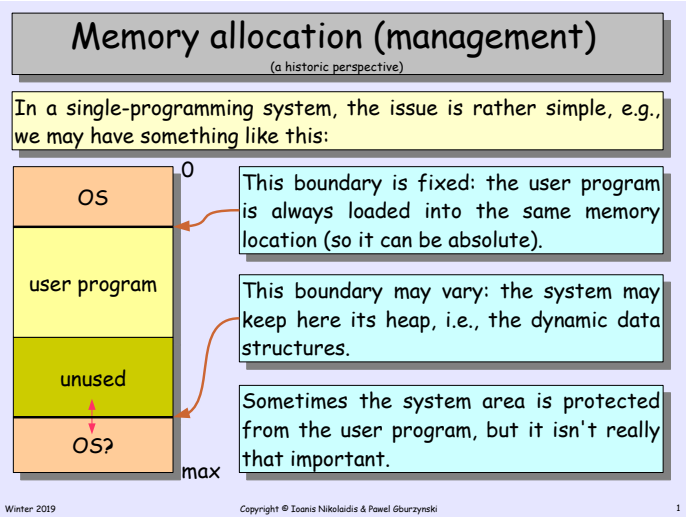
P1 can now get access to the critical section.



9: Memory Management

There are different levels of abstractions regarding memory.
There is the physical, logical, and the virtual. The last two are the process's view of memory.

Memory Allocation (management)

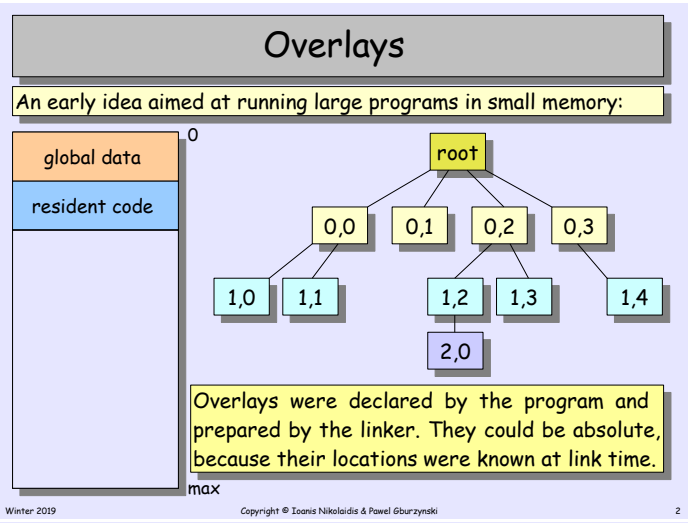


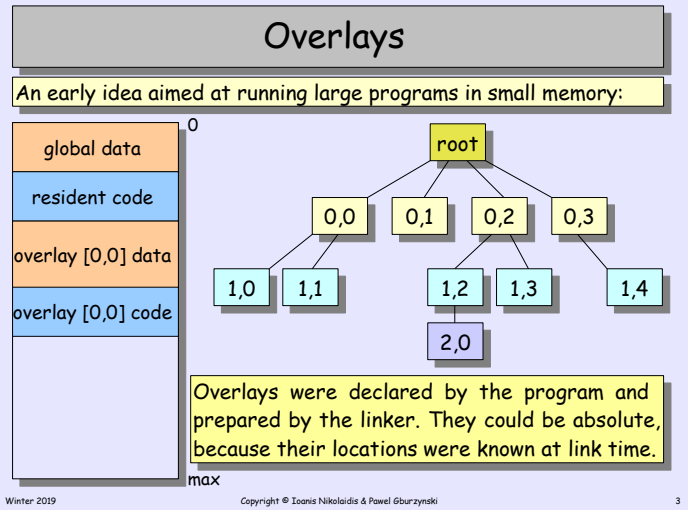
Overlays

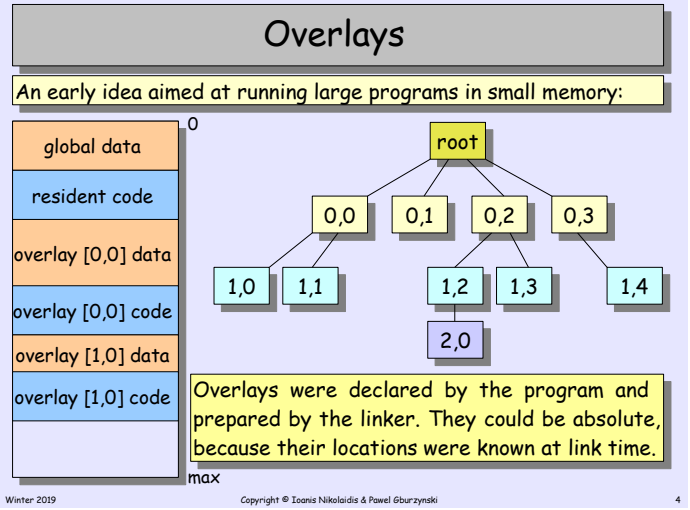
One of the first ways to manage memory is thought overlays. The program is partition by the programmer where he or she can override certain parts of memory if that portion is not being used. Similar to scoping.

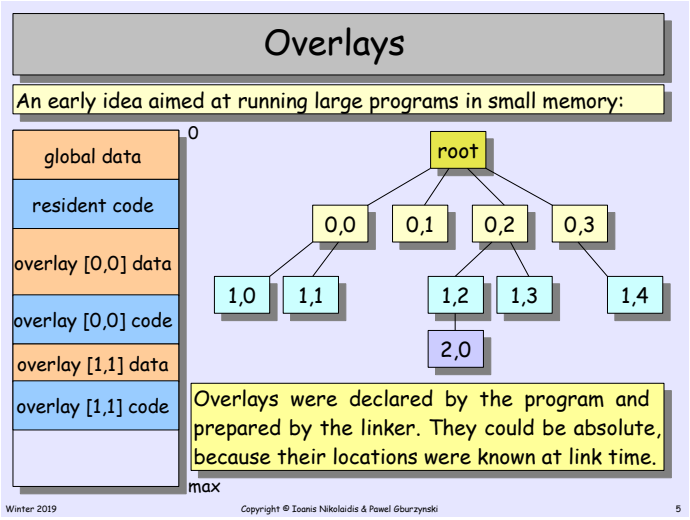
The actual overlaying is called by the programmer in the parent overlaying node.

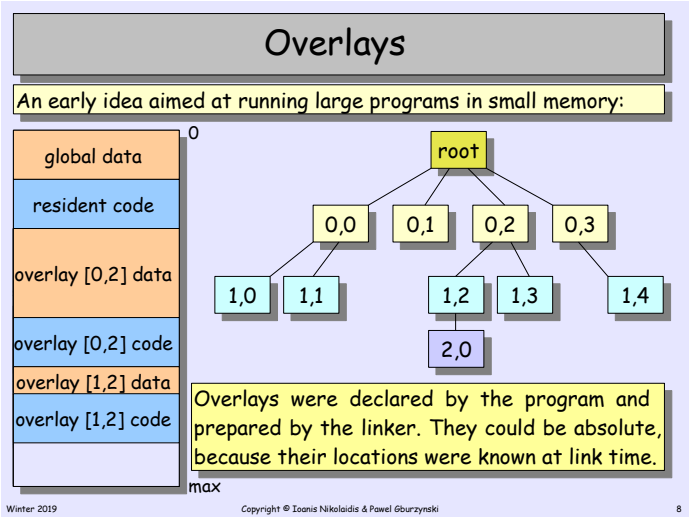
The overlay is called in a DFS like fashion, and this overlaying process was not automated.

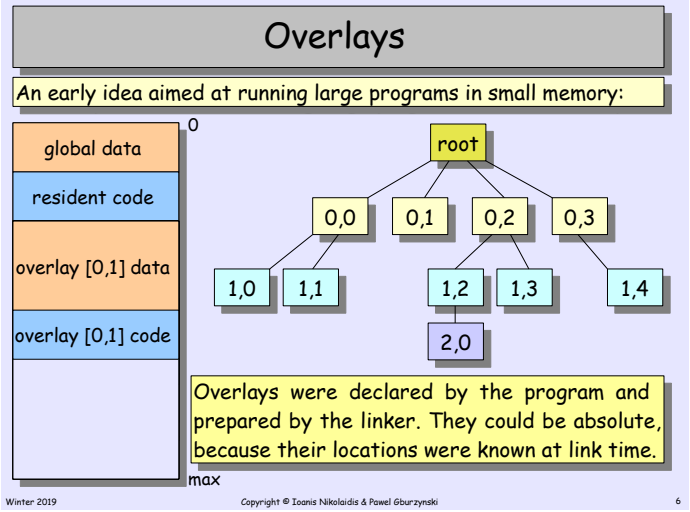


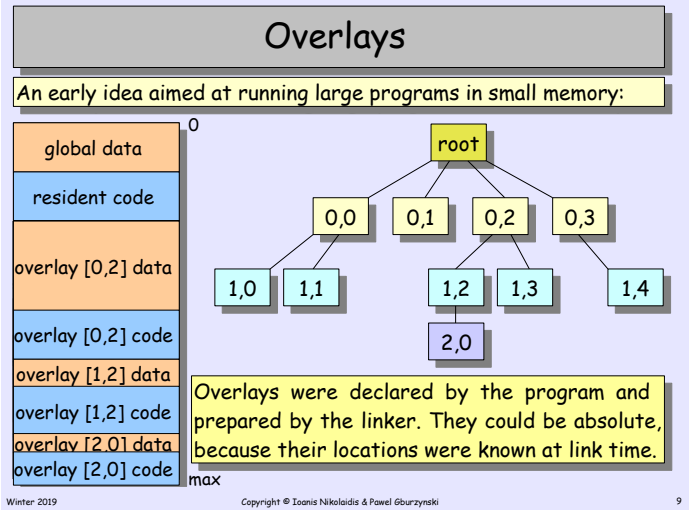


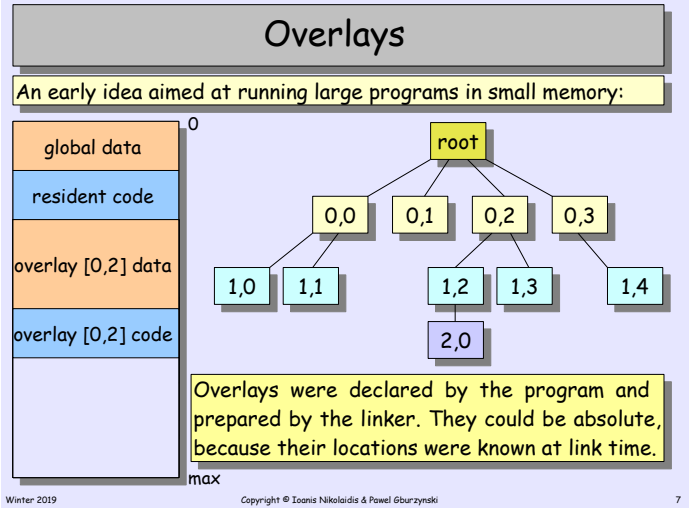












etc, you get the idea.

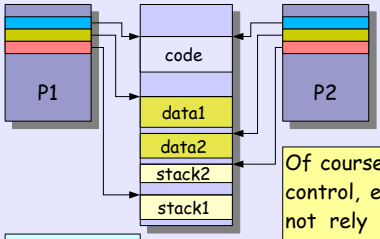
Memory layouts of early programs

Memory layout during the early days of computing were rather simple. It wasn't obvious that stacks are generally useful and that code and data should be separated. The prevailing programming language were FORTRAN for efficiency and COBOL for convenience. They were both static, neither allowed for recursion officially. Consequently, nobody cared about a built-in stack. Few of the popular CPU architectures supported it. Memory was viewed as a single, continuous, mostly static, chunk.

A digression

Q: What is the single most important advantage of separating code from data?

A: Reentrancy, i.e., the ability to share code among multiple processes.



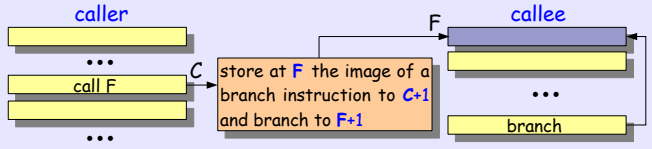
Prerequisites: 1) being able to enforce the read-only status of the code area, 2) having separate pointers to code and data.

Of course, the mechanics of program control, e.g., subroutine calls, should not rely on storing anything in the code area.

Recall **fork** !

Example

Hardware mechanism for subroutine call on CDC 6000 machines:



The code image of the called function is modified.

There is a single static place (per function) where the return address is stored. Thus, there is no way to call the function recursively, unless you implement your own stack by hand.

All memory references issued by a program went to a single preallocated chunk of physical memory, so it wasn't easy to share.

This basically tells you that stacks are very useful.

Sometimes one misses the old days...

One reason for modifying code on the fly was efficiency. Its legitimacy can be disputed, but ...

```
void crunch (int N, double *M) {
    for (int i = 0; i < N; i++)
        if (some_global_option)
            M [i] = 1.0/M [i];
        else
            M [i] = M [i]/M_PI;
}
```

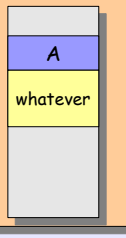
```
void crunch (int N, double *M) {
    for (int i = 0; i < N; i++)
            M [i] = 1.0/M [i];
}
```

Such modifications were popular among library functions that would auto-tune themselves to global options, hardware parameters, or self-measured perceived performance.

A digression: Trampolines

Even these days the purists have to yield sometimes. Recall Lazy Linkage. In some cases, a dynamically created function (typically put onto and run from the stack) is demonstrably the most efficient way to accomplish something tricky.

```
int A (int u, int v) {
    int c;
    int B (int w) {
        return w + c;
    };
    ...
    return B (v);
}
```

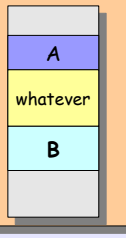


B is a nested function (such functions are legal in modern C (like gcc, for example). Note that, when called, it must know where A's local variables are.

A digression: Trampolines

Even these days the purists have to yield sometimes. Recall Lazy Linkage. In some cases, a dynamically created function (typically put onto and run from the stack) is demonstrably the most efficient way to accomplish something tricky.

```
int A (int u, int v) {
    int c;
    int B (int w) {
        return w + c;
    };
    ...
    return B (v);
}
```

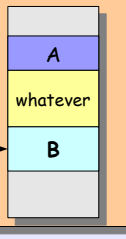


B is a nested function (such functions are legal in modern C (like gcc, for example). Note that, when called, it must know where A's local variables are.

A digression: Trampolines

Even these days the purists have to yield sometimes. Recall Lazy Linkage. In some cases, a dynamically created function (typically put onto and run from the stack) is demonstrably the most efficient way to accomplish something tricky.

```
int A (int u, int v) {
    int c;
    int B (int w) {
        return w + c;
    };
    ...
    return B (v);
}
```

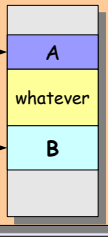


B is a nested function (such functions are legal in modern C (like gcc, for example). Note that, when called, it must know where A's local variables are.

A digression: Trampolines

Even these days the purists have to yield sometimes. Recall Lazy Linkage. In some cases, a dynamically created function (typically put onto and run from the stack) is demonstrably the most efficient way to accomplish something tricky.

```
int A (int u, int v) {
    int c;
    int B (int w) {
        return w + c;
    };
    ...
    return B (v);
}
```



B is a nested function (such functions are legal in modern C (like gcc, for example). Note that, when called, it must know where A's local variables are.

Everything is fine until we want to have pointers to such functions.

Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 24

Look at this:

```
int A (int u, int v) {
    int c;
    int B (int w) {
        return w + c;
    };
    int (*fp) (int);
    fp = B;
    c = u;
    return C (fp, v);
}
```

```
int C (int (*fp) (int), int q) {
    return (*fp) (q);
}
```

B is blind without an extra pointer that tells it where c can be found. Thus, it needs such a pointer whenever run. So how to represent pointers to functions like B? Note that the same issue concerns pointers to member functions in C++.

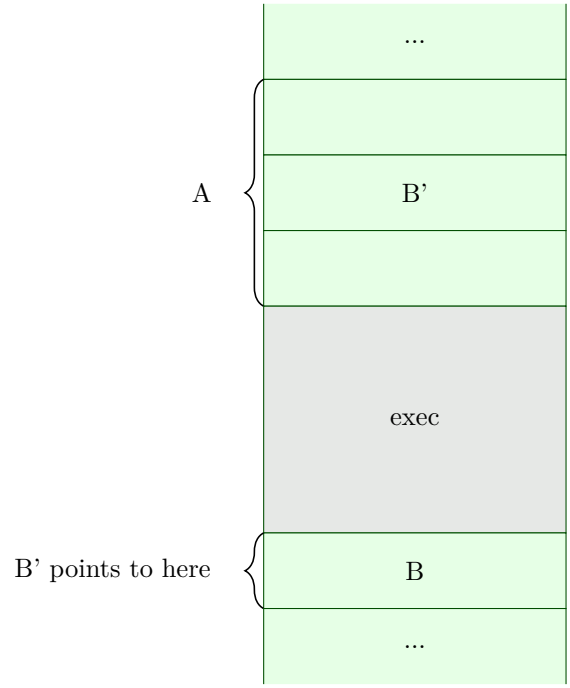
But note this: in whatever context B is legitimately called, its A must be present on the stack. The pointer to A's stack frame is a "momentary constant" of that context.

Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 25

The Most efficient way out is to have on A's stack a trivial chunk of code. This is what it does:

1. Load the pointer to A's frame into the place (a register) when the called function expects it
2. Jump to the function according to its bare pointer (provided in some register)

Such chunks of code (called trampolines or thunks) are used in some other places, notably, for invoking signal handlers.

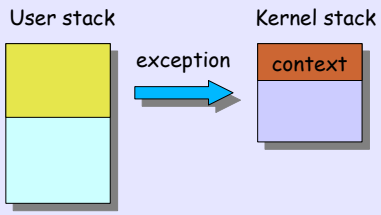


B' is responsible for executing B on the stack. Moreover, it is responsible for executing the jump and the passing of the function pointer to B.

Signal Handlers

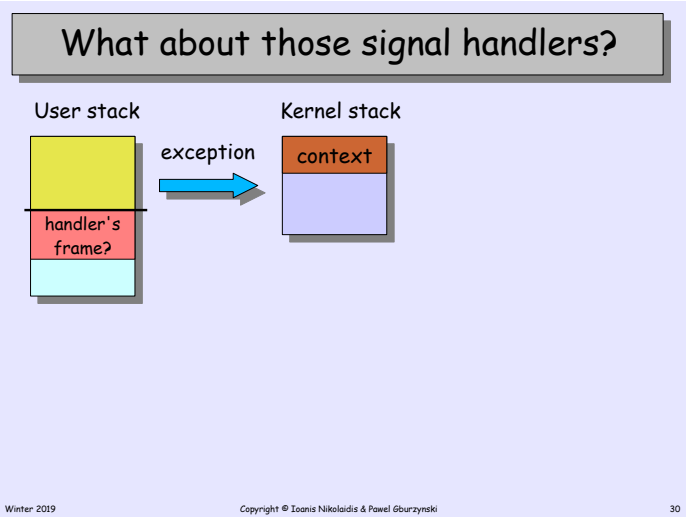
Because signal handles are executed in the context of the receiving process started by the schedule.

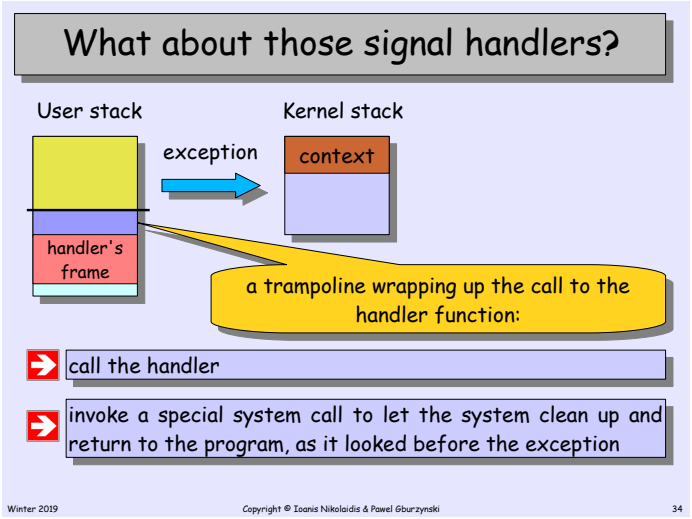
What about those signal handlers?



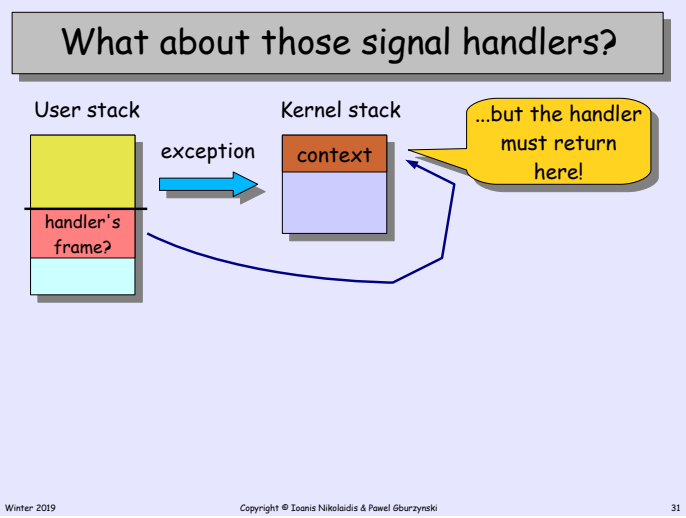
Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 28

Once the TS expiration is reached to the process. The important contents of the process is stored into the kernel's stack. Things like PC, things that are needed to resume.





The kernel needs to create an activation frame for the signal handler in the user stack, as the user does not have control.



The trampoline contains executable codes that execute system calls that tells the kernel that the receiving process did receive the signal. It also chain multiple signal handlers together and call the return signal system call to the kernel to tell that the current process is finished handling system calls.

The Goal of Memory Allocation schemes

The main goal of memory allocation is:

- **Efficiency and flexibility combined with the full dynamics of multi-programming.** Ideally, if there is free space to run a new program, it should be able to run it regardless of the condition of memory.
- **Separation of concerns** The system deals with the physical memory, while the program deals with the logical memory, the process only sees and deals with the logical memory.
- **Protection coexisting with collaboration opportunities.** collaboration like sharing should only be done in controlled and well defined environments.

Fixed Partition

At the hardware level, memory is not uniform.

But at the software level, memory should be uniform as it's easier to work with.

Uniform model with addressing, but we specialize regions into segments. The general idea is that these segments which are continuous, can be placed anywhere in any order within memory.

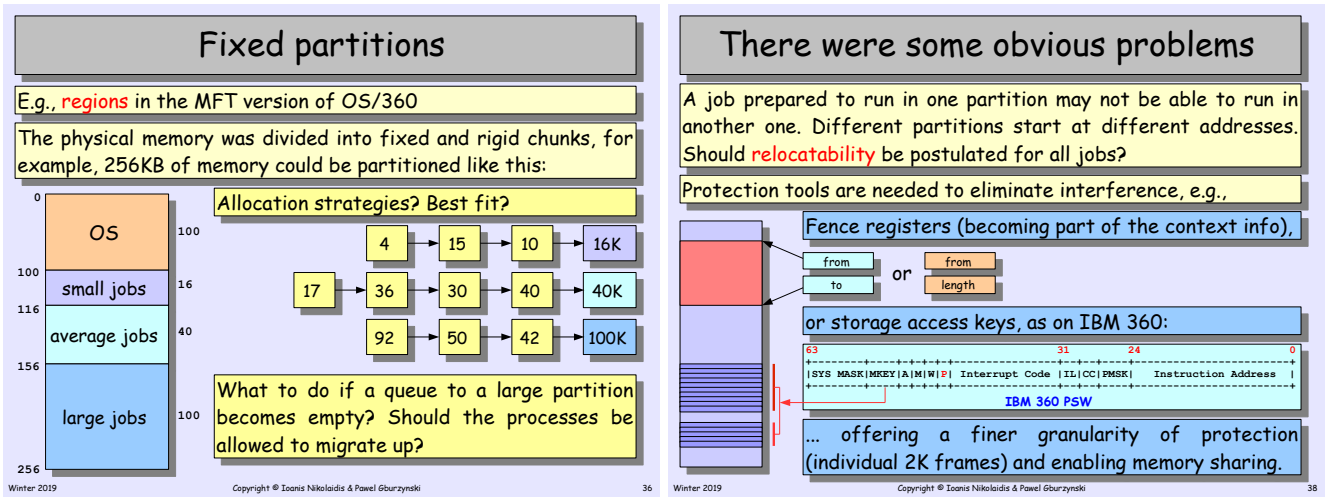


Figure 3: Fixed partitions and its issues

These types of memory allocation were used in the 60s. Programming languages during these time didn't support recursion or other run-time dependent factors that affects execution size. So the size of the execution of the program can be determined during compile time.

The problem with this type of memory allocation is that only a fixed number of processes can be running at any given time.

Fence registers are used as a way to prevent illegal memory access of the current process. These registers are part of the the context info and are known to the CPU.

For the storage access key method each segment of memory have a key associated with the region. Each process will have a key to memory location where it have access. Mismatched keys are considered illegal access. Catch all keys or all access keys are also valid

Fragmentation

Types of fragmentation

The partitioning of memory into fixed sizes poses two problems.

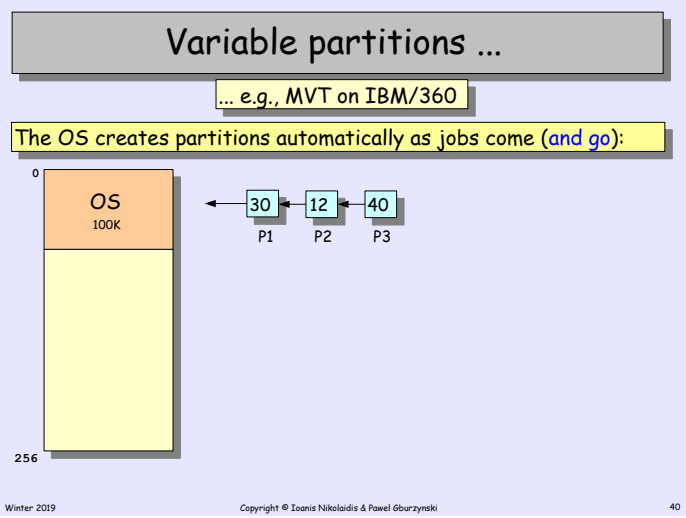
- **Internal Fragmentation**: the unused leftovers of partitions
- **External Fragmentation**: the fact that the partitions exist at all. i.e. the program is bigger than the partition even though you have enough physical memory.

What we are trying to solve is the elimination of fixed size partitions using dynamic partitioning of memory. What we want:

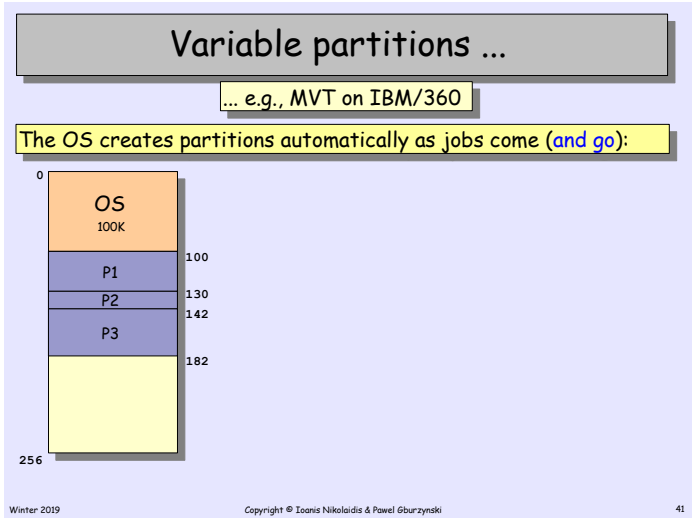
- Flexibility with the size of code.
- relocatability (seen this from position independent code)
- protection
- sharing (controlled)

All of this has to be done in a space (no waste) and time (with CPU support "in real time", and "instantly") efficient manner.

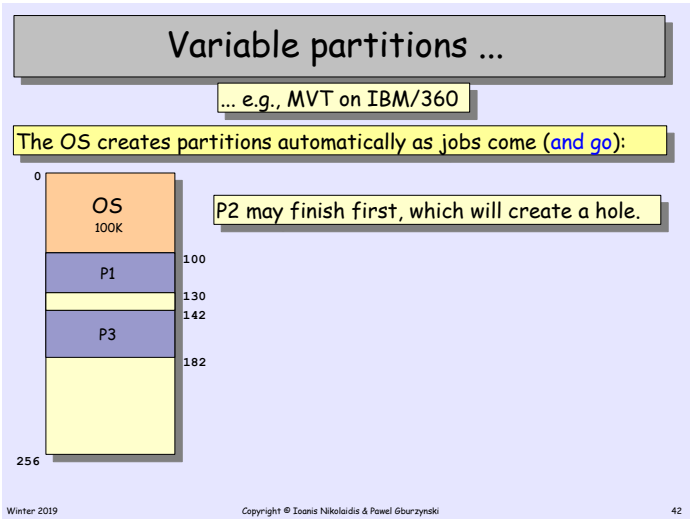
Variable partition



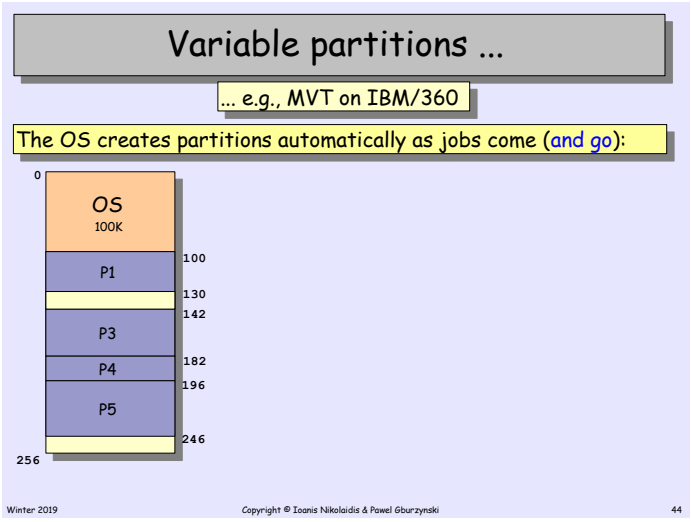
Processes are coming in



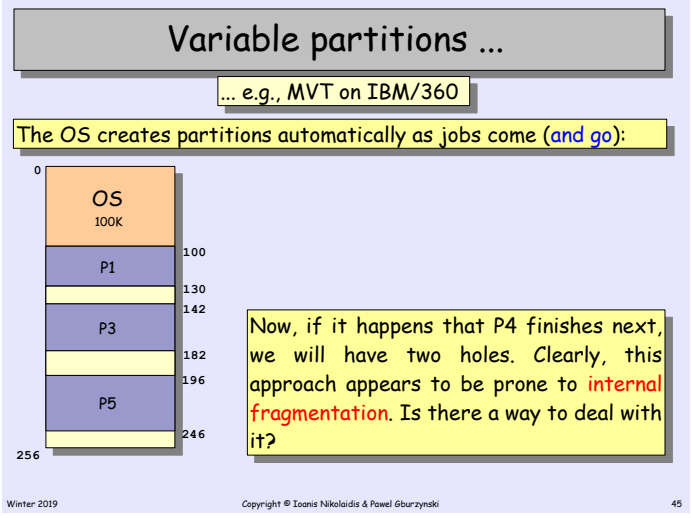
They are loaded into memory



P2 is done and is removed from memory.



More are loaded



Gap filling strategies

- **Best Fit:** The program is put into the tightest hole that can accommodate it. Trying to make the smallest hole possible.
- **Worst Fit:** The program is put into the largest hole.
- **First Fit:** Just put it into the first available spot.

Best fit vs worst fit

The logic behind worst fit is that after fitting in the program into memory you are likely to have enough space for another program. While best fit is more likely to have wasted space. First fit is a don't care policy.

Compaction

Can we move programs in memory to reclaim a single continuous free chunk?

There is no trivial solution to this problem as very few programs are actually movable, **Recall: out discussion between relocatability, position independence, and movability.**

This leads to the idea of abstracting physical address to logical addresses.

CDC 6000 Example

The CDC 6000 have two special (protected) registers **RA and FL**. The two registers are reloaded as part of the context switch whenever the CPU is assigned to a process. This process does add a bit of complexity but accomplishes one important goal: we can begin to differentiate between logical and physical addresses in memory.

Here is the hardware implementation in C.
The logical addresses of CDC 6000 starts from 0 - MAX.

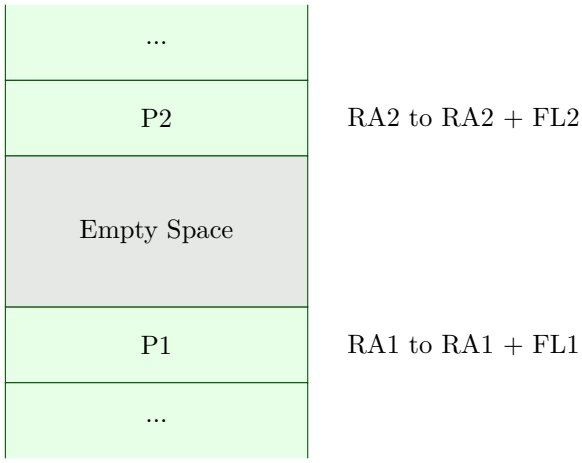
```
1       // checks if the logical address is
    within the program
2       // p is the phsyical address
3       // RA (relative address) is the
    phsyical address where the process being
    stored in memory begins (starts from 0)
4       // FL is the length of the address
5       if (i < FL){
6           p = RA + i;
7           }
8
```

Memory Sharing

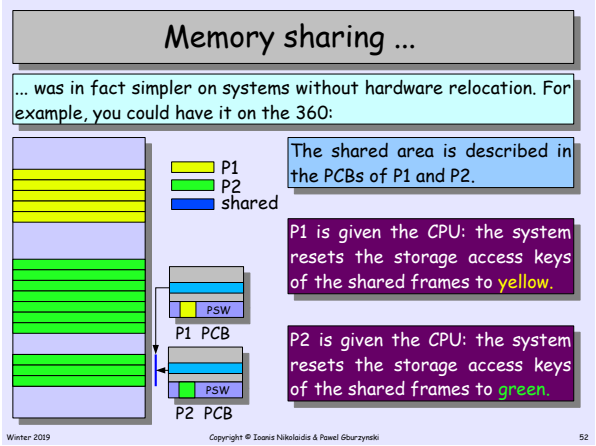
Due to the memory protections provided by the CPU there is no way to share memory between different processes. You can emulate it using system calls and pass messages around, but that involves passing things that shouldn't be done by the kernel in terms of responsiblities. One can try to overlap but that leads to a mess.

Another problem is that everything in the process is writeable. There is no protection within the process itself.

Below is the memory representation of processes



Memory Sharing



On the IBM 360 allows sharing based on an ownership key. Moreover, there was a global ownership key where all processes can gain access.

Swapping

Turning memory into a preemptible resource. Programs can be suspended and swapped out, then swapped in later. This is useful particularly with movable partitions. This allows you to have a pool of programs in execution be larger than the physical size of memory. This does lead to another level of schedulling where the program in swap needs to compete to get loaded into memory.

This allows the system to acconmodate important programs without having to kill any of those currently in execution.

When to swap out?

- The program has exceeded its inter-swap interval, and there are other programs waiting to be swapped in (fair processing)
- The program has reached a stage where it has to wait for a longish time. Not necessary an I/O request, but "truly external". This is because a program's memory can become a non-preemptible resource.
- A high priority program requires immediate execution and there is no other way to acconmodate it.
- With some partition systems, like one with compactions, a program can grow or shrink. When a program grows too much it may need to be swapped out until a more favorable situation arise.

Paging

Pretty much every OSes that is worth taking about have paging as a feature in memory management.

Frames and Pages

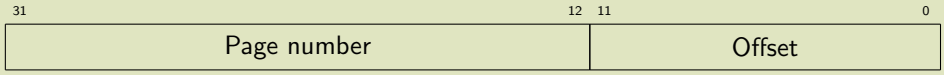
Frames are physical memory that is divided into fixed-size blocks. The logical address space of a program is divided into chunks of the same size and are called **pages**.

There is a mechanism for mapping any page into any frame. Some parts of the logical space may be unmapped (holes are OK).

Typical page sizes	
Intel x86	4096 bytes
MC68451	256 bytes minimum
IBM-370	2048 or 4096 bytes

The page size is always a power of two. This makes the interpretation of a memeory address easier. Below is the format for memory access in a 32-bit system.

Logical Memory Address



The **page number** is the offset from the head of the page table to get the correct entry for the frame number.
The **offset** is the offset within the page/frame.

Page Table Entry

The **frame number** is the frame index in physical memory.
The **attributes** tells the status of the frame. i.e., is it **valid** (is the page mapped at all), **accessible** (R/W, R/O, X/O)

The mapping of pages into frames is described in a speical data structure called the **page table**.
When memory is being referenced the page number is transformed into a frame number (the phsyical address) by the page table. This operation is done in hardware, essentially at every memory reference (at least in principle).
The page table is part of the process context where it describes the complete memory layout of the process.

The **Page Table Pointer** pointing to the head of the page table. They are stored in a CPU register for the current process that is being executed.

The benefits of this are: **No external fragmentation, very little internal fragmentation. Logical address space is separated from physical memory, natural protection, and easy memory sharing**.

Caching

To cache access to the memory we use the Translation Lookaside Buffer. Once a page is accessed the entry is copied into the TLB so lookup is faster. During a context switch, we need to flush the buffer as the values are garbage, only valid for the process that was using it.

TLB

The entry in the TLB is the frame pointer to the physical memory as we want the physcial translation. This is becuase we want locality, in general the next instruction is going to be near the frame stored in the TLB.

Address Translation

There are situations where the system must operate in physical memory. The actual translation of the address can be switch on or off by the system by changing a certain bit in a protected register.
Reason why?

- Accessing speical locations, e.g., corresponding to registers of peripheral devices.
- Supplying addresses (buffers) to perihperal devices as parameters of physical I/O operations.
- Setting up a page table

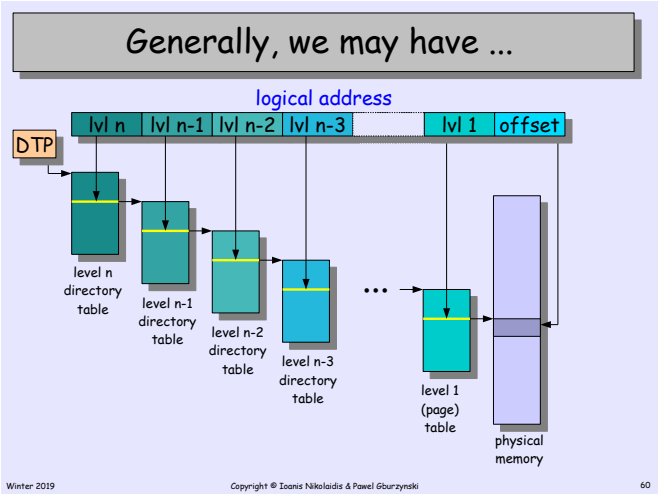
Peripheral Devices

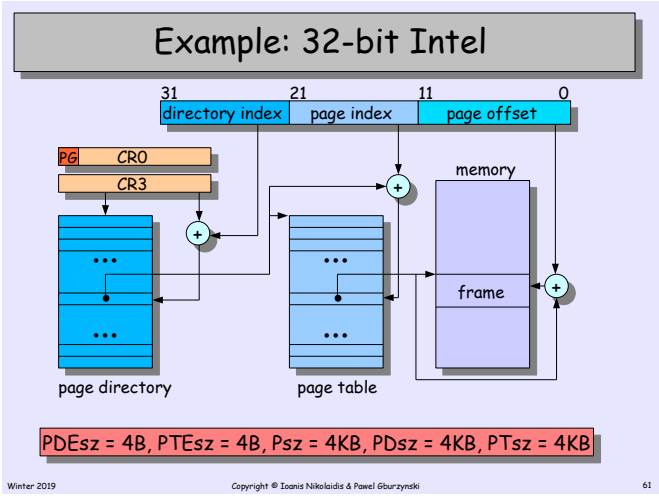
Note: generally, peripheral devices cannot take advantage of address translation as these translations are context-relative (different processes see different logical address spaces). When a periperial device is running, the CPU context usually describes an unrelated process.

Multi-level Tables

The actual (used) portion of a program's logical address space does not need to be continuous. Also, different chunks can be assigned different accessibility attributes. Thus, it makes sense to assume that the logical address space has the same generic layout for all programs (large and small).

Problem is that some programs are so small that having a fixed size page table for all programs will be overkill. To solve it we use herarchical design. This design can be slow for reading as you have to read n levels of redirections but the buffer (TLB) fix this issue.





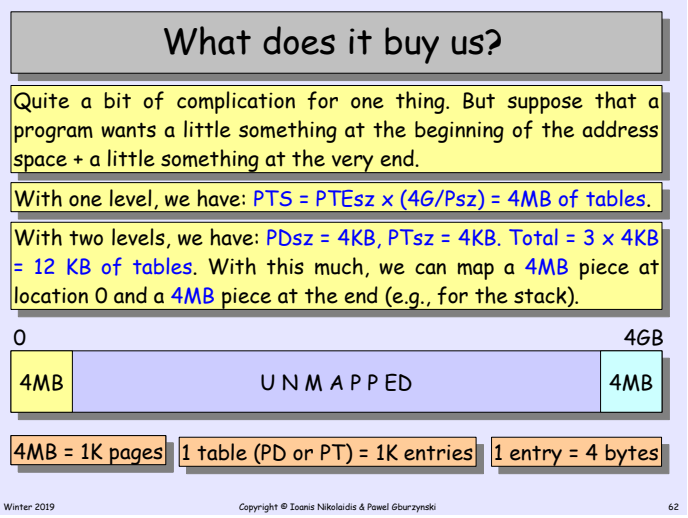
The n-th level contains the offset for the n-1-th level in the directory table to gain access to the page table. From there using the offset allows us to get access to the frame without storing a whole bunch of invalid page tables.

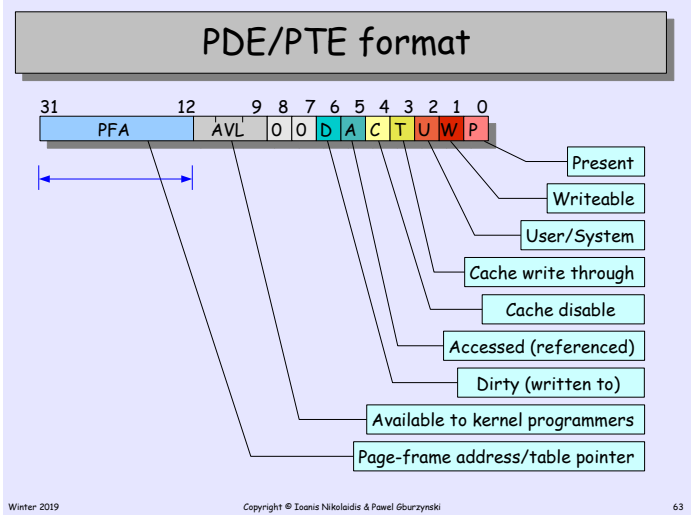
The 32-bit Intel is an example of how a herachical design can work. This is a two level entry.
Page directory entries that are invalid will not have a page table as they would be completely empty. Therefore, you will only have page tables that have a valid entry.

Intel exception

Intel also does segments as well as this form of memory magement.

What does it buy us?

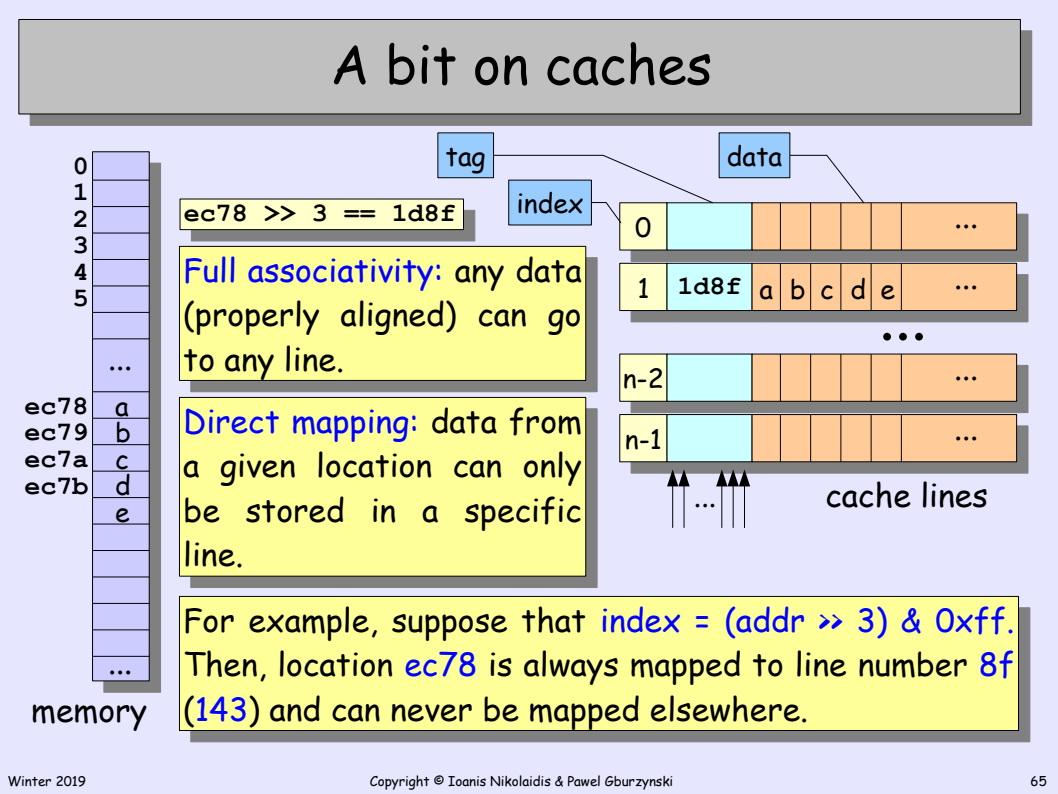




A Few notes

The size of the PFA determines the maximum amount of physical memory that the system can use. Attritbutes can be specified on a per-page or per-directory basis. Generally, it makes sense to associate accessibility attritbutes with chunks larger than pages. Sometimes, in two-level hierarchies, the directories are called **segments**. This reflects the fact that they provide a coarser grain of allocation - one that better corresponds with logical segments of programs. With all these levels of indirection, the cost of a memory reference is bound to skyrocket, unless something is done. This is taken care of by some extra hardware called the **TLB**.

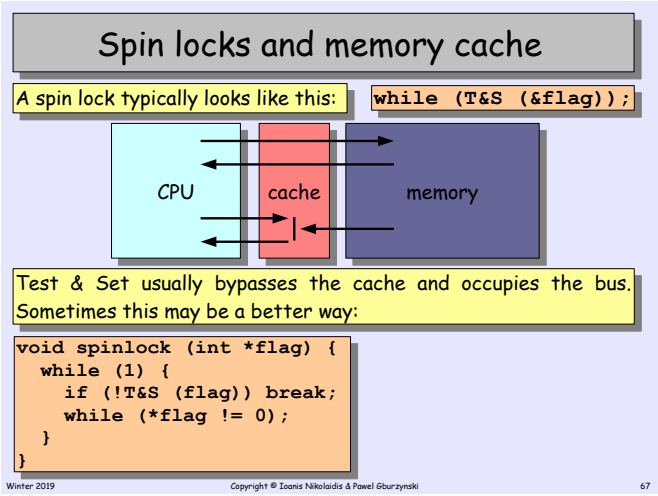
Caching

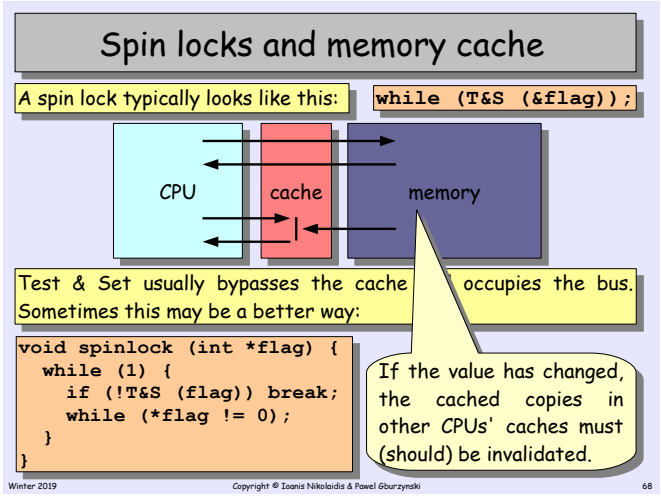


The tag is the prefix in memory where the content in memory is stored. Generally the cache line is fixed and is a power of two, in this case its 8 bytes. The write back policy is to update the entry in cache, and not to the memory unless the cache needs to be flushed. The write through policy writes it both to cache and memory. Both are important in certain cases.

Characteristics of a cache

- **Cache Size:** The number of lines and line size.
- **Associativity rules:** Fully associative, direct mapped, some hybrid.
- **Replacement Policy:** How the lines are recycled, Note associativity rules may restrict this policy.
- **Write policy:** write-through vs write-back – write to memory every time the cache is been updated vs write to memory when the cache is flushed and has been modified.

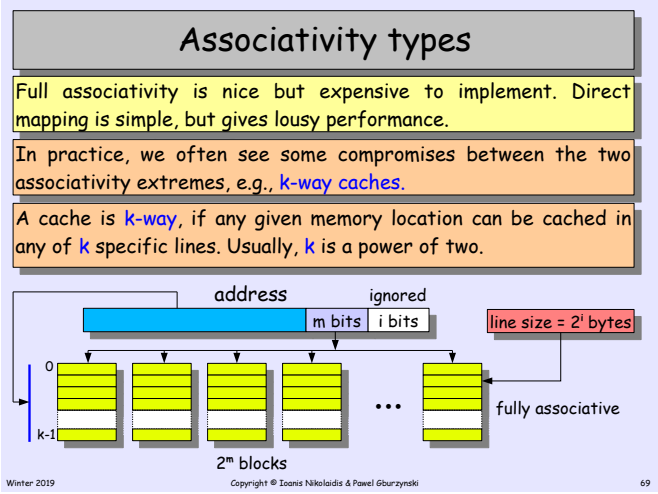


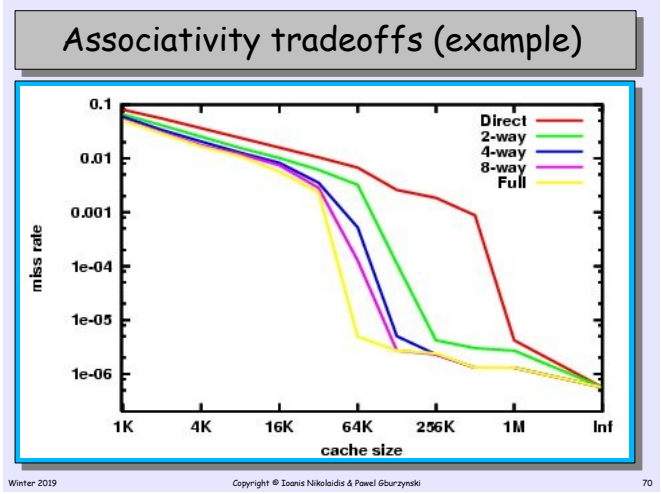


The during a spin lock the cpu will consume the entire memory's bandwidth during this situation as it wants fresh information, not information from the cache. This is undesirable.

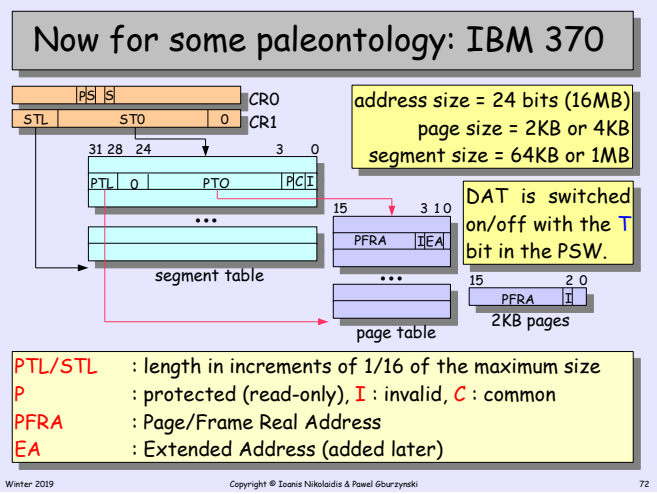
The test and set is done only once reading directly from memory, if the core didn't win the competition then read from the cache. Once the flag in memory is updated the cache in the cpus are updated/invalided so that they can start to read from the memory.

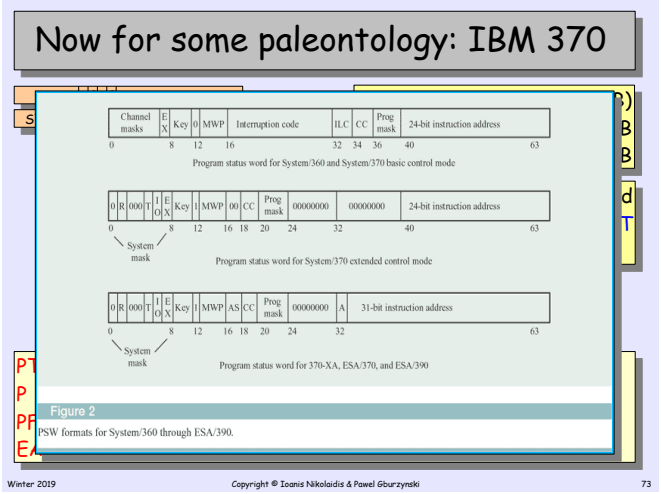
Associativity types

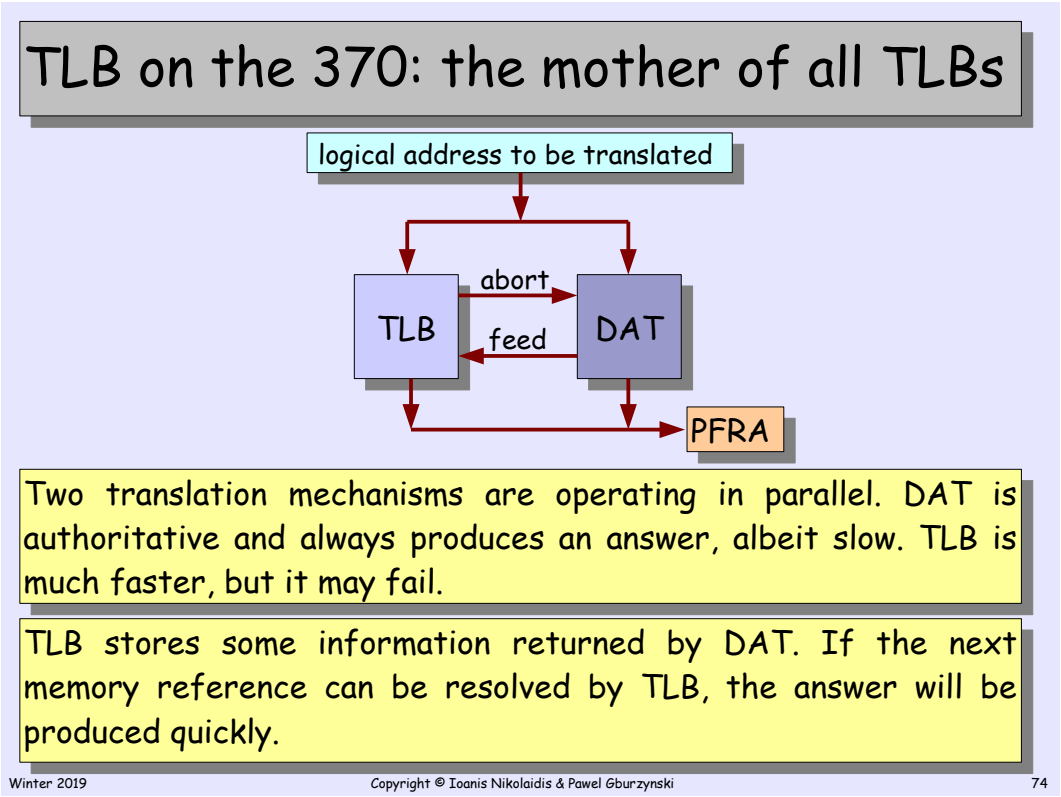




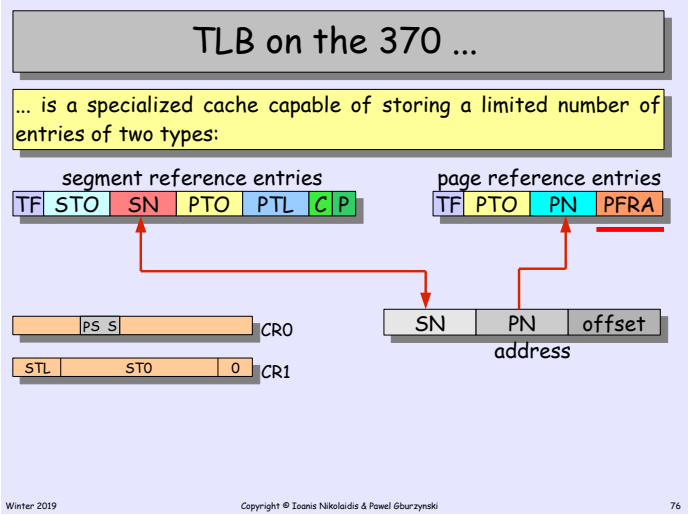
You get the best performance with full assoicativity. But is more expensive.



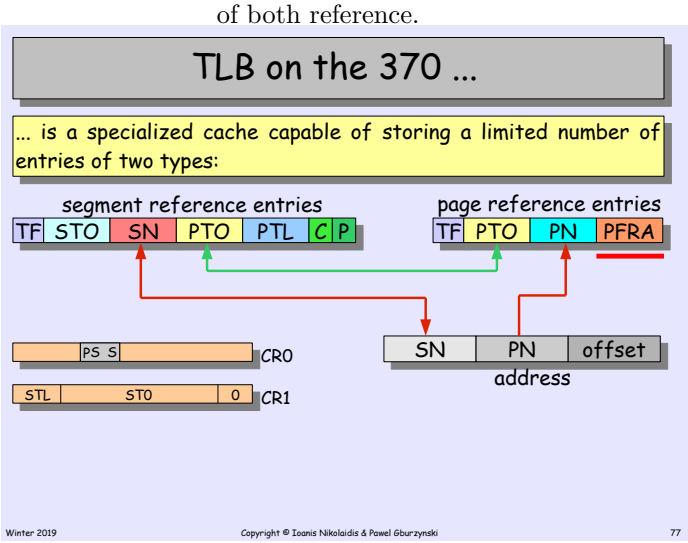




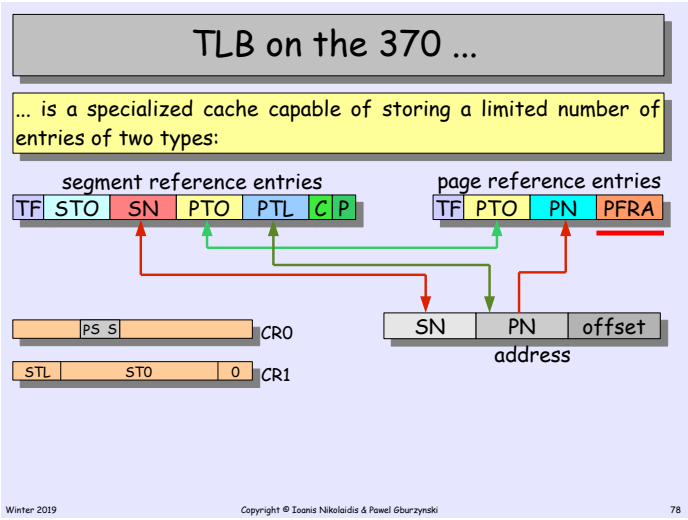
The IBM 370 have two forms of translation mechanisms, the faster TLB which may fail, and the slower but always correct DAT. These two mechanisms operate in parallel when the cpu is trying to access memory. The TLB will always finish faster if the memory address is stored within its cache. However, if it does not finish then the DAT will resolve the translation and store the result into the TLB for caching purposes.



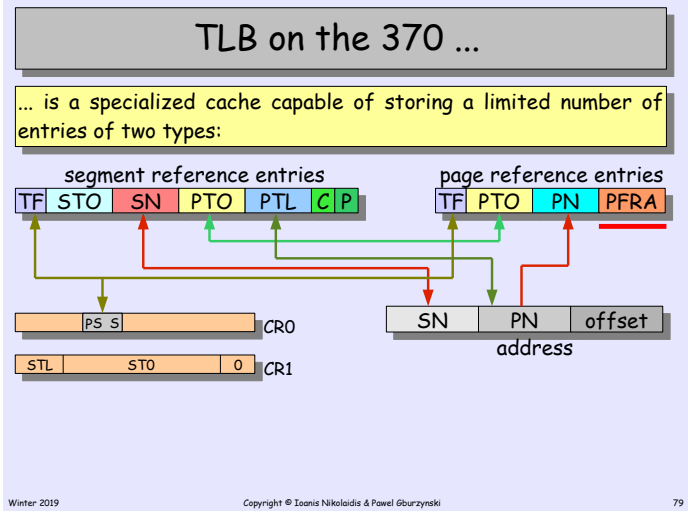
Try to find a match of both SN and PN in the TLB of both reference.

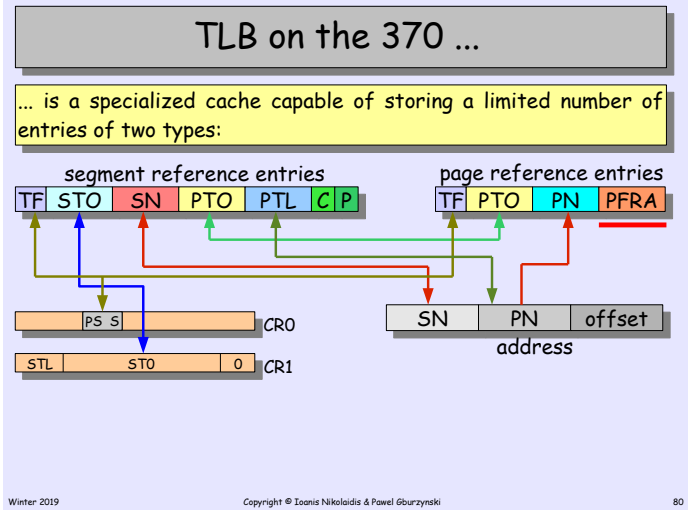


We maintain the PFRA and the PTO in the TLB. There must be sanitiy checks done to be sure that data is part of the process.

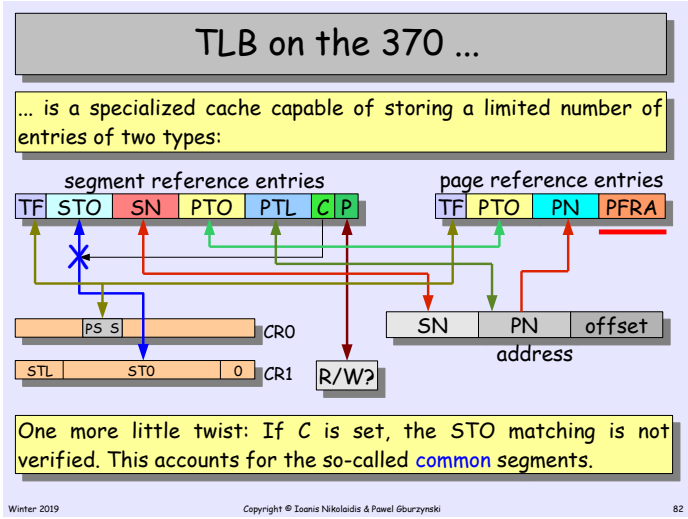


is the PN less than the PTL? This is part of the sanitiy check.

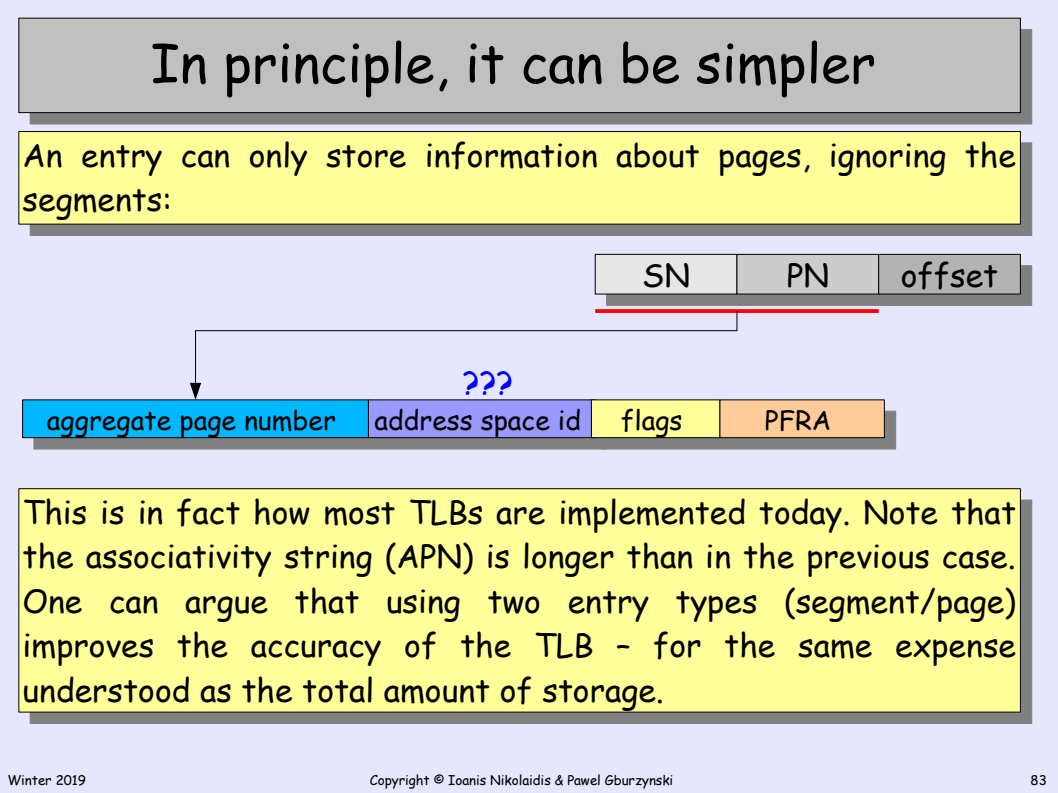




Is the STO in the segment reference entry match the CR1, if so then its a cache hit.
This is done very quickly (sub nano seconds) but very expensive.



P is the premission of the segment reference



In the defense of today's implementation, space is cheap.

Coherence of TLB caches

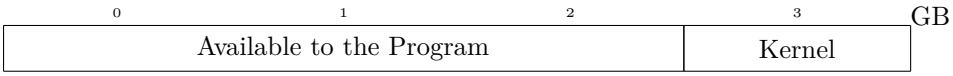
Entries in the TLB may become **obsolete** (and incorrect) when the actual mapping changes, e.g., a page is unmapped or remapped. This tends to happen quite often. Upon a context switch, completely new tables become active. There are two solutions to this problem.

1. **Completely invalidate the TLB**. This used to happen on the old Intel CPUs. Writing to CR3 automatically invalidates the TLB.
2. **Retain all entries, but make it possible to distinguish different address spaces** (the 370 approach)

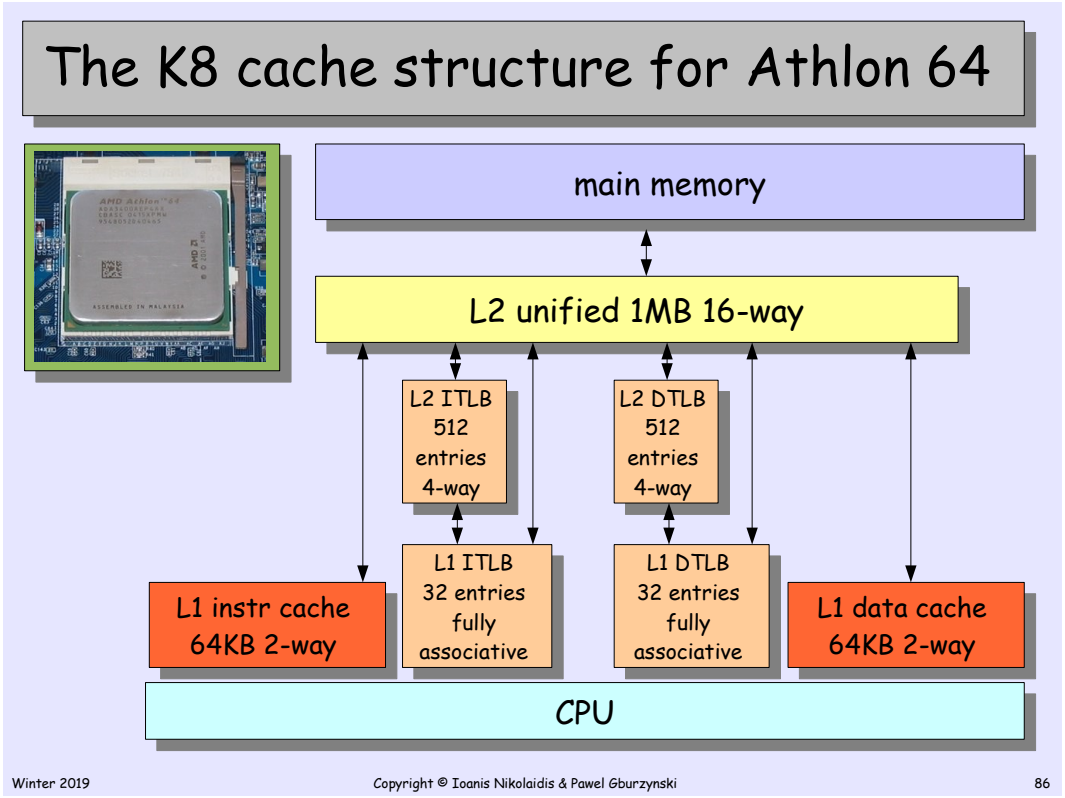
The latter is more efficient in the face of frequent context switches. Additionally, the entries pertaining to the common segment are never invalidated (as long as its mapping doesn't change)

Common Segment

A popular approach to organizing the logical address space of a process is to include the kernel memory in it. This is the case in older Linux kernels.



We not only execute the kernel on the process's stack, but also within the process's address space. The kernel part is in fact a common segment: it occurs in all address space (always in the same location). Currently, modern 32-bit systems tend to depart from this approach.

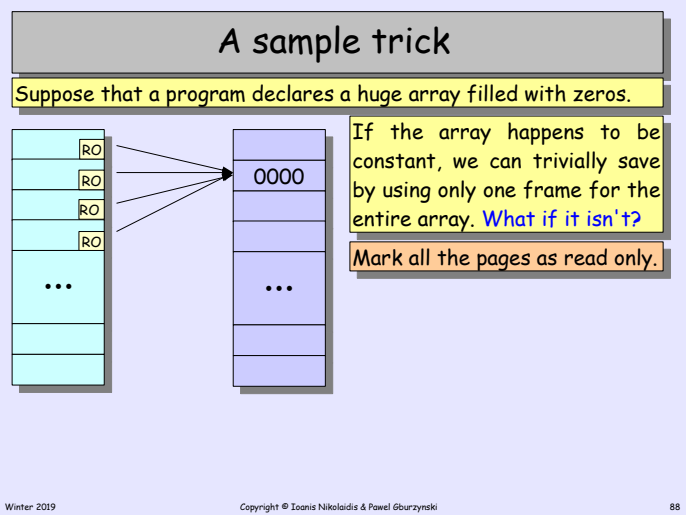


Starting from a clean state will cause the cpu to access the main memory quite often, during which will fill the cache. But once the loading is done then subsequence calls will probably be faster due to the locality of data and instructions.

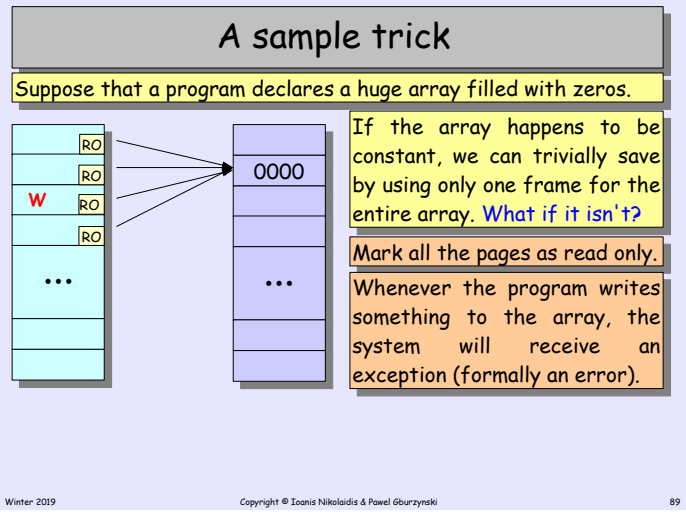
Dynamics of address spaces

By now, we should be comfortable with the idea that the logical address space of a process contains holes. Suppose that a program gets loaded into memory. How does it get loaded. DLL are one method, but what about the stack? How can we know how much of it the program is going to need? Can we somehow figure it out automatically (without forcing the program to tell us anything) when the program is going to use some memory, so that we can bring it in as needed?

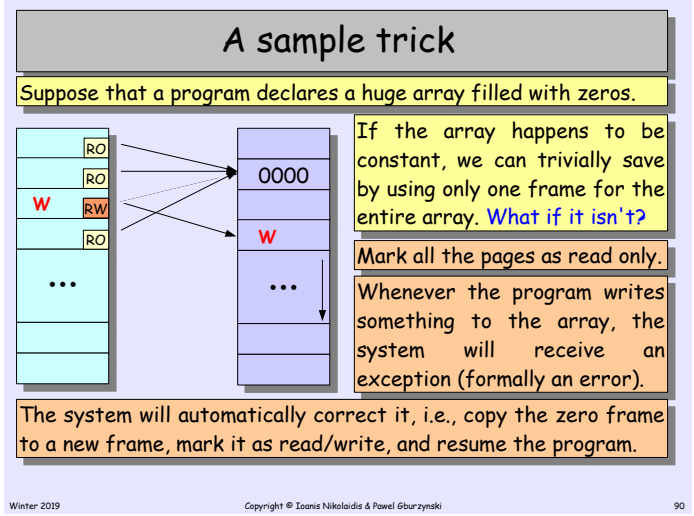
A situation of an unintialized array.



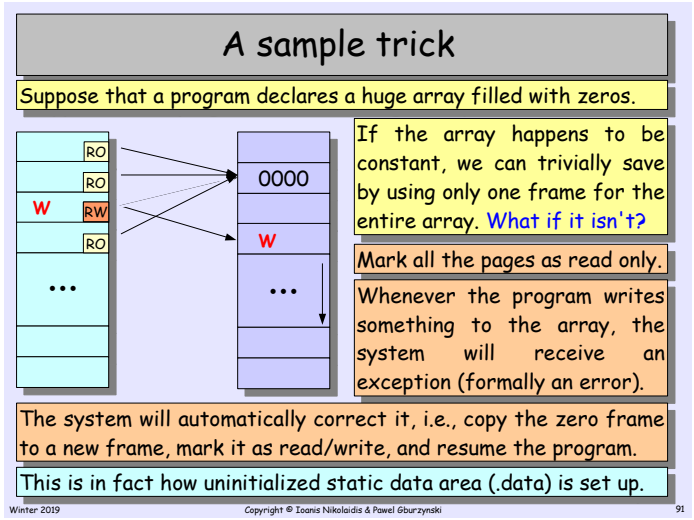
Imagine an array of unintialized data. To prevent redundent frames, the elements of the arrays all point to the same frame. The elements are read only as they don't want to change the default value.



Lets try to writing something into the 3rd element. But all the pages are marked read-only.



For that page the system remaps it to a new frame where it can now be written.

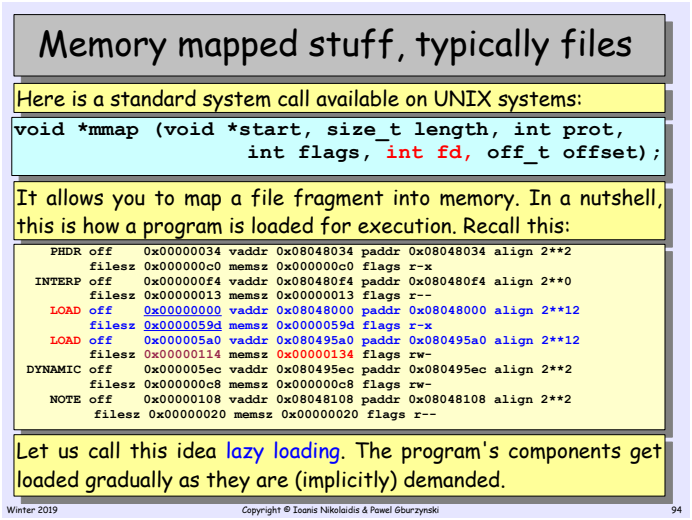


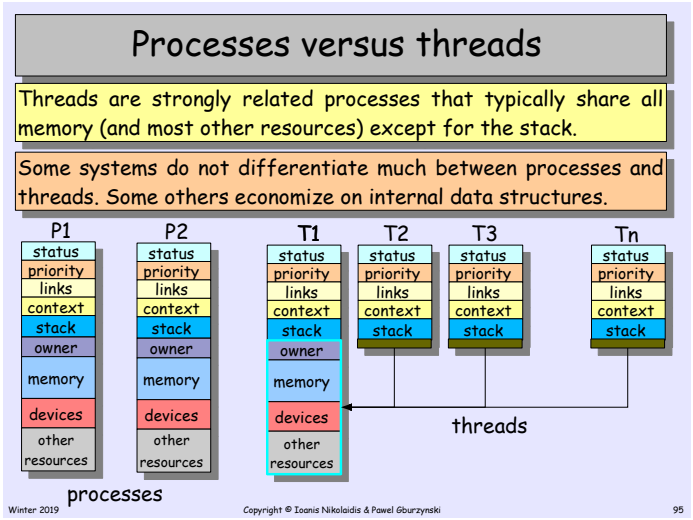
Copy on Write

Recall fork. This is how the operation is carried out. The child inherits exactly the address space of the parent. No memory is copied. All memory that is not to be actually shared is marked as Read only + Copy On Write. For as long as the two processes **do not write to some area**, they are going to share the frame. This is why fork is so simple and inexpensive physically, while being so powerful as a concept. This hints at a more general idea. Let us call it CRAPE for "Creatively Responding to Apparent Program Errors." The program errors we have in mind are called **page faults**.

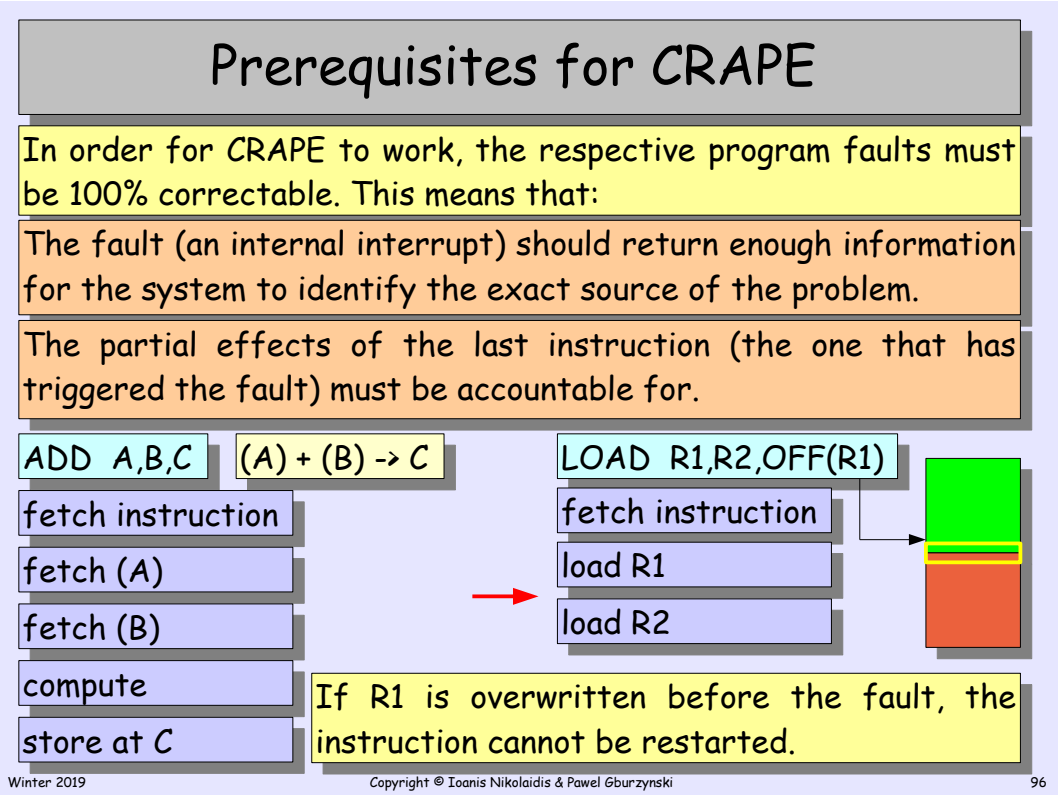
More tricks

When a program pushes something onto a nonexistent page of its stack, the system allocates a blank new frame and resumes the program. **No need to load the entire program at once.** As the program runs through its code, the system receives an exception whenever the program references a new (so far untouched) page. Then it brings the new code fragment into memory and resumes the program. **No need to allocate the entire data at once.** by the same token as before. Suppose that the system wants to **emulate** something (e.g., a device) as a bunch of memory locations. It can easily intercept all references to a given page range and fool the program into thinking that is actually reads from and writes to memory.





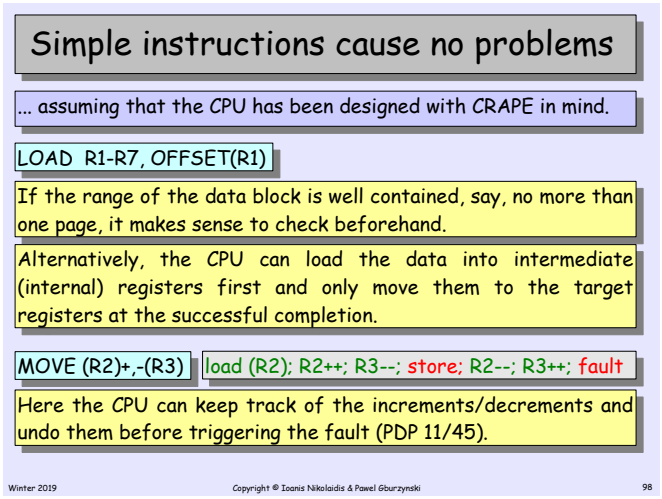
Prerequisites for CRAPE

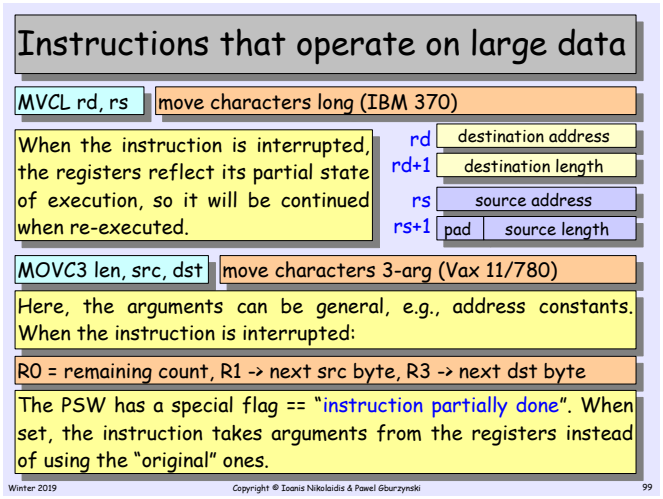


There are two approaches to solve this issue

1. Before executing a potentially unrestartable instruction, check if all the data it needs are in memory. If not, trigger an exception before starting to execute the instruction.
2. While exectuing the instruction, keep track of its partial effects. There are two alternative solutions regarding what to do after a fault.
   (a) Undo the partial effects before triggering the fault.
   (b) Save the partially done state, and when the instruction is restarted (or resumed in this case) - just keep going

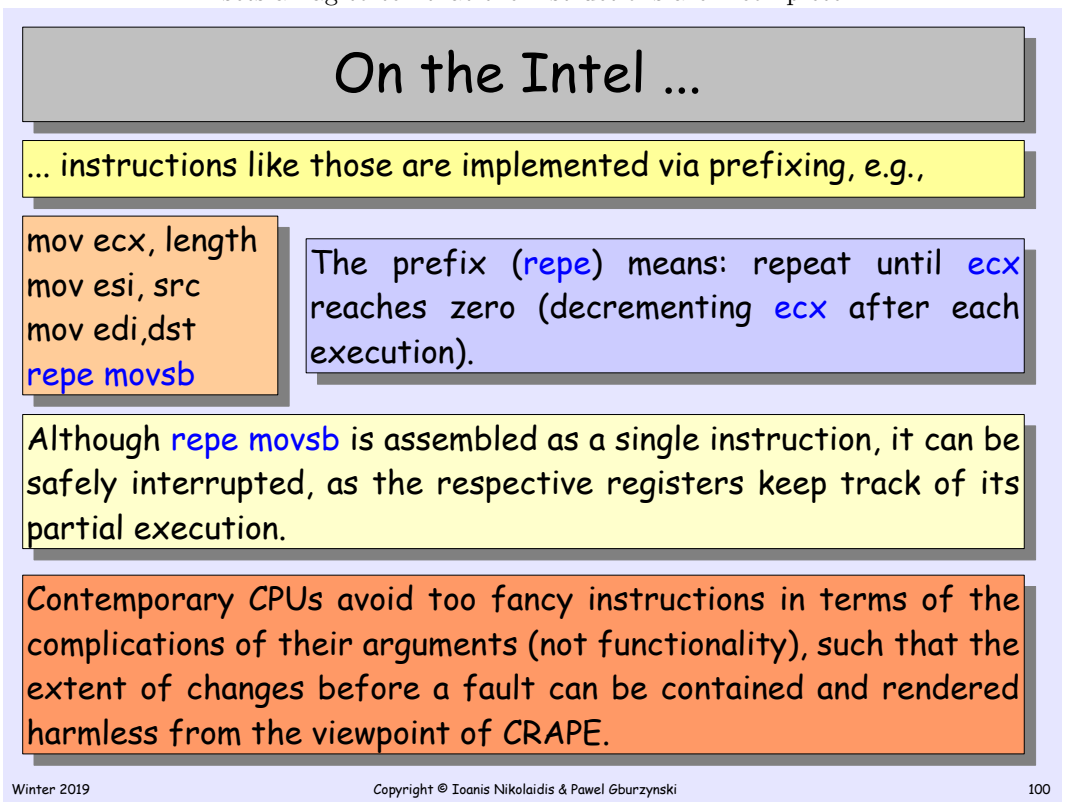
   The first one is tough, the second one eaiesr





R2 is used as a location of the address. load the content of R2, and then in one step auto increment R2. R3 is the store location, decrement R3 then load the content. This bascially copy an array.

The first implementation use additional registers to keep track of partial completion, the second implementation sets a flag to tell that the instructions are incomplete



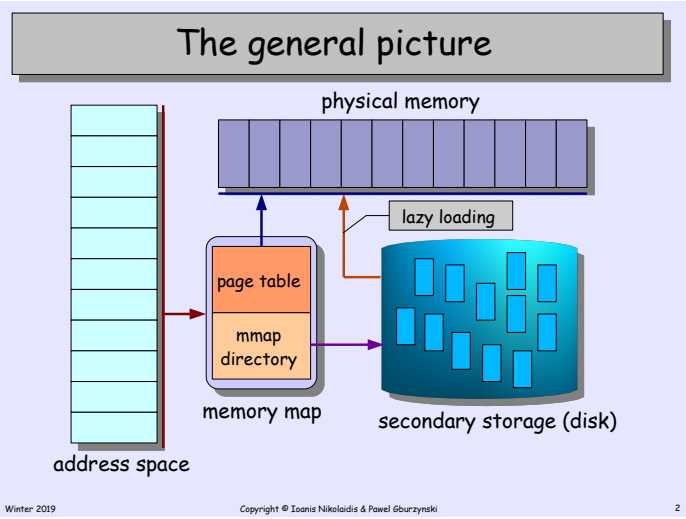
On Intel, the ecx contains how many steps are left when the process suffered a fault.

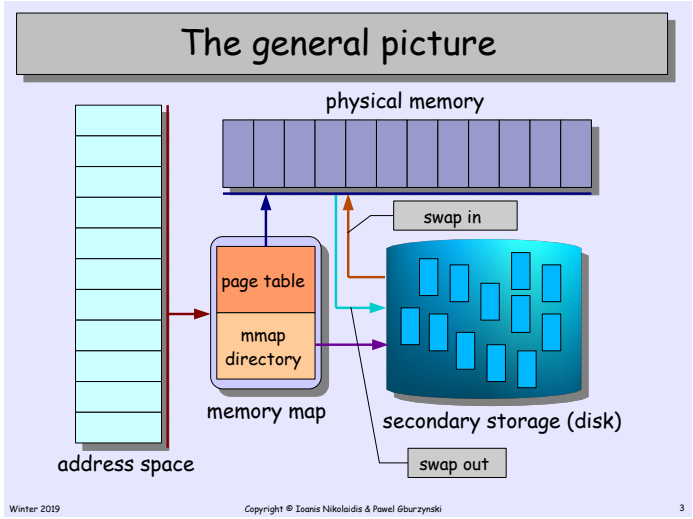
10: Virtual Memory

We determined that swapping is indispensable:

> Physical memory is never tied up to the current configuration of processes in execution. Priority jobs can always be admitted. Physical memory is made preemptible. Thus, memory-related deadlocks can be avoided.

In a partition-based system, a process can be either completely swapped out or completely swapped in. A partially swapped-in state is useless (from the viewpoint of the process). With CRAPE, we can be more flexible. For example, lazy loading upon (implicit) demand looks like a partially swapped-out state. The process can run even though some of its stuff is not in memory.





Swapping out

Note that the static components of the program's address space, i.e., code + constants, are already "swapped out" before the program starts. They are never going to change. Thus, only the dynamic components, i.e., data + stack, will need a special swap area in secondary storage. Also, only these components will ever have to be physically swapped out.
The idea behind this whole scheme is this:

> A partially swapped out program may still be able to run. It may not be needing some parts of its allocated memory area at this very moment. Thus, we may be able to run more programs in whatever physical memory we have than it would seem at first sight.

Swapping at the page-level

When a program demands a new page, the OS may have to find room for it. Basically, it needs to decide what page should be evicted.

The question now becomes how do we do this optimally, i.e., minimize damage, minimize the cost of eviction, etc.

How much memory does does a process need

The system may come up with some idea and assume that the process will be able to live within the confines. This imposes another restriction on the degree of the fanciness of machine instruction.

The question now becomes: What is the maximum number of pages that may be needed simultaneously by the most fancy machine instruction?

Page replacement policies

Goal

To find the page with the highest likelihood of not being immediately needed. We want to maximize the time during which the victim page will not be demanded back. This interval determines the page fault rate in the system.

This is easier said then done due to the random nature of operating systems. The best we can do is make an educated guess.

There are two cases for a page to be purged. It can be **clean** or **dirty**. In the former case, it doesn't have to be sent back to secondary storage. The dirtiness of the page can be determined by a flag that is sent once the page has been modified. Some logical segments are always clean like code, constants, etc. But the data page can be clean as well: typically at least 80% of data references are for reading rather than writing.

Some ideas

Random replacement: The victim is selected at random. This policy does not get good numbers, but not horrible to implement.
Random replacement+: The victim is selected at random from the pool of clean pages first before considering dirty pages. This idea is generally bad as it will purge out all code/constant pages after a while. We want to find a page that is likely not going to be needed for a long time. Then look at its clean/dirty status. **Not the other way around!**

FIFO

Select the oldest page in memory. This one is easy to implement.

The system maintains a list of pages. Upon a page fault it manipulate the link list by evicting the first page in front of the list. The new page is appended onto the back of the list.
The operation is constant O(1)
The rationale makes sense: a page that has been in memory for a long time is likely to have been explored and thus not needed.

Problems with FIFO

However, it does not account for permanently needed pages. An example is shown below.
Being old does not mean useless. Sometimes following the doctrine of the Dawi is a good thing.

```
1     double sum;
2     int i;
3     double HugeArray [N];
4     // more code
5     sum = 0;
6     for(i = 0; i < N; ++i){
7         sum += HugeArray[i];
8     }
9     // more code
10
```

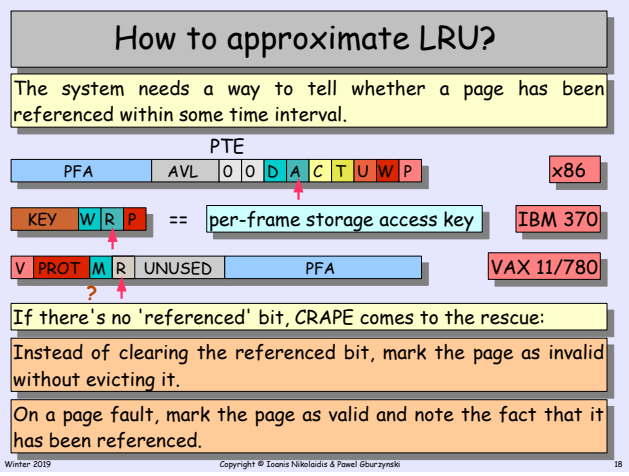
i and sum are permanently needed, even though the traversed and explored fragments of HugeArray may be not.

This is rather typical. In any computing problem there are data that are needed all the time like counters.

LRU (Least Recently Used)

Purge the page that has not been used for the longest time. This is not difficult conceptually and can use the same data structure from FIFO. However, its behavior is different. It manipulate the link list at **every** access. For every access you have to bring that data to the back of the list. Therefore, it is more complicated than FIFO. When we need to purge a page, the head points to the page that hasn't been used for the longest time. Unfortunately, this calls for an expensive action to be performed after every memory reference. This is why LRU can only be approximated.

However, if you create the data structure simply then each operation can be quick.
Moreover, "LRU" support distinction of referenced vs not-referenced pages.

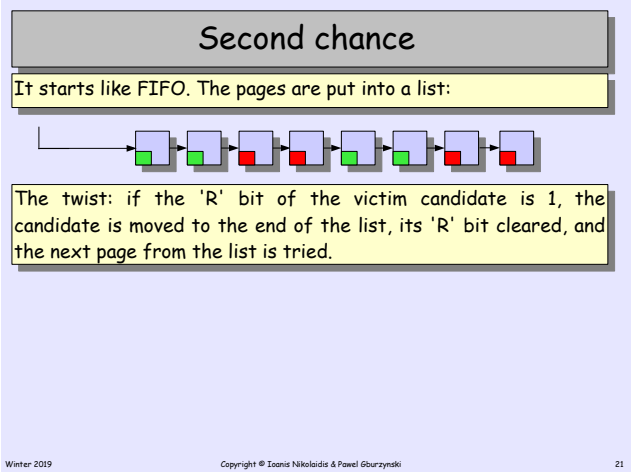


Time interval

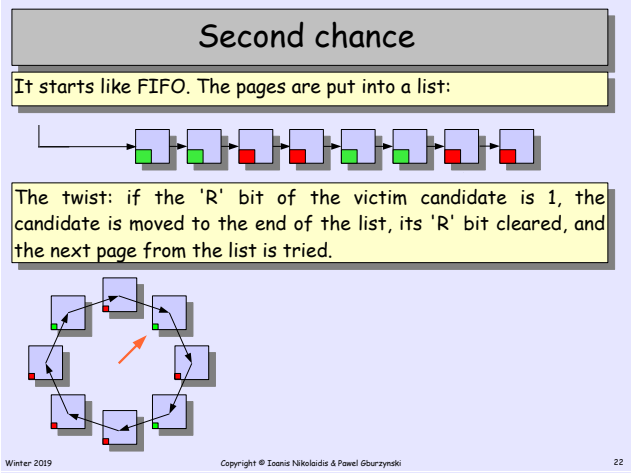
An idea to solve this LRU approximation is to can go though the list at a fixed time interval, or in this case # of memory references. If it finds a page whose 'R' bit is set, it moves the page to the end and clears the 'R' bit. This isn't very smart. What if the last page gets referenced after having been checked and just before the fault? We can do better.

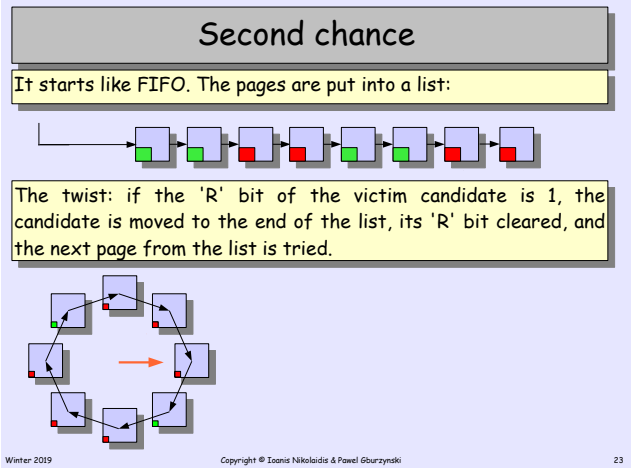
Second Chance (FIFO Optimization)

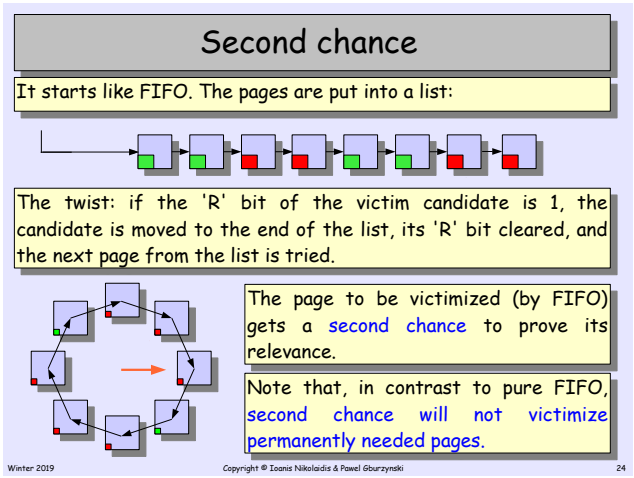
We stated that FIFO has problems storing permanent pages. Second change will help fix this problem.



The twist: if the 'R' bit of the victim candidate is 1, the candidate is moved to the end of the list, its 'R' bit cleared, and the next page from the list is tried. Worst case: having to traverse the entire list again, only happens on page faults. The behavior is LRU-like.







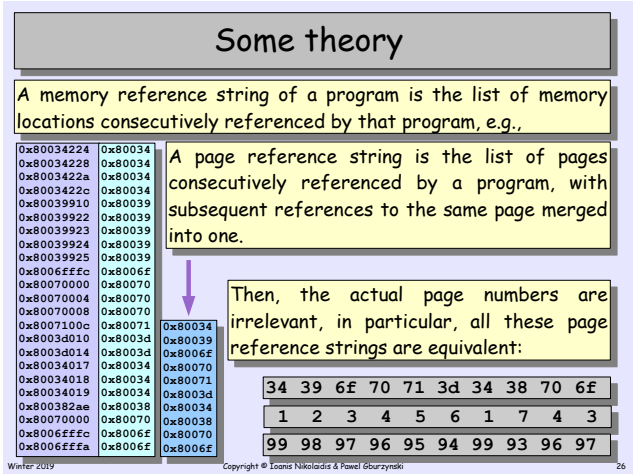
The page to be victimized (by FIFO) gets a second chance to prove its relevance. This is different from pure FIFO as the second chance will not victimize permanently needed pages.

Performance

Second chance is a very good approximation of LRU, especially for its cheap price. In some variation it is the most popular policy being used in actual OSes.

However, there are other components involved. With multi-programming, it is also important how to divide the available physical frames among multiple processes.

Some theory



But we can make it reasonable.
Can we Eliminate page faults? NO.

Monotonic strategies

Suppose that the system has admitted a program for execution. It says: you run with a local replacement strategy. Here is your real memory, k frames total. This is how much I can afford you at the moment. Needless to say, I don't care how grandiose ideas you may have regarding the size of you **virtual memory**.

The program runs and generates too many faults for the OS to be comfortable with.

So it says: OK, I have manged to scramble a few more frames. So now I will give you a bit more. Hopefully this makes your life easier.

It is natural to assume that more physical memory (for the same program) will translate into fewer page faults.

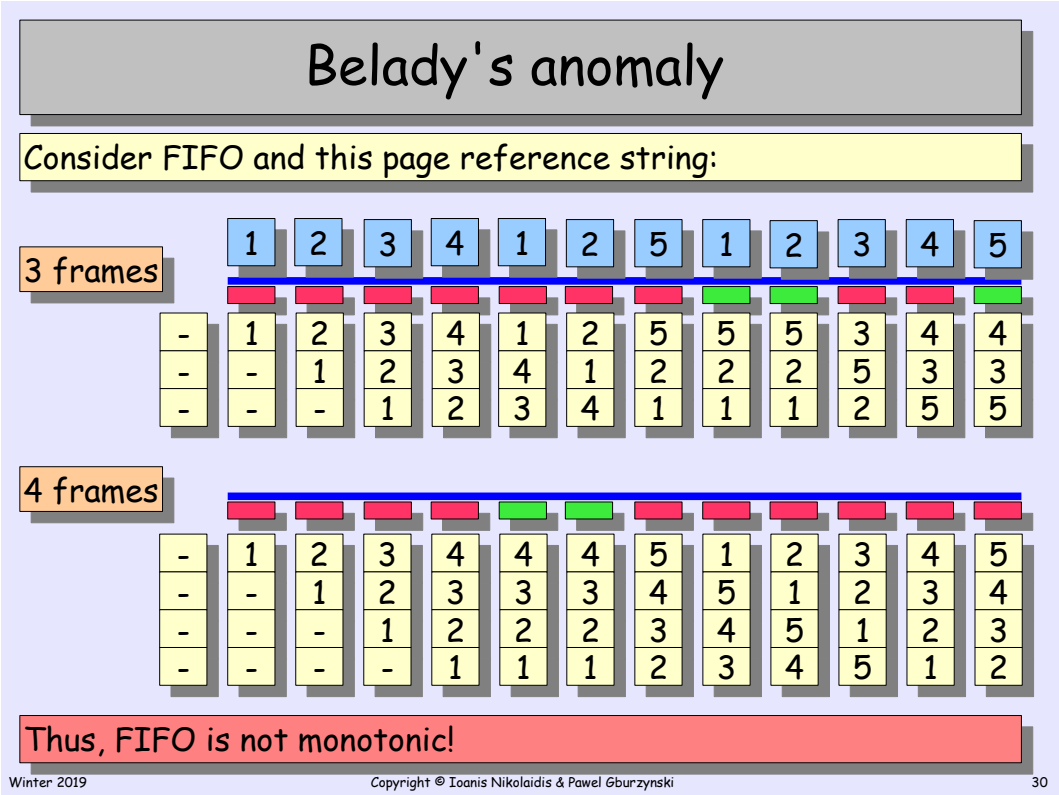
Definition

A strategy is **monotonic**, if no program i.e. page reference string (under a local variant of the strategy) will **ever** generate more faults when run in more physical memory.

An example is LRU.

This is because the set of k+m most recently used pages includes k most recently used pages.

A program running in k+m frames cannot generate more faults then when run in k frames: **at any time, it has the same frame as before + something extra**.



As you can see FIFO is not monotonic as there are more faults with 4 frames (10 faults) than with 3 frames (9 faults).

What can a page fault mean?

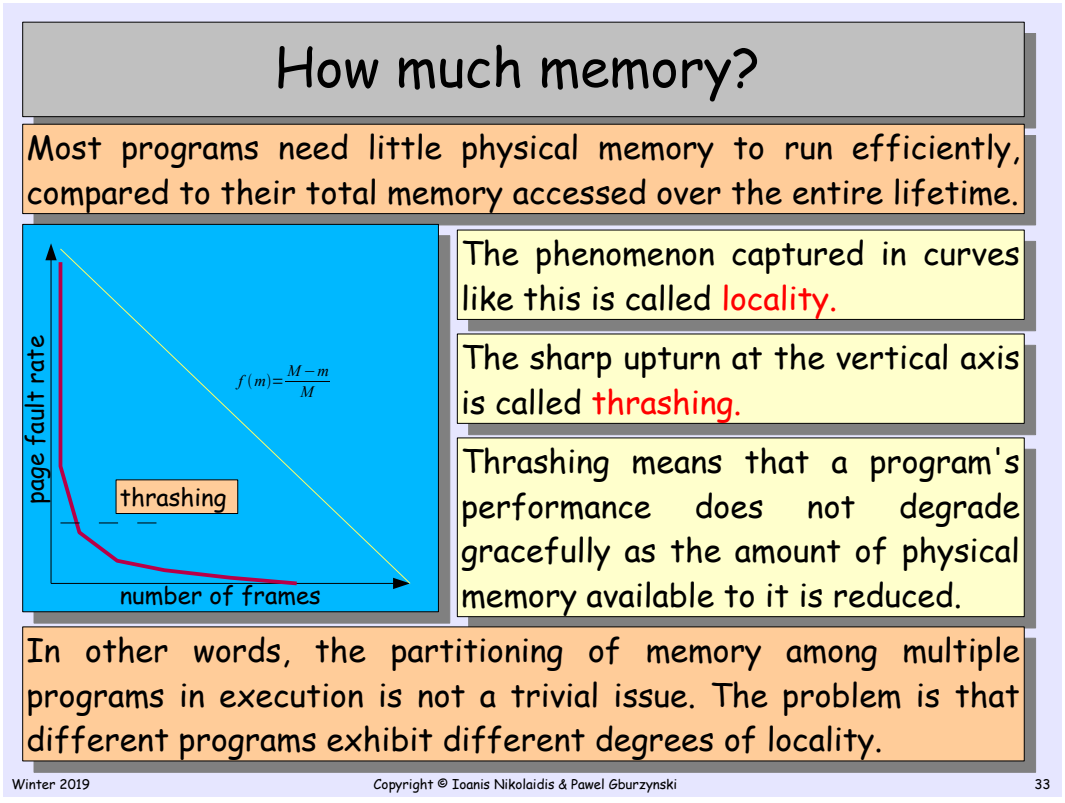
1. An error (an actual illegal memory reference)
2. A CRAPE trick that only requires some change of flags, etc.
3. Request for a brand new page, e.g., the stack has grown.
4. Request for a static page (code) that must be fetched from the executable file (or a mmap'ped file).
5. Request for a data page that was once swapped out and is now demanded back.

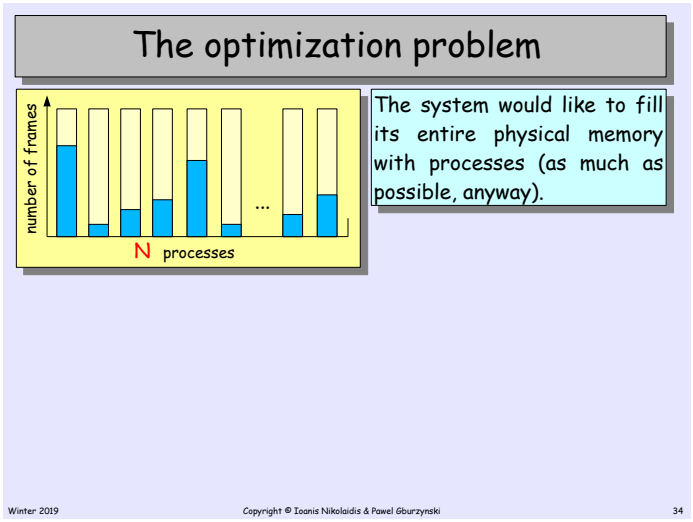
What caused it.

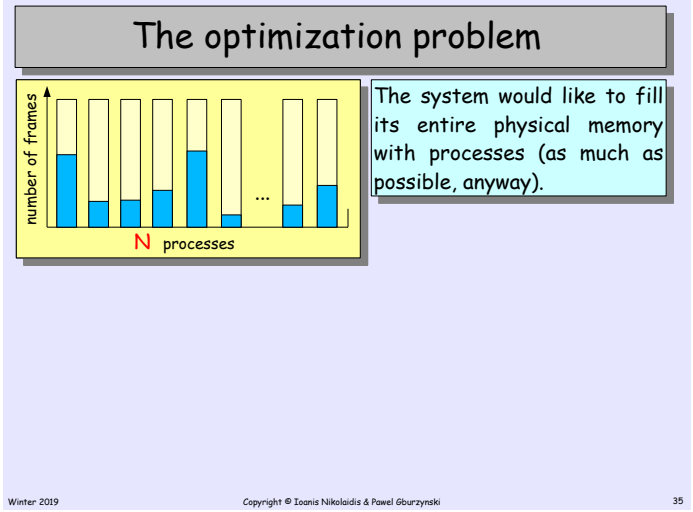
1. This is a programming error, fix your damn program.
2. No need to do any page transfer at all -> no blocking
3. A frame may have to be evicted. This may require sending it back to the swap area -> possible blocking.
4. Same as 3 + A page has to be brought in -> mandatory blocking. Possibly one more I/O transfer to send the evicted page pack to the swap area.
5. Same as 4

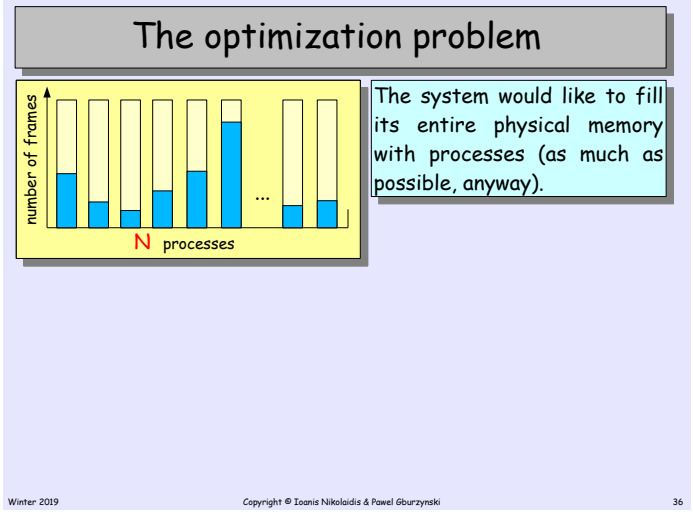
It's easy to determine error 0, the OS looks at the page table and see if the process owns that page. error 3 and 4 are what we are going to look at.

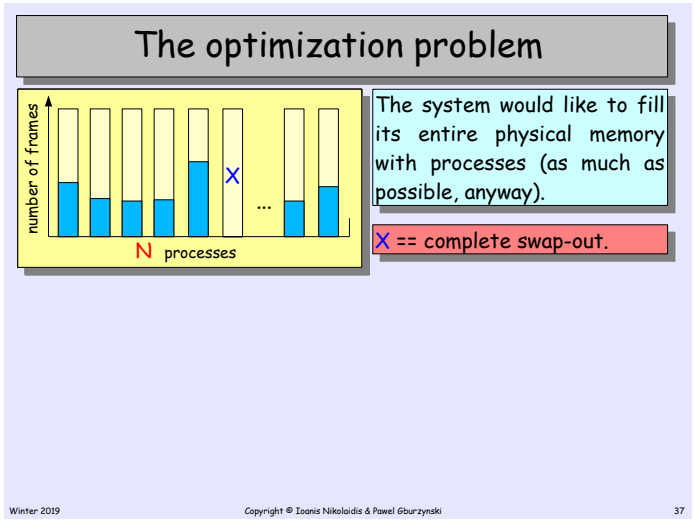
How much memory?



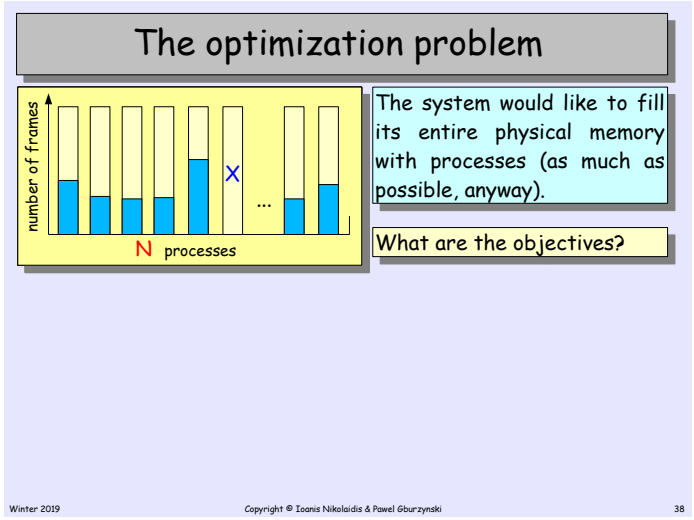


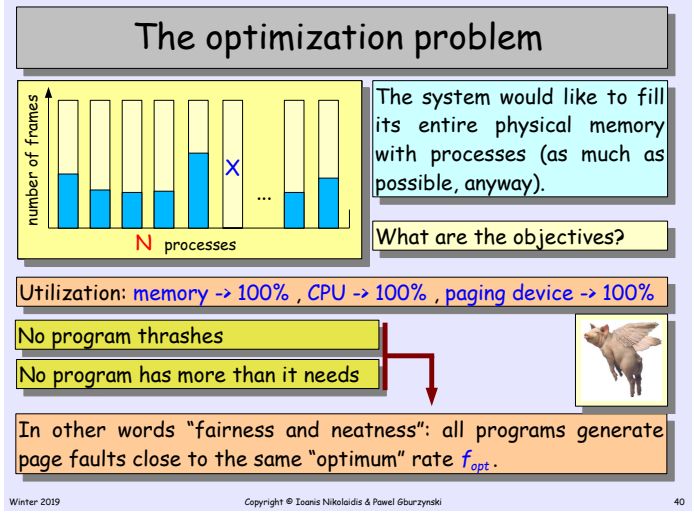






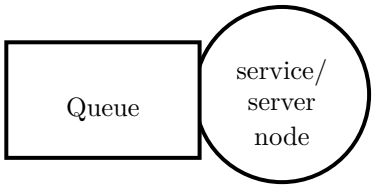
processes may need to be swapped out in order to reduce the page fault rate.





The goal is to maximize all three resources, CPU, Memory, Paging Device. You want to maximize the paging device as you want to guarantee to maximize the memory. Basically if you under utilize the paging device means you don't have enough processes.

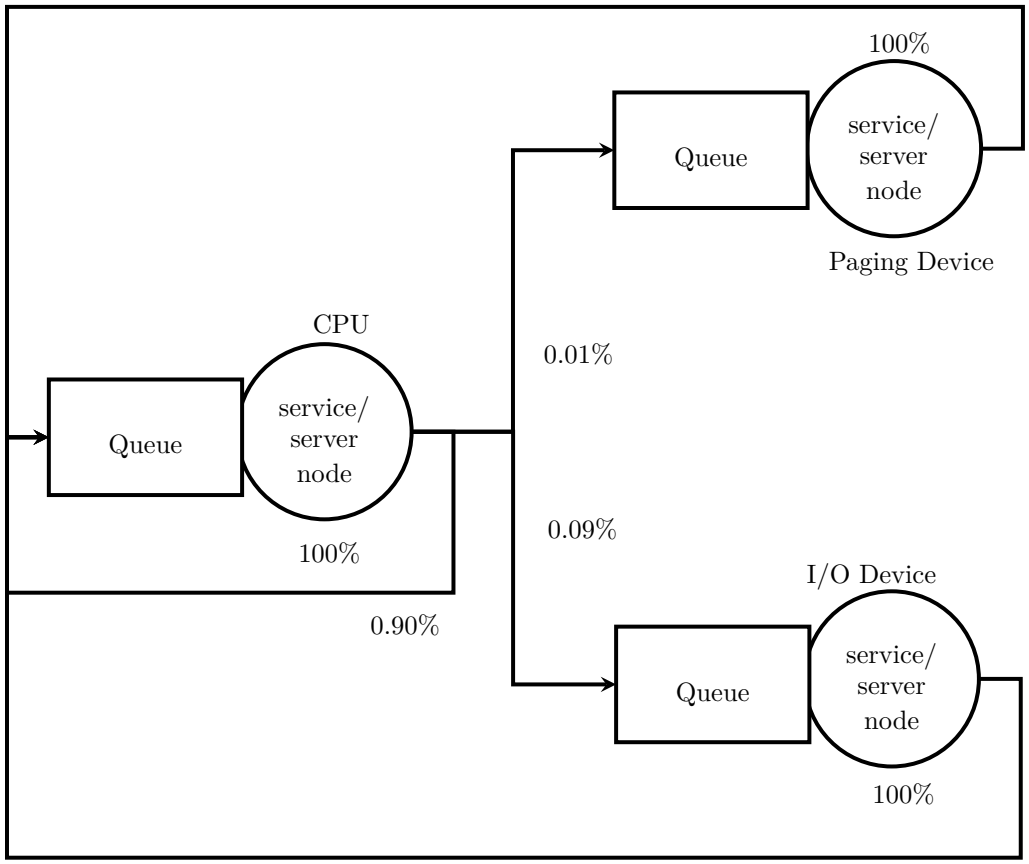
Queuing Convention



You have a queue and a server that is processing the queue. Jobs or in this case processes are pushed to the back of the queue and the server processes the process at the head of the queue. Once the job is done it is either put back to the same queue, goto another queue, or is removed from the system.

Notation

Server: Takes time to complete -> server. Example, the CPU is a server as it is serving or executing something.



Here is a more comprehensive example.

Global vs Local thrashing

Global Thrashing
Is where you have too many programs running and the system cannot handle it. The way to fix this is by suspending and swapping out programs.

Optimum page fault frequency

The paging device has some "average" processing time of a page fault, say t_f. Note that different faults may have different actual processing times. The predominant component is the time to carry out a disk I/O transfer. Let f denote the observed page fault frequency. Note that it is physically impossible for f to be steadily greater than $\frac{1}{t_f}$. This is because the number of processes and the queue to the paging device are finite.
The goal of the paging device is to always be busy, i.e., being utilized, not being idle. We want f_{opt} to be close to $\frac{1}{t_f}$. Now, t_f is essentially equal to the cost of a disk transfer, thus:

$$f_{opt} \approx \frac{1}{t_{io} \times (1 + \alpha)}$$

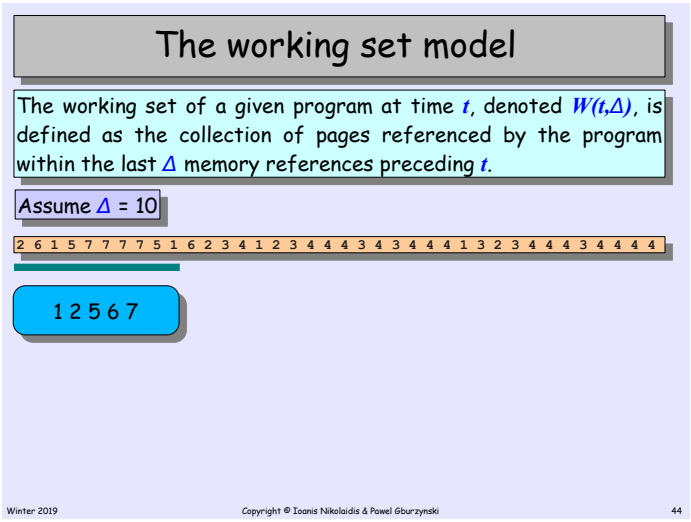
t_{io}	is the average time to perform a disk transfer
α	is the dirty factor

So how much memory?

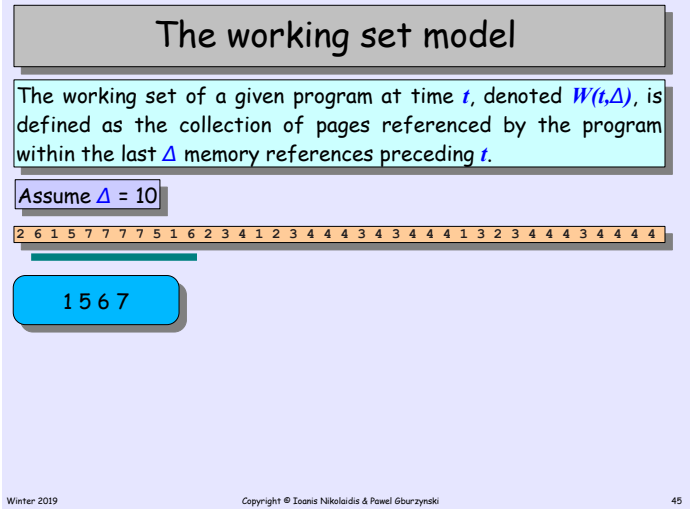
$$\Delta = \frac{1}{f_{opt}}$$

Δ is the number of memory references that a program should be allowed to carry out successfully before it hits a page fault. This gives us a guideline for memory allocation. i.e., Δ is the number of frames that a program should have. Of course this is all wishful thinking as the future is unpredictable, but it's a target nonetheless. Of course we can also look at the program's history as well.

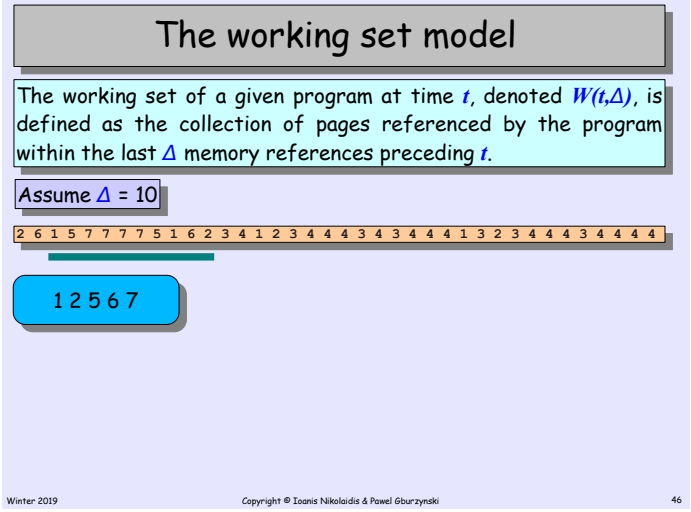
Example

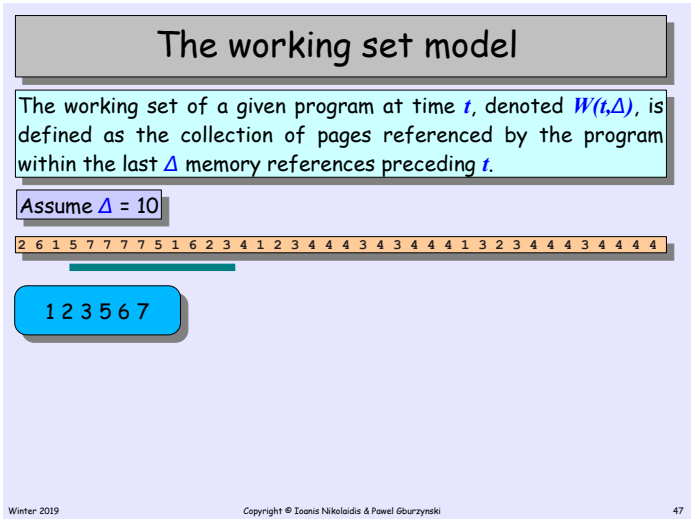


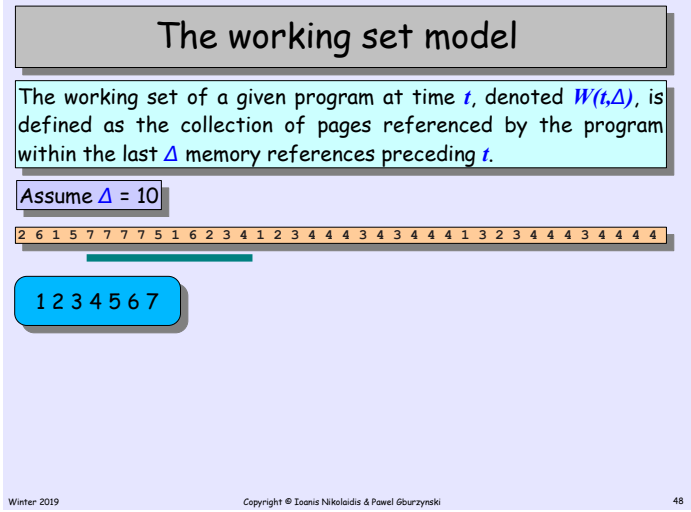
Time is the number of memory references. The blue box holds what memory I've referenced in the past 10 references.

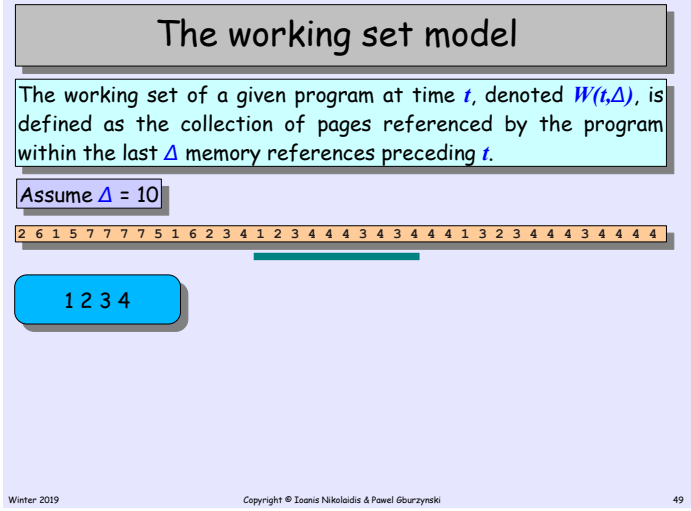


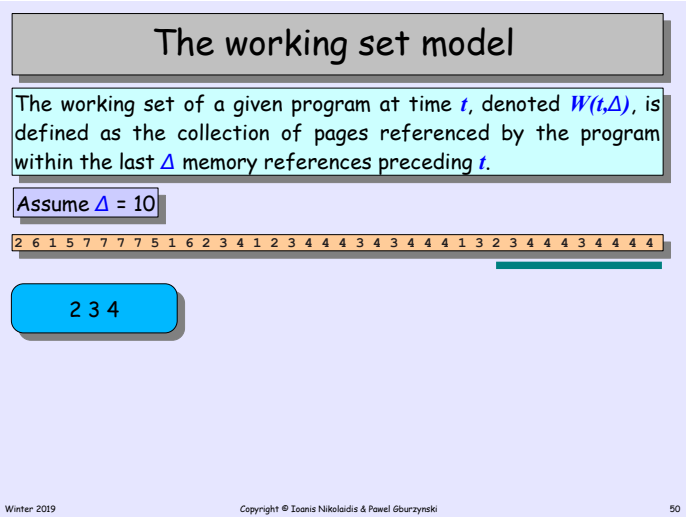
shift time forward and update the list

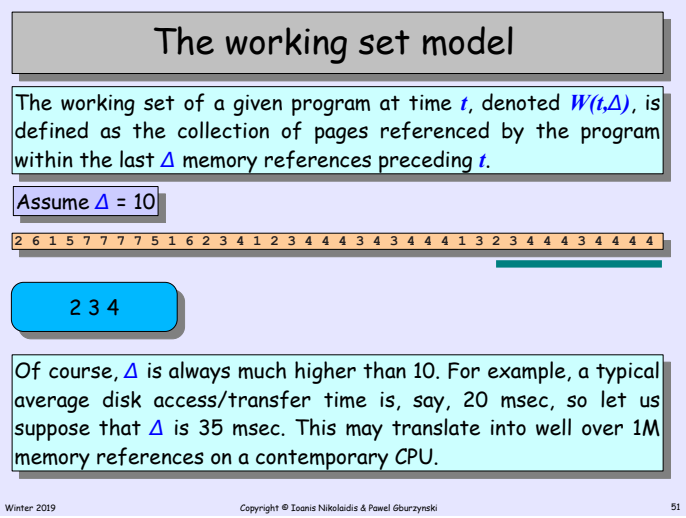












A simple solution

Assume that time is measured in memory references (and nothing more).

Let	
N	be the number of processes in execution
M	be the total amount of physical memory available to processes
$\|W_i(t,\delta)\|$	denote the size of the working set of process P_i at time t

Total demand for frames at time t

$$D_t = \sum_{i=0}^{N-1} |W_i(t,\delta)| \qquad (3)$$

$D_t < M$ the system may be able to admit a new program for execution.
$D_t > M$ the system cannot fulfill the demands of the present programs and may have to consider a complete swap-out of some.

Unfortunately there is no way for the system to know exactly the working set size of a process. Even if the system could magically know that size, it wouldn't make sense to reconsider the program admission or swapping-out after every memory-reference.
Thus, this is all just theory. But it gives us ideas and hints for a practical implementation. For example, the system can monitor the page fault frequency and respond to its fluctuations.
$f > f_{opt} + \epsilon$ add more memory to the program
$f < f_{opt} - \epsilon$ take some memory away from the program.
In any case, it makes no sense to be jumpy with drastic decisions.

The triple point goal

The goal of is to have 100 percent utilization of the CPU, paging device, and memory. That is you want all three components of the system to be doing something and not idling. If you system cannot reach this goal then:

- you may not have enough jobs to run
- you system may be misconfigured

In general it doesn't make to buy:

- more memory than you need
- a faster CPU than you need
- A faster (and larger) disk than you need

How it all looks like in real life

The theory is OK, but its practical embodiment tend to be somewhat confusing to the uninitiated. Here are the reasons:

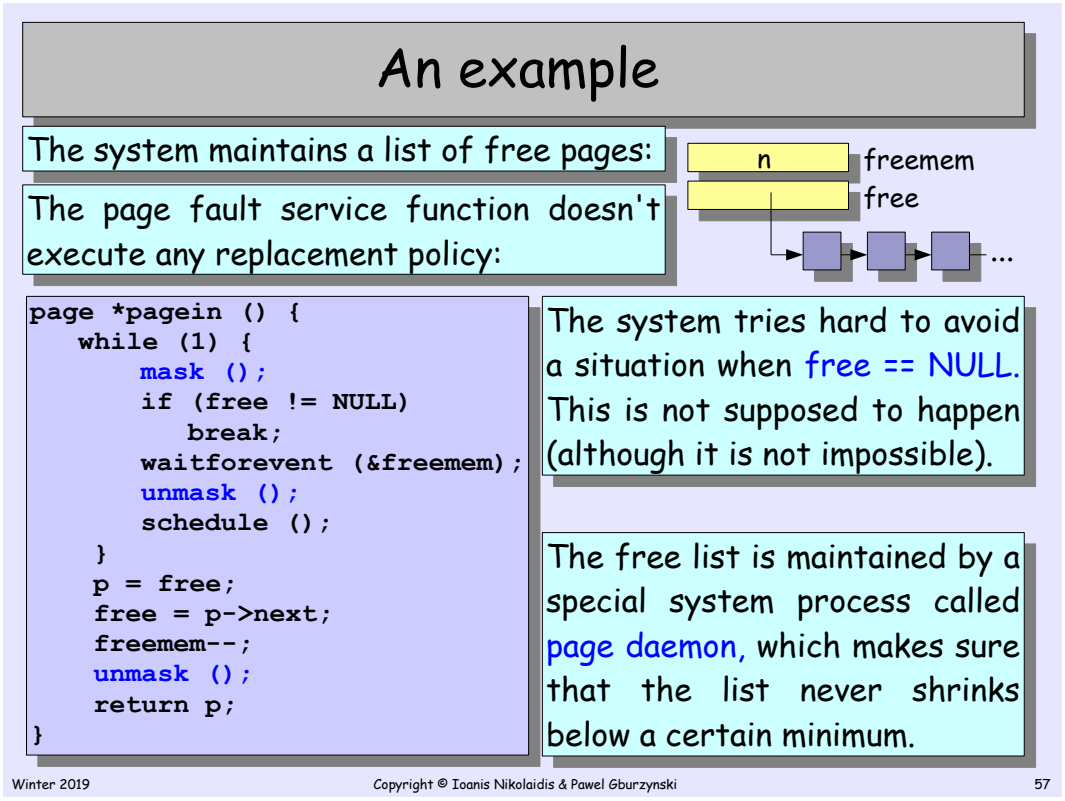
- Pages from processes' address spaces are not the only type of stuff stored in memory (file cache is another important thing)
- It makes sense to separate reclaiming free memory from servicing page faults.
- Creative strategies of optimizing disk i/o operations (e.g., read ahead) add finesse to our options.

Recommendations

- Follow a strategy approximating LRU. Second chance gives you a generic hint: before you victimize, make sure to have looked at the 'R' bit.
- Detect thrashing the easy way (monitoring page fault frequency, looking at the utilization of the paging device). Swap when thrashing!
- Don't worry about memory partitioning. It will take care of itself, if you treat all processes as a single (global) system.

We shall start from an overview of some simple generic (but realistic) approach, which corresponds roughly to old BSD UNIX, and then comment on a few more modern refinements.

Example



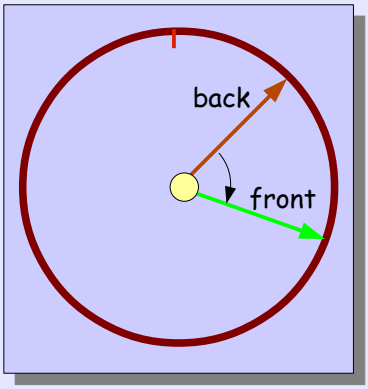
The page daemon

```
void pagedaemon () {
    while (1) {
       if (freemem >= lotsfree) {
          delay (some_standard_longish_interval);
          continue;
       }

       victimize_some_page ();

       if (freemem < desperate)
          try_to_swap_out ();

       delay (victimize_interval (freemem - desperate));
    }
```

Conceptually, the shortest victimize_interval (which gets shorter as freemem gets closer to desperate) corresponds to f_{opt}. If free memory continues to shrink at this interval, it means thrashing.

Victimizer

The victimizer

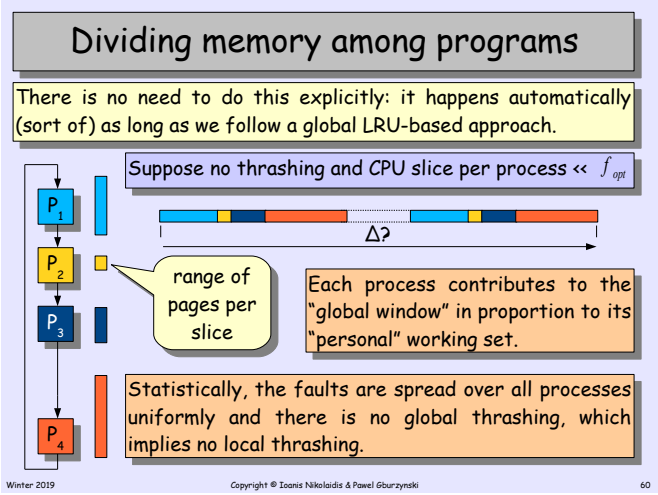


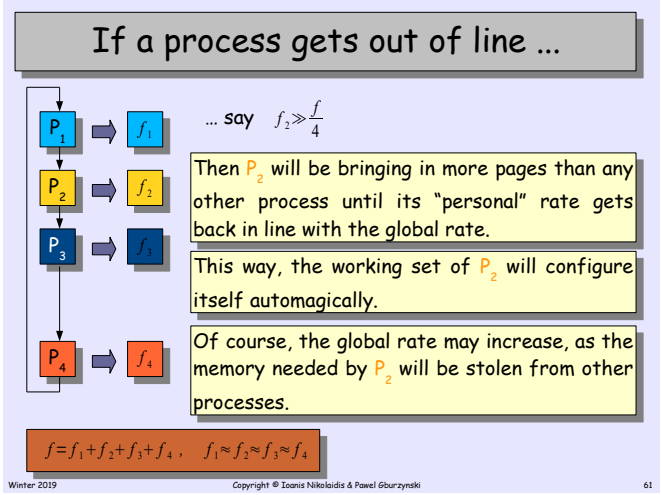
The front hand clears the "referenced" bits, and the back hand looks at them again. If the back hand finds a page with the referenced bit cleared, that page is victimized.

Originally, there was a single hand, and the strategy looked even closer to second chance.

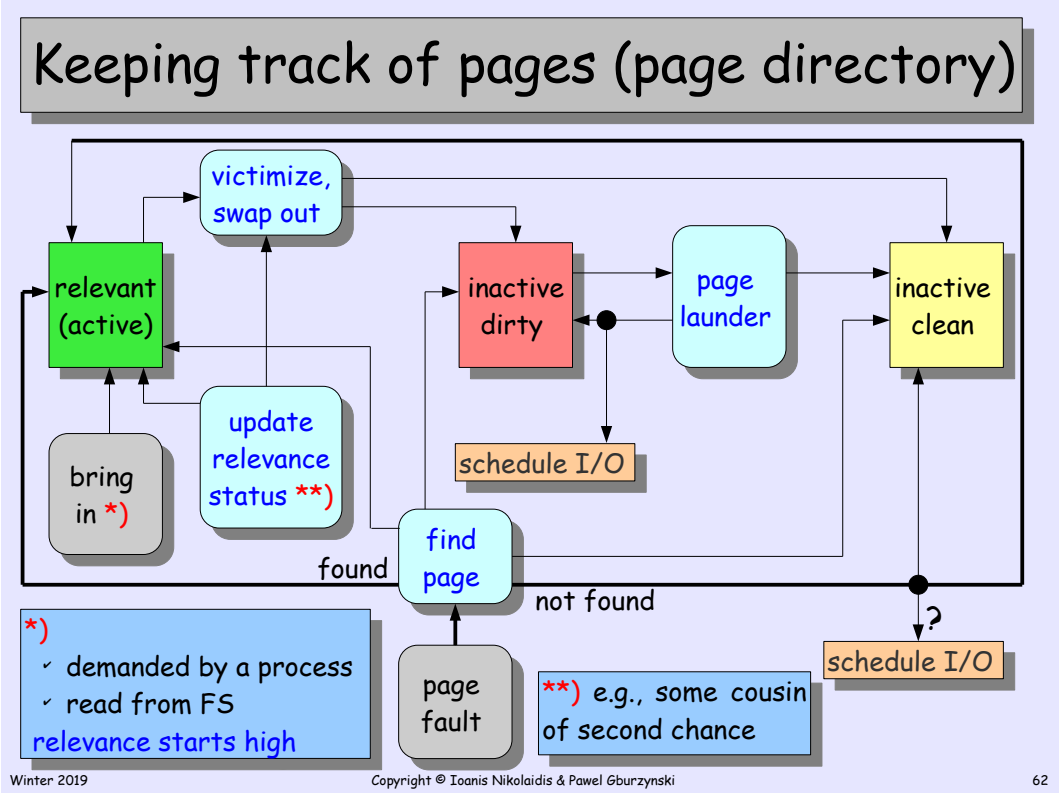
Note that the victimizer only starts when freemem < lotsfree, which means that it may start in an old stale configuration. Thus, the system should refrain from victimizing any pages until the back hand looks at something that has been actually given a (second) chance.

Dividing memory among programs





Keeping track of pages (page directory)



Note that:
When a page is moved from Active to Inactive (dirty or clean), it doesn't have to be unmapped from its process's address space! It will be unmapped when actually "grabbed" (i.e., re-cycled). Migration to states Relevant and Dirty happens automatically upon reference.
For as long as the page is not actually re-used (i.e., filled with something different) the tag reflecting its current content is stored in the page directory. Thus, the page can be found when demanded and easily "un-victimized".
The act of victimizing a page is never such a simple (one-step) operation as it would seem from the simplified models. In particular, the action of laundering a dirty page takes time during which things can happen.

Modern trends

How do we deal with 64-bit address spaces? How to implement page tables for them?
Lets do some calculation:

Let	
A	be the number of address bits (e.g., 64)
l	be the number of bits in a page offset (e.g., 12)
k	be the number of levels
$u = (A - l)/k$	# of address bits per level
2^u	# of entries in a table
E	is the size of one entry (8 for a 64-bit address)
$E \times 2^u$	the size of a table is
$1 + 2 \times (k - 1)$	total # of all tables

The overhead

$$O = E \times (1 + 2 \times (k - 1)) \times 2^{A-l/k} \tag{4}$$

Some examples

A=64, l=12, E=8	
k	overhead
1	3.6x10 16
2	1.6x10 9
4	470K
6	35K
8	10K
10	5K
12	3.5K

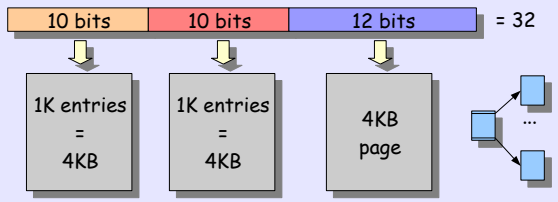
A=32, l=12, E=4	
k	overhead
1	4.2x10 6
2	12K
4	896

A=64, l=14, E=8	
k	overhead
5	72K
4	320K
6	28K

Of course, not all possibilities are realistic, but the trade-off is clear. Now, would you really want to have 5-level tables?

Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 66

Consider x86



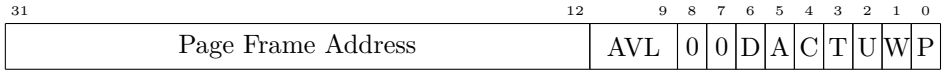
Overhead for the extremely hollow address space is 3 tables, i.e., 12 KB.

This is good for up to 4GB of logical address space and up to 4GB of physical memory. That used to be a lot, but isn't any more.

Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 67

Going beyond 32-bits

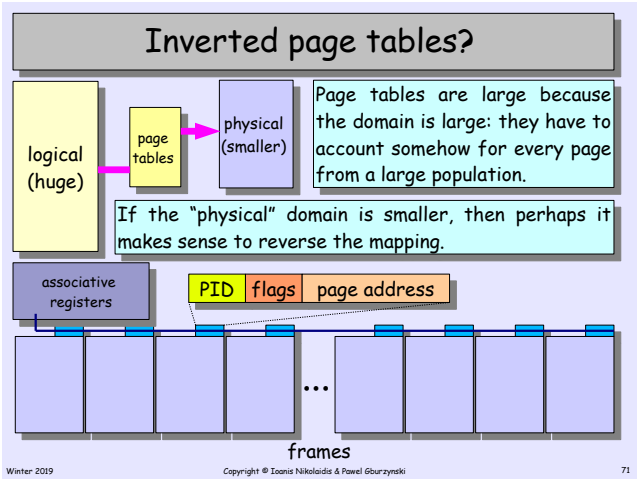
The physical memory limit is determined by the PDE/PTE (page table entry) format, and this is also the first problem to overcome. Here is the page table entry for a 32-bit system.



The lower 12 bits were used for attributes for the page table entry.

To expand the memory address range we can us a Page Directory pointer that points to the the page directory we want to reference. From there it just another multi-level page table. This however can get out of hand as you are increasing the level of re-directions that the computer needs to handle for each reference.

Inverted page tables



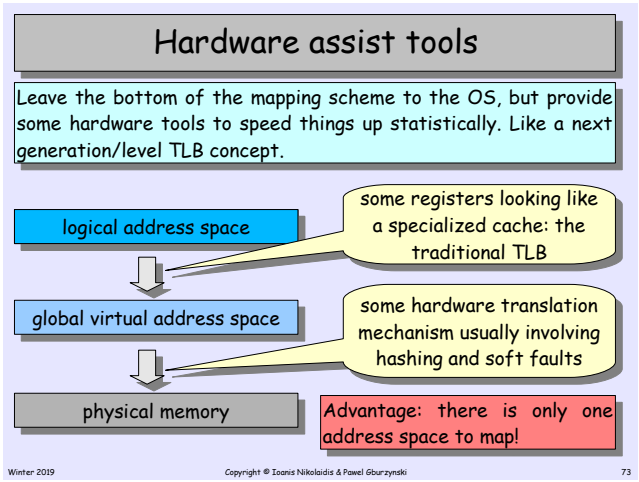
What about having no tables at all

The worst problem with inverted page tables is that they make memory sharing difficult. Remember the TLB. It resolves the vast majority of all references without resorting to the tables at all. So this is the idea:

> Make TLB misses perceptible by the kernel (as correctable faults), and let the system take care of filling the entries. The system can implement the mapping the way it pleases. Instead of hardware mapping we use software.

The cost of forcing the hardware to go through five levels of tables may be comparable to the cost of doing something smarter entirely in software.

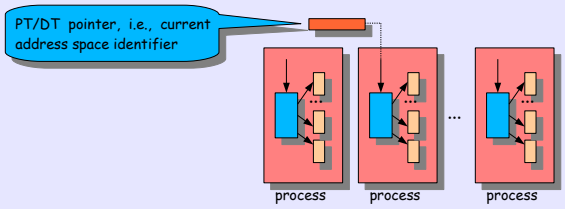
Hardware assist tools



PARADIGM SHIFT

The paradigm shift

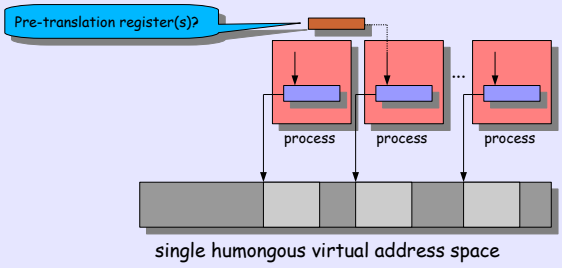
With the traditional approach, different sets of tables are used for each logical address space.



Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 74

But once we agree that ...

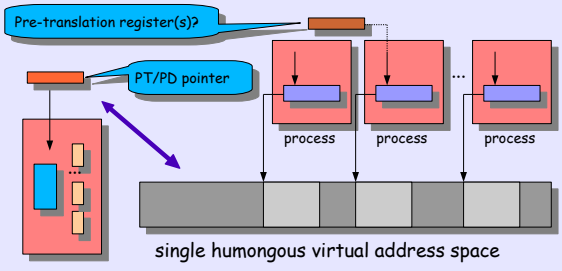
... the virtual address space is humongous (and the traditional approach doesn't work any more), we may as well do it (conceptually) like this:



Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 75

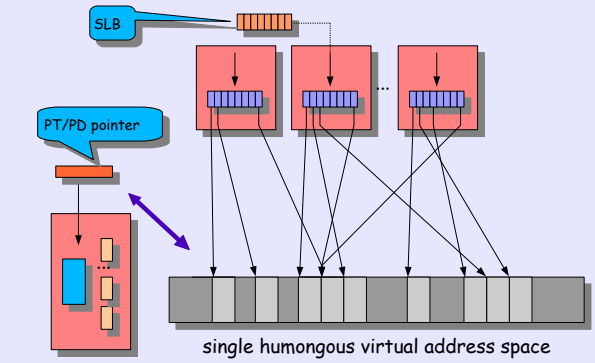
But once we agree that ...

... the virtual address space is humongous (and the traditional approach doesn't work any more), we may as well do it (conceptually) like this:



Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 76

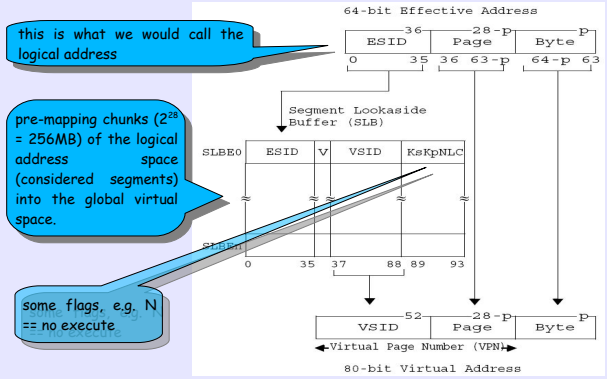
The conceptual view



Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 79

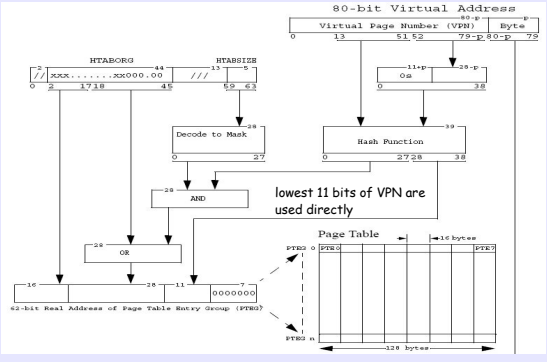
POWER PC

PowerPC address translation



Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 78

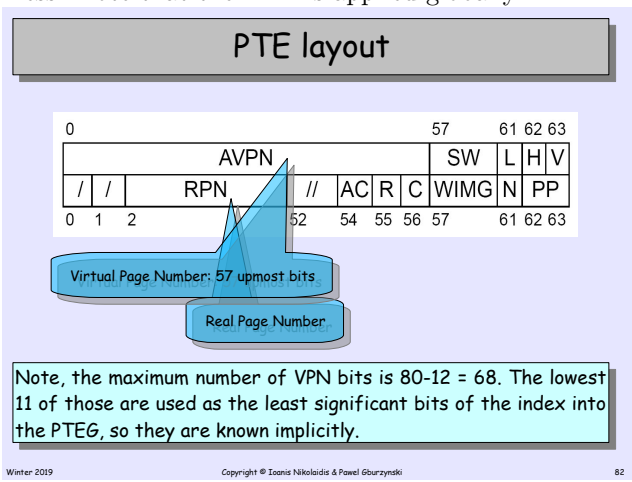
Now for the virtual to physical part



Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 80

The rules

The hash function produces two PTEG (Page Table Entry Group) pointers. In fact, two slightly different hash functions are applied. They return pointers to the primary and secondary PTEGs.
If the page is found in the primary group, the frame address is taken from there, if not, the secondary group is scanned. If the page is not found there, we have a fault.
A single group contains descriptions of up to 8 pages that may hash to the same value.
Of course, there is a (traditional) TLB (applied after the SLB), which statistically circumvents all this mess. Note that the TLB is applied globally.

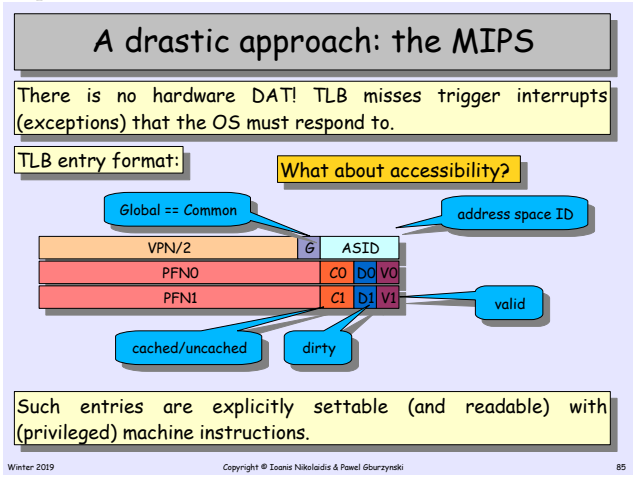


How much does it cost. The recommended size of the hashed page table is half the total number of real pages to be accessed (i.e., that many PTEGs).

The penalty, if the table is too short for a 4 GB real memory and 4 KB page sizes, is about 1.6 percent of all memory is reserved for the PTEG which is pretty good, considering that the table is global

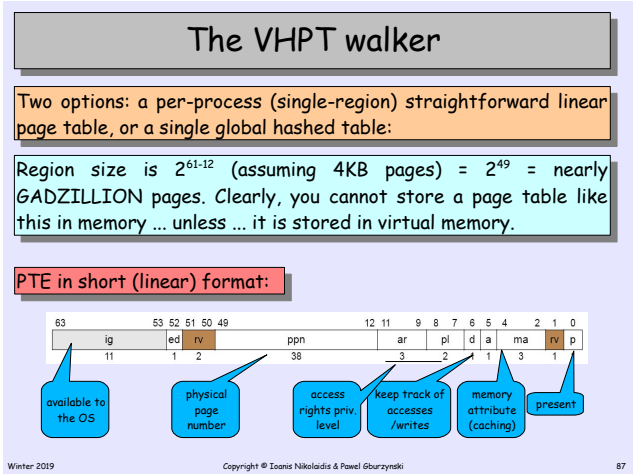
MIPS

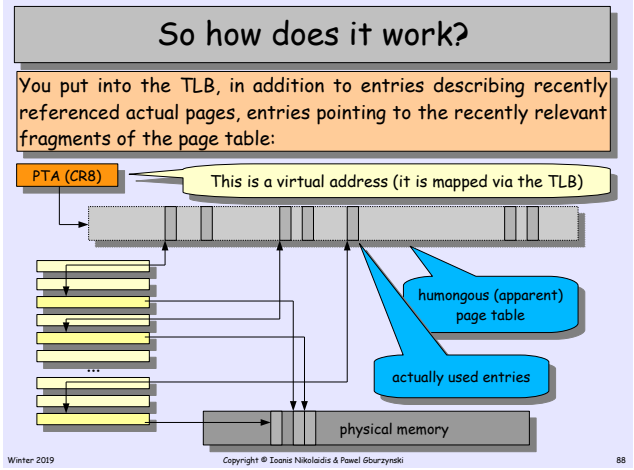
MIPS does not have a hardware DAT! TLB misses triggers interrupts (exceptions) that the OS must respond to.



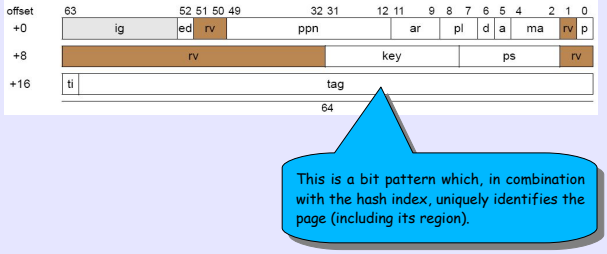
IA-64

The DAT is optional on IA-64. The architecture allows the use of the VHPT walker.





The hashed variant: PTE



This is a bit pattern which, in combination with the hash index, uniquely identifies the page (including its region).

In contrast to PPC, the walker examines only one entry (the one directly pointed to by the index). If the page doesn't match, it triggers a fault, and the OS will have to take over.

Superpages

Superpages

Some contemporary architectures (including PPC and IA) allow you to have pages of variable size. This means that TLB entries can describe physical memory chunks of different sizes, e.g.,

A large continuous range of physical memory can be described with much fewer entries. This means:

- reduced size of tables
- more TLB hits

But also more burden for the OS: promoting/demoting pages, fragmentation.

11: File Systems

A file is an identifiable collection of information, typically stored on a non-volatile medium for safe archival and retrieval.

Tape-based file system

In the beginning each file is stored on a separate tape. Physical labels where used to label files.

Problems

- **Poor utilization of tapes** Either every tape must be large enough to accommodate the largest conceivable file, or we have to deal with tapes of different sizes and force the program to specify the needed size when it creates a file.
- **Poor protection.** what if the operator mounts the wrong tape?
- **Files are sequential.** No direct access. Difficult to update file "in-place".
- **inconvenience.** You had to physically load each file by hand.

There where some protections methods like physical ring locks or **logical vsn label** on tape.

Blocking

Practically all storage media require information to be blocked, but not always for the same reason. Tapes would have gaps between blocks so that the read head can stop before the start of the block so that the inertia doesn't go too far ahead. In this case a larger block leads to better tape utilization.

Disks

The evolution of tapes. Disks are also blocked. A single physical block (sector) is the minimum unit of information that can be transferred (and addressed).

Surface mapping

P	the number of platters
2P	is the numbers of heads
C	the number of cylinders
T	the number of sectors per track
$S = 2PCT$	the number of sectors
i	the sector index of the disk from 0 to S-1
$c - i/(2PT)$	is the cylinder number
$u = i \mod T$	is the sector number within the track
$h = (i/T) \mod 2P$	is the head number

Hopscotch

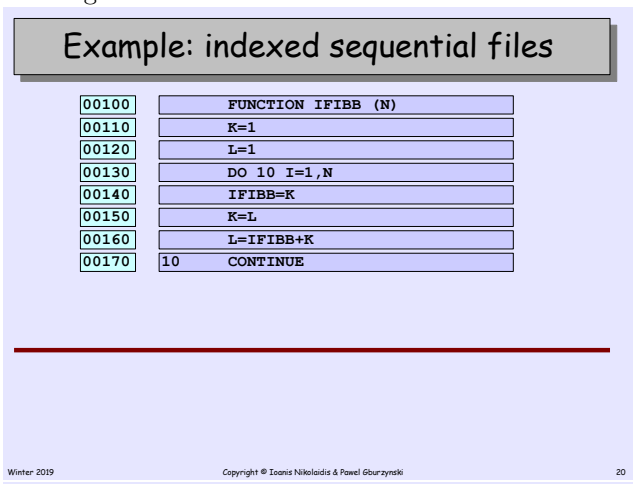
0	1	2	3	4	5	6	7	8
0	5	1	6	2	7	3	8	4

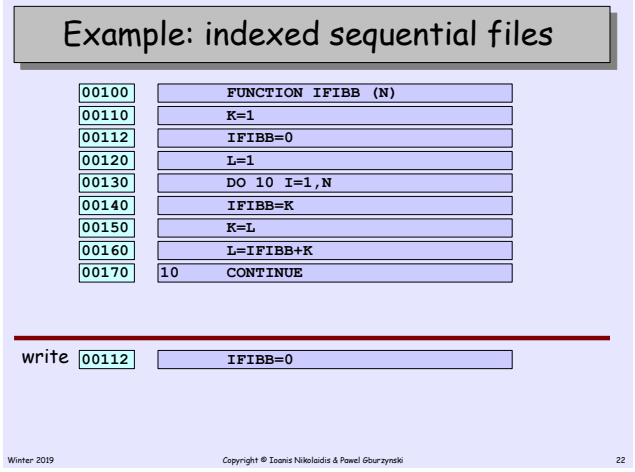
Sometimes the read would be staggered as each block will undergo some error checking and correcting so the hopscotch will stagger the data such that by the time the error checking and correction is done for the current block, the next block the head is passing by would be the next block in sequence.

Design issues in a file system. The user-visible part: file id and location (hierarchy). operations on files, efficiency, reliability, resilience, integrity. Internals: organization of disk storage, caching, integration with virtual memory, reliability, resilience, integrity.

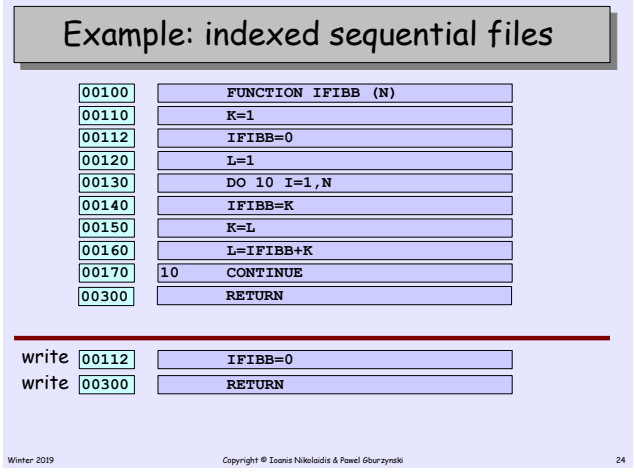
Index sequential files

Starting from this file.

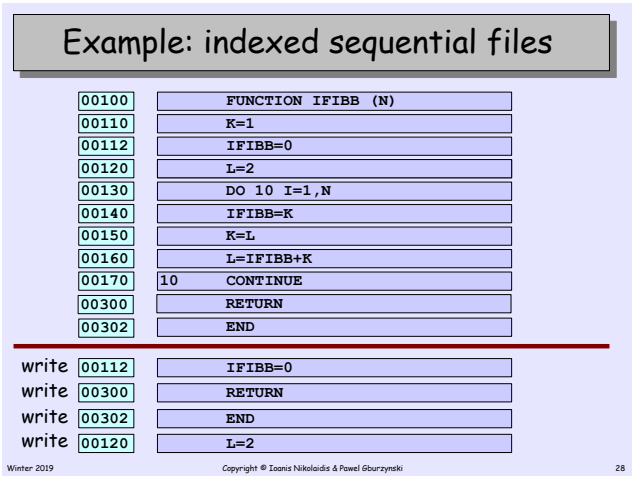




We wrote IFBB=0 to index 00112 in the file which doesn't exist so we wrote it in between K=1 and L=1.



We wrote RETURN to index 00300 which does not exist so we created a new entry at the end.



We wrote L=2 to index 00120 which does exist so we overwrote the previous data on index 120

These files were very popular on early time-sharing systems as a standard representation of text for editing. A file like that was immediately accessible as a straightforward sequential file (with the ordering of records determined by the number sequence). At the same time, the very same file was directly editable. Today, "modern" editors don't modify the files directly but store changes in the buffer which is then written to the file when you save.

Direct access files

They have functions where they can directly access files based on a key.

```
1 write(file,key,record);
2 read(file,key,record_buffer);
```

Where key was numerical or textual, fixed or variable length. The record length was fixed or variable, etc. Early systems often tried to implement as much of them as possible internally.

Representing files

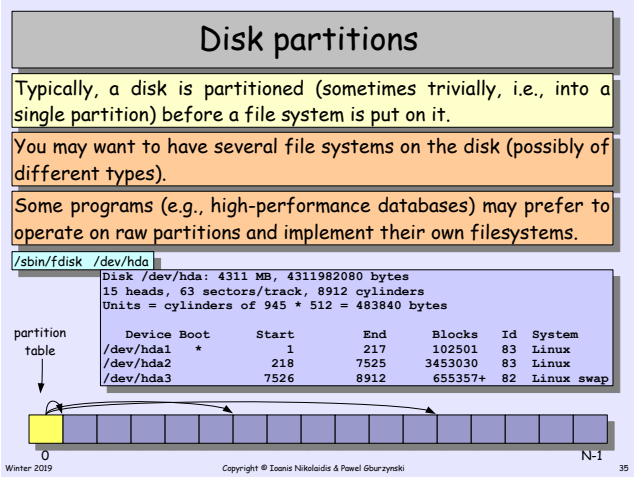
A disk can be envisioned as a simple array of sectors. Every sector is directly accessible, independently of other sectors. Files are sequences (or sets) of sectors described in some data structures. Ideally, those data structures should facilitate.

- fast location of subsequent sectors for sequential access
- fast location of any sectors for random access
- efficient file growth and shrinkage
- efficient usage of disk storage - no fragmentation
- reliability: low risk of serious damage from minor malfunctions

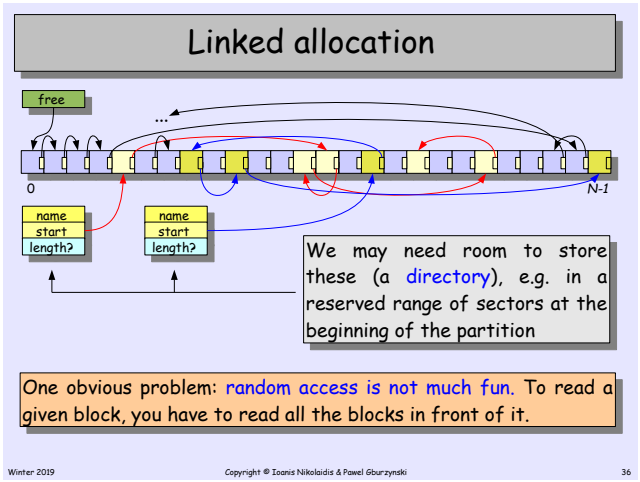
Continuous allocation

You have a single contiguous segment for a file.
This makes it easy to describe a file, have fast access, and allow fast access to both sequential and random ways.
However, the file size should be known in advance, it may be difficult to grow a file, and fragmentation.
As naive as it appears, this approach is quite popular in some applications. A program may receive a disk partitions to implement its own "file" covering a consecutive range of sectors.

Disk Partitions

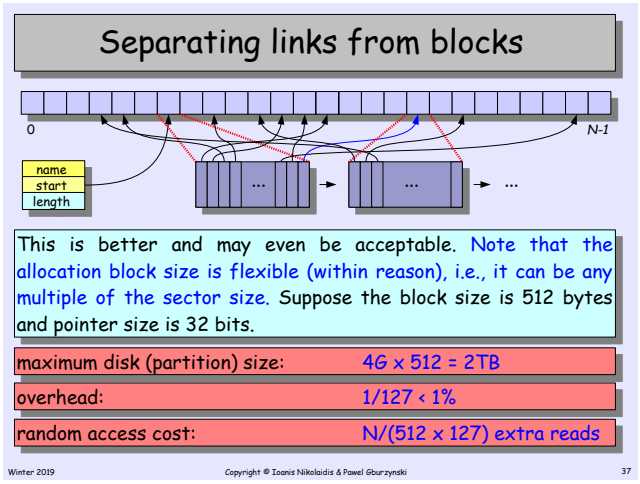


Linked allocation

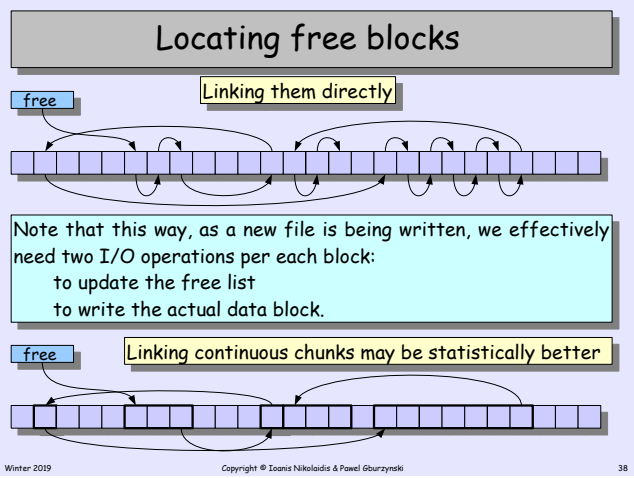


great way to solve fragmentation but has no direct or random access, has poor sequential access.

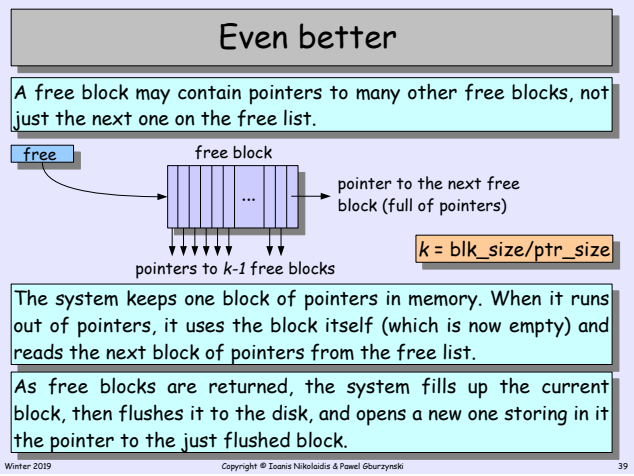
Separate links from blocks



Adding a level of redirection allows us to reduce fragmentation by having elements within the head of the file to contain offsets to blocks of data. Note: these blocks may not be in order but are called in order by the sequential calling of the page offset entries in the head page of the file. 512 is the sector size as well as the block size.



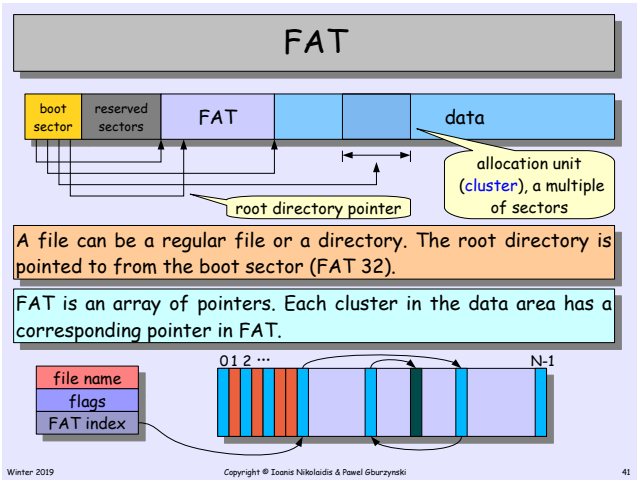
The head block of a free sequence of blocks contains the address to the next group of free blocks as well as the length.



Bitmap

Each allocatable block has one status bit in the bit map. If the status bit is 1, it means that the block is empty (or perhaps full). The bit maps must be eventually written to the disk, but the system can cache them in memory. Otherwise, a block acquisition/release would always require an extra I/O operation to update a bit in the map.
However, there are trade-offs: Block size is reduced which leads to less fragmentation, but more complexity and higher access time. And Sync intervals which leads to less overhead, but a higher likelihood of a disastrous damage.

FAT

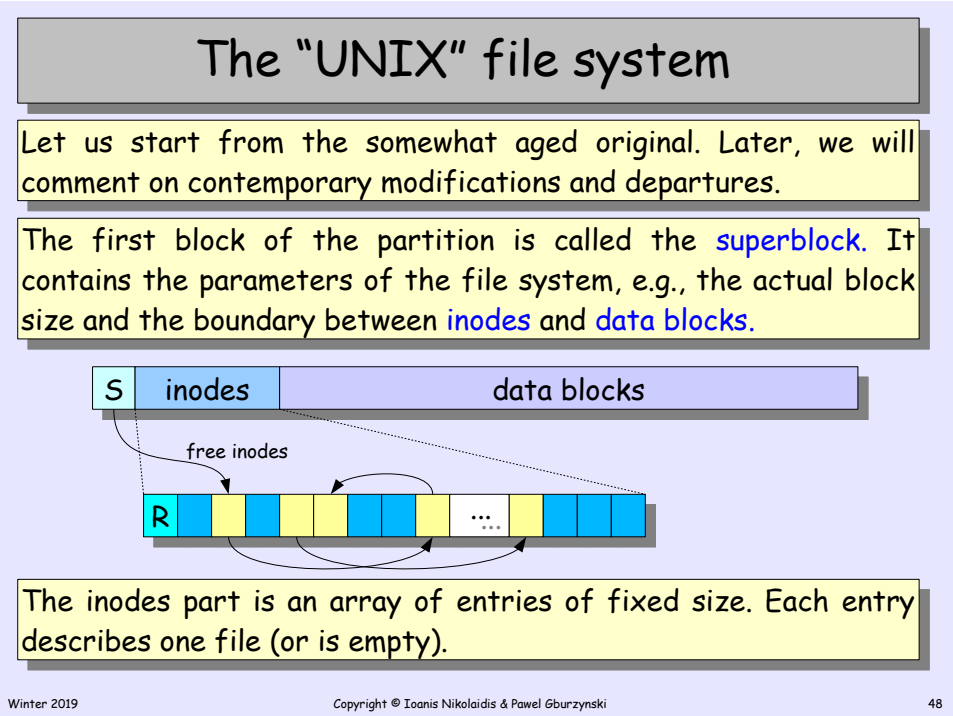


What is the best block size

This statistic is outdated but in 1993 the average file size is 22k.

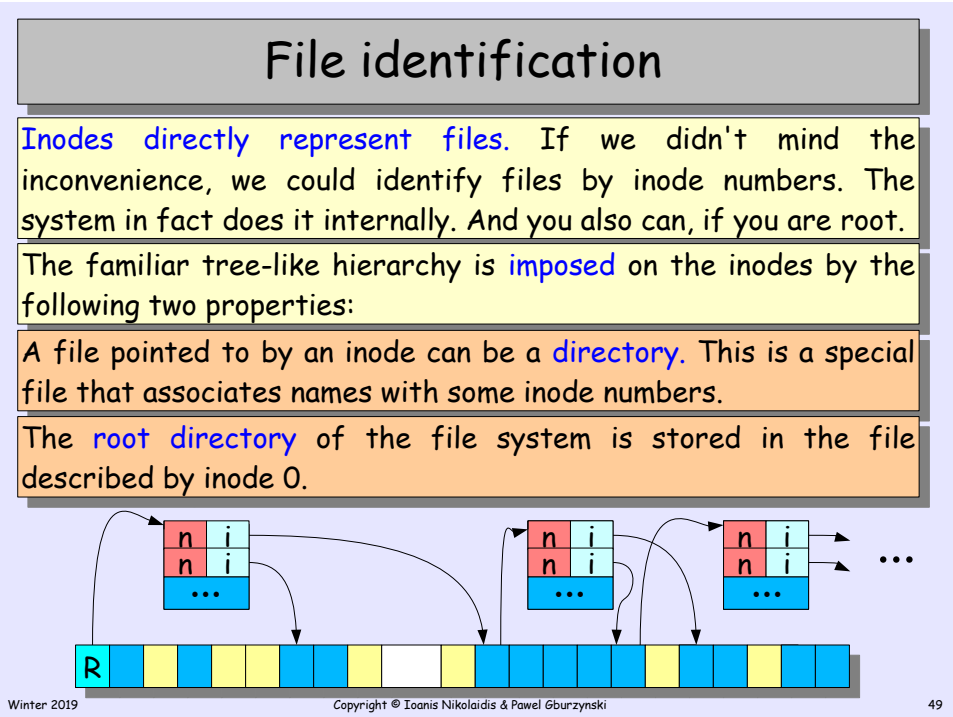
Most random files were smaller than 2k, pick a random byte and it probably in a file larger than 512k.
In general 89% of files take up 11% of the disk space, but 11% of files take up 89% of the disk space.

Today, the average file size is much bigger.



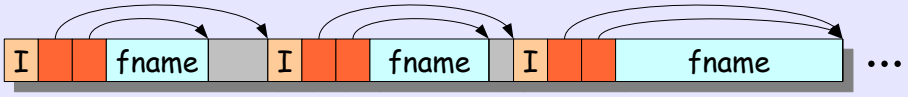
Whenever a file is opened, its position is set to byte number zero. Whenever k bytes are read or written, the file position is advanced by k. Random access is implemented by making it possible to change the file position explicitly.

FILE IDENTIFICATION



Directory structure

A directory entry contains precisely two items of information: file name + inode number. The entries are linked:



Note that some fragmentation problems may arise as files are created, destroyed, and the directory entries are recycled.

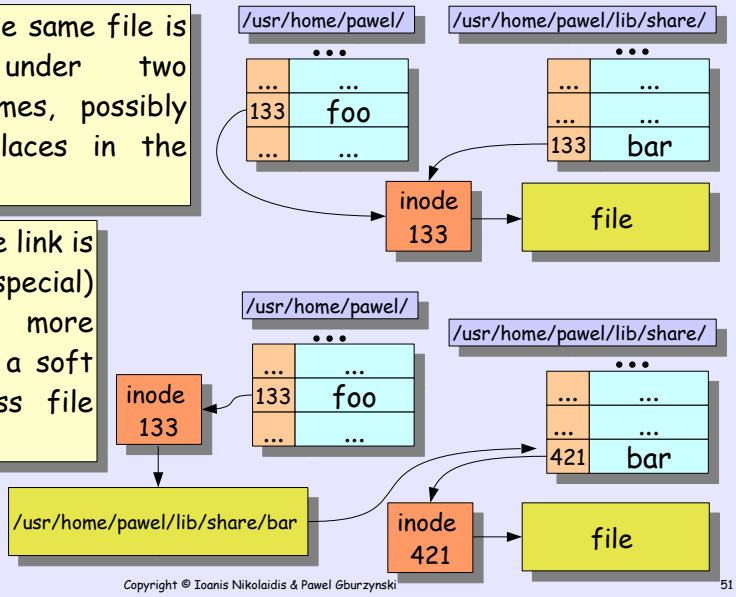
The important point is that a directory entry contains no information about the file. All that is kept in the inode. This brings about a few interesting features.

Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 50

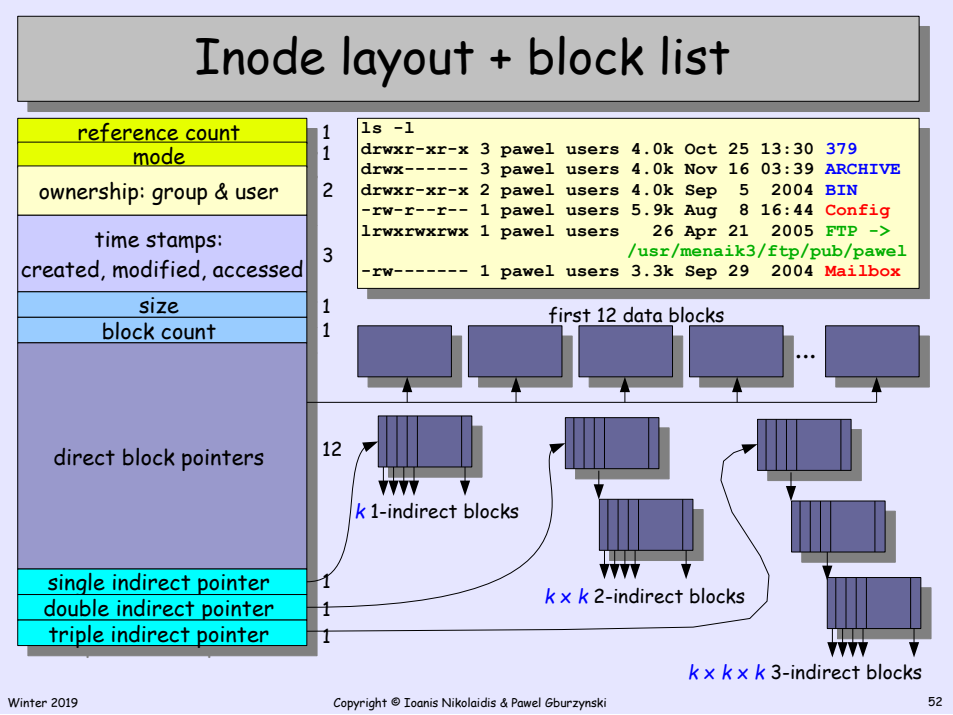
Links

Hard links: the same file is accessible under two different names, possibly at distant places in the hierarchy.

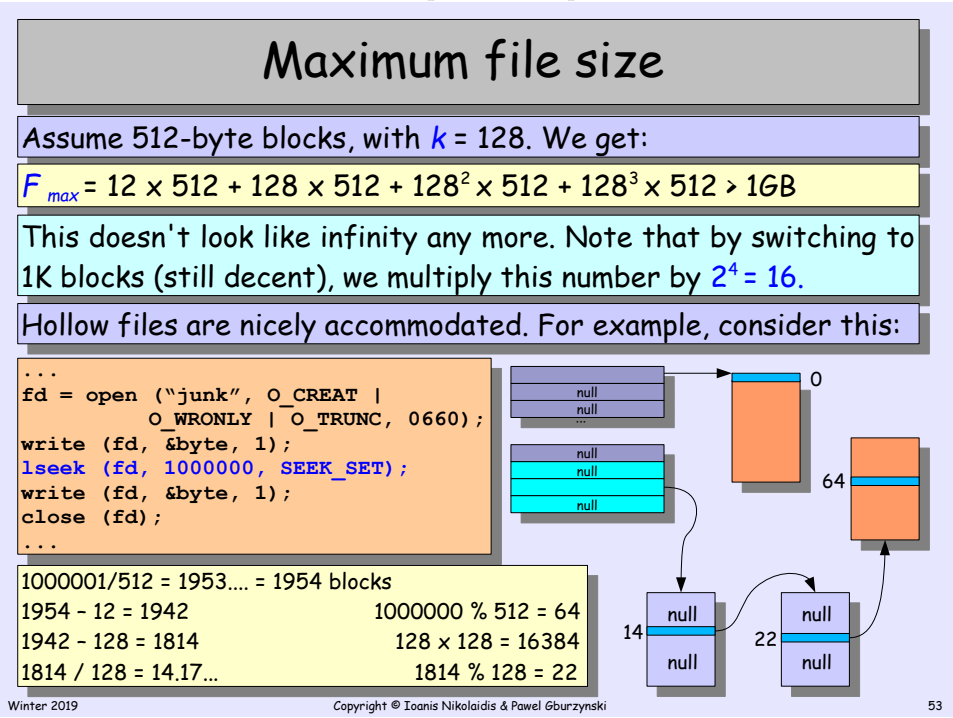
Soft links: the link is stored in a (special) file. This is more flexible, e.g., a soft link can cross file systems.



Winter 2019 Copyright © Ioanis Nikolaidis & Pawel Gburzynski 51



We can have multiple indirect pointers blocks



Free space management

The original system used a free list where you have a list of fixed sized arrays of size k-1 elements which are pointers to free blocks. The head of the list is stored in the superblock. When the file-system was mounted, the first free block was read into memory, providing immediate access to (up to) 127 + 1 free blocks.

Dual block size	Low overhead for large files, low fragmentation for small ones
Grouping	Splitting inodes into chunks and replicating the superblock
Bitmaps	instead of free lists. They are also grouped
Journaling	Neat tricks to practically eliminate the possibility of a serious damage when somebody putts the proverbial plug.

We really want small blocks to reduce fragmentation. However, larger blocks reduce the overhead on disk transfers, and thus improve the performance of heavily I/O bound programs like databases.

Dual block size

1 block = k fragments, e.g., 4KB blocks, each consisting of four 1K fragments. Bit maps are used for representing block status. Individual bits are applied to fragments rather than blocks, for example.

blocks	0	1	2	3
fragments	0-3	4-7	8-11	12-15
blocks	0000	0011	1100	1111

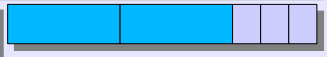
Suppose 1 means free. Only block 3 is free. Block 0 is used entirely. Blocks 1 and 2 are fragmented. They cannot be used as blocks, but their unused fragments are available for allocation to small files

Trick.

The trick

There is no difference between a large file and a small one (in its representation on the disk). Any file consists of a number of blocks + possibly a number of fragments (never more than $k - 1$) at the end.

For example, assume that the file length is 11,000 bytes. The file takes two blocks and three fragments.



11,000 = 2 x 4096 + 2 x 1024 + 760

The trick

There is no difference between a large file and a small one (in its representation on the disk). Any file consists of a number of blocks + possibly a number of fragments (never more than $k - 1$) at the end.

For example, assume that the file length is 11,000 bytes. The file takes two blocks and three fragments.

11,000 = 2 x 4096 + 2 x 1024 + 760

The trick

There is no difference between a large file and a small one (in its representation on the disk). Any file consists of a number of blocks + possibly a number of fragments (never more than $k - 1$) at the end.

For example, assume that the file length is 11,000 bytes. The file takes two blocks and three fragments.

11,000 = 2 x 4096 + 2 x 1024 + 760

The trick

There is no difference between a large file and a small one (in its representation on the disk). Any file consists of a number of blocks + possibly a number of fragments (never more than $k - 1$) at the end.

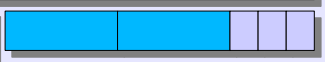
For example, assume that the file length is 11,000 bytes. The file takes two blocks and three fragments.

11,000 = 2 x 4096 + 2 x 1024 + 760

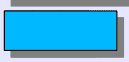
The trick

There is no difference between a large file and a small one (in its representation on the disk). Any file consists of a number of blocks + possibly a number of fragments (never more than $k - 1$) at the end.

For example, assume that the file length is 11,000 bytes. The file takes two blocks and three fragments.



11,000 = 2 x 4096 + 2 x 1024 + 760



The trick

There is no difference between a large file and a small one (in its representation on the disk). Any file consists of a number of blocks + possibly a number of fragments (never more than $k - 1$) at the end.

For example, assume that the file length is 11,000 bytes. The file takes two blocks and three fragments.

11,000 = 2 x 4096 + 2 x 1024 + 760

...

The important fact is that just by looking at the file size we know which pointers point to blocks and which point to fragments.

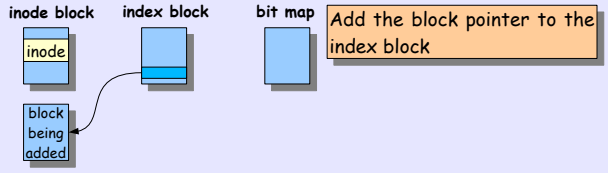
Copying can be avoided, if the system allocates consecutive fragments from full blocks.

Journaling

Suppose that a block is to be added to a file. These are the steps involved in the operation:

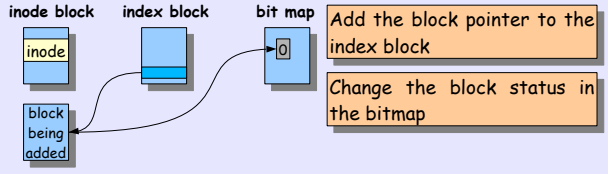
Journaling

Suppose that a block is to be added to a file. These are the steps involved in the operation:



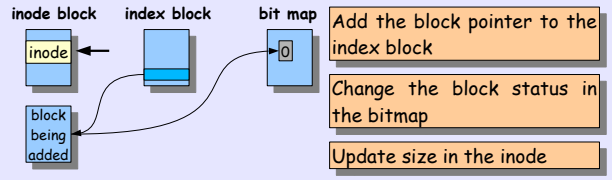
Journaling

Suppose that a block is to be added to a file. These are the steps involved in the operation:



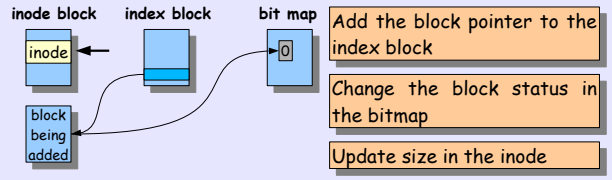
Journaling

Suppose that a block is to be added to a file. These are the steps involved in the operation:



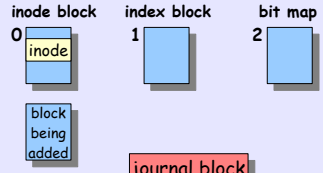
Journaling

Suppose that a block is to be added to a file. These are the steps involved in the operation:



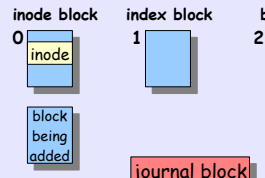
Three blocks must be written to disk to complete the operation, i.e., to make the disk structure fully consistent with the changes. If the system crashes after the first write but before the completion of the last one, we may be in trouble.

Similar to transaction roll-back in DB



To modify:
Block 1, bytes ... should be ...
Block 2, bytes ... should be ...
Block 0, bytes ... should be ...

Similar to transaction roll-back in DB

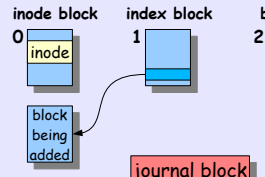


Store the journal entry

journal block

To modify:
Block 1, bytes ... should be ...
Block 2, bytes ... should be ...
Block 0, bytes ... should be ...

Similar to transaction roll-back in DB



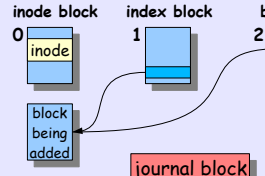
Store the journal entry

Add the block pointer to the index block

journal block

To modify:
Block 1, bytes ... should be ...
Block 2, bytes ... should be ...
Block 0, bytes ... should be ...

Similar to transaction roll-back in DB



Store the journal entry

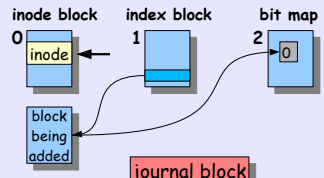
Add the block pointer to the index block

Change the block status in the bitmap

journal block

To modify:
Block 1, bytes ... should be ...
Block 2, bytes ... should be ...
Block 0, bytes ... should be ...

Similar to transaction roll-back in DB



Store the journal entry

Add the block pointer to the index block

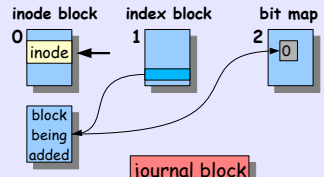
Change the block status in the bitmap

Update size in the inode

journal block

To modify:
Block 1, bytes ... should be ...
Block 2, bytes ... should be ...
Block 0, bytes ... should be ...

Similar to transaction roll-back in DB



Store the journal entry

Add the block pointer to the index block

Change the block status in the bitmap

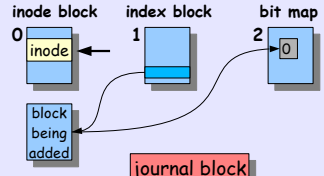
Update size in the inode

Remove the journal entry

journal block

To modify:
Block 1, bytes ... should be ...
Block 2, bytes ... should be ...
Block 0, bytes ... should be ...

Similar to transaction roll-back in DB



Store the journal entry

Add the block pointer to the index block

Change the block status in the bitmap

Update size in the inode

Remove the journal entry

journal block

To modify:
Block 1, bytes ... should be ...
Block 2, bytes ... should be ...
Block 0, bytes ... should be ...

System consistency does not depend upon a success of multiple writes to the disk. After every single write, the system either is consistent, or is able to complete the unfinished transaction.

Extents

Extents

I told you that the simple idea of continuous allocation is too valuable to be discarded as naive. Perhaps we shouldn't use it directly, but it may make sense to build files of continuous chunks (block ranges) called extents.

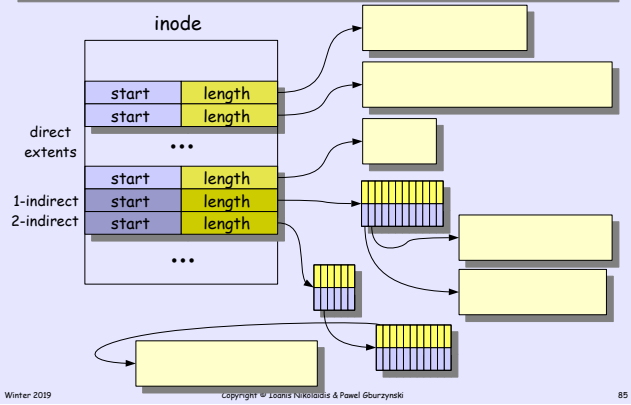
Statistically, even a large file may need just a few extents.

If long extents tend to be frequent, a file's description will tend to be short.

This means that access time to the file will tend to improve – both for sequential and random operation.

Also, describing free storage with extents may be more efficient (fewer things to keep track of, faster location of free space).

For example



Comments

Holes cause no problem: they are taken care of by empty extents with start == NULL.

One problem is the apparent complication of random access to distant blocks.

But we hope that the extents will tend to be large, so there will be few of them, so traversal of index extents will be infrequent.

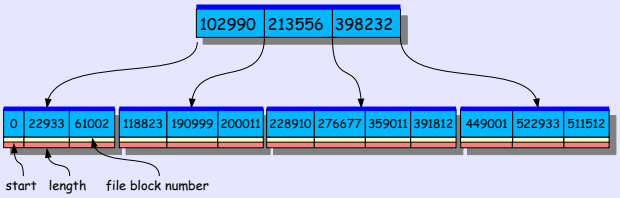
Caching will also help, assuming that the "random references" are not completely random.

Besides, there are other (more efficient from the viewpoint of random access) ways of representing series of extents than the straightforward extrapolation of UNIX indexes and index blocks.

B-Trees

B-Trees

You know what they are if you have taken a database course. It used to be a prerequisite for 379.

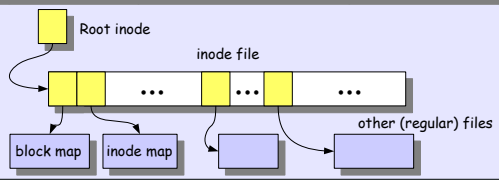


A file can be viewed as a database of extents with block numbers used as keys.

WAFL

Other ideas: WAFL

Write Anywhere File Layout. All metadata is kept in files:

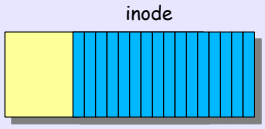


The entire file system forms a tree of blocks rooted at the single (Root) inode. This means that:

No blocks are dedicated to inodes, and there is no need to draw any lines.

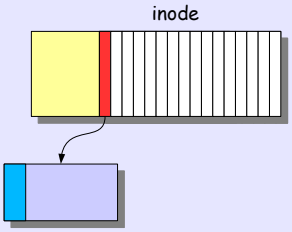
An arbitrary number of filesystems can coexist on the same partition.

All pointers in inode are the same level



There are 16 pointers in the inode. If the file is not larger than 64 bytes, it is kept in the inode. No data blocks are needed!

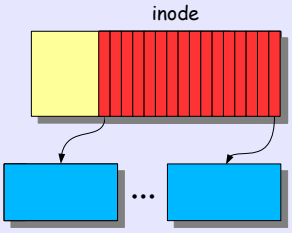
All pointers in inode are the same level



There are 16 pointers in the inode. If the file is not larger than 64 bytes, it is kept in the inode. No data blocks are needed!

When it grows beyond 64 bytes, the data is copied to a block.

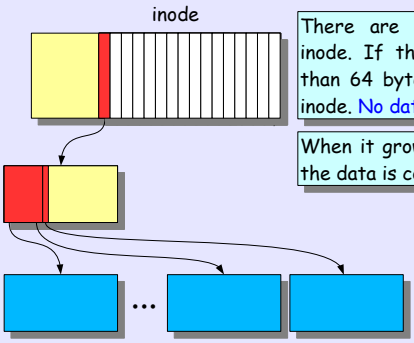
All pointers in inode are the same level



There are 16 pointers in the inode. If the file is not larger than 64 bytes, it is kept in the inode. No data blocks are needed!

When it grows beyond 64 bytes, the data is copied to a block.

All pointers in inode are the same level



There are 16 pointers in the inode. If the file is not larger than 64 bytes, it is kept in the inode. No data blocks are needed!

When it grows beyond 64 bytes, the data is copied to a block.

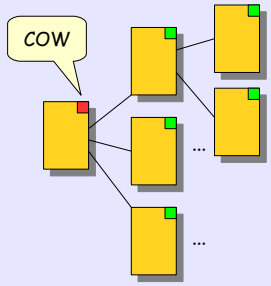
Whenever the maximum file size at the current level is exceeded, the inode pointers are copied to a new index block and a new level is started.

Snapshots

Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

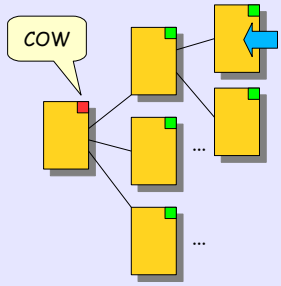
1. duplicate the root inode
2. enter a COW mode



Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

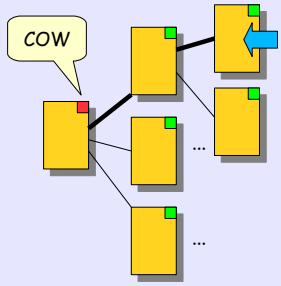
1. duplicate the root inode
2. enter a COW mode



Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

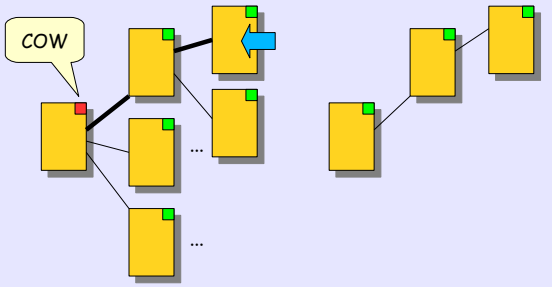
1. duplicate the root inode
2. enter a COW mode



Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

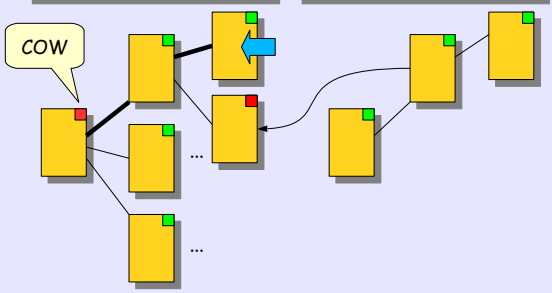
1. duplicate the root inode
2. enter a COW mode



Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

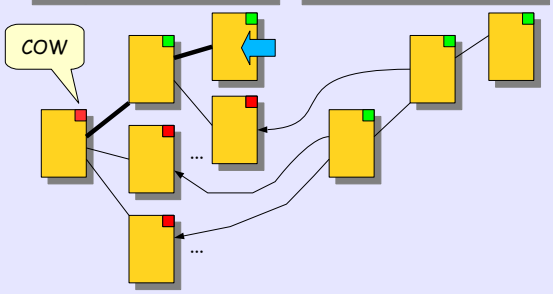
1. duplicate the root inode
2. enter a COW mode



Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

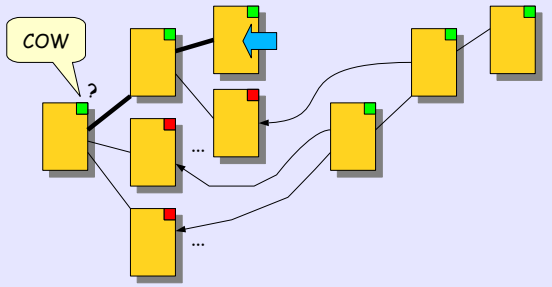
1. duplicate the root inode
2. enter a COW mode



Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

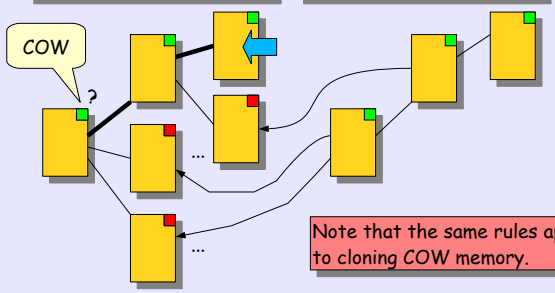
1. duplicate the root inode
2. enter a COW mode



Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

1. duplicate the root inode
2. enter a COW mode



Note that the same rules apply to cloning COW memory.

12: Disk scheduling

I/O request to disks may arrive from several places at pretty much the same time (in bursts), occasionally faster than they can be handled by the disk. This is because

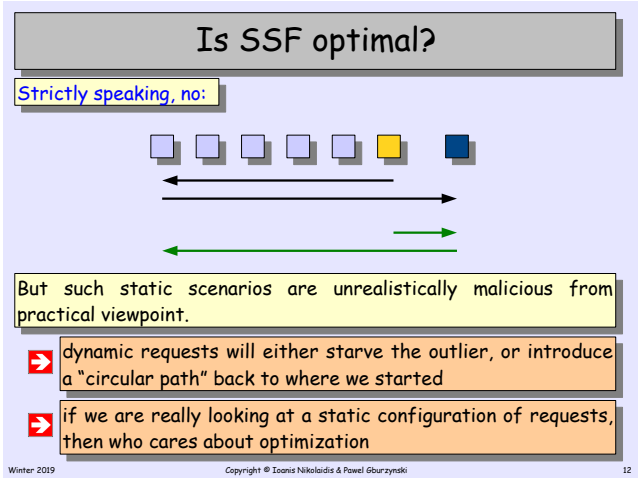
- **Multi-programming**: multiple programs doing I/O concurrently
- **Paging**: programs don't have to do explicit I/O to trigger I/O
- **caching**: I/O can happen independently of programs' intentions
- Moreover, various performance enhancing tricks like read-ahead contributes to this picture.

Seek time is the absolute time/distance it takes for the head to move from the current cylinder to the desired cylinder. The **total seek time** is the summation of these seek times.

SSF: Shortest Seek First

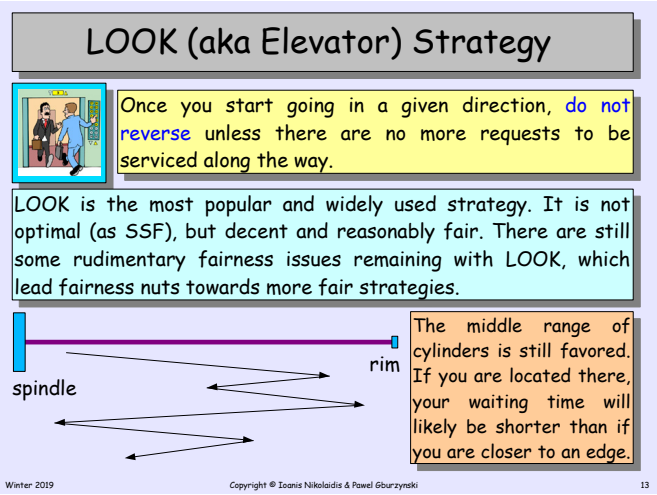
Having completed processing a request, select the next request with the shortest seek distance from your current position. This is similar to SJF, and it looks like it would be optimal but its not. It still have starvation issues and weird edge cases where it is sub optimal.

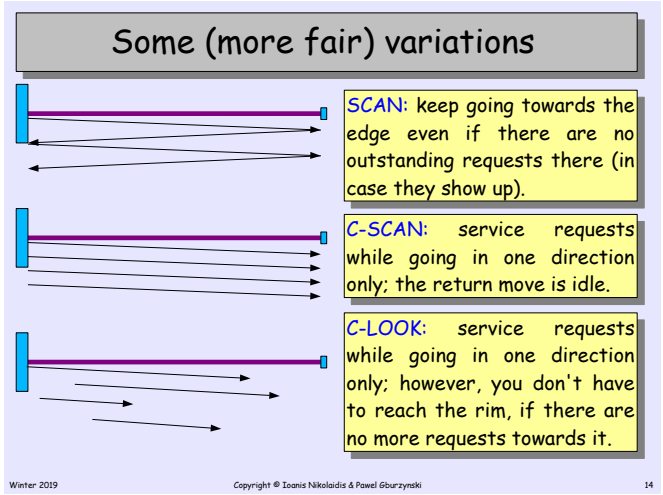
IS SSF optimal



Lets assume that the head starts reading from the yellow cylinder and moves leftward as those cylinders in the queue have the shortest SSF. However, after the most leftward cylinder finishes processing it then needs to go all the way right to process the next cylinder in the queue. If the device waited a little longer it would've seen the blue cylinder first, processed it then move leftward, decreasing the seek distance. However, this scenario are unrealistically malicious from a practical viewpoint.

LOOK strategy





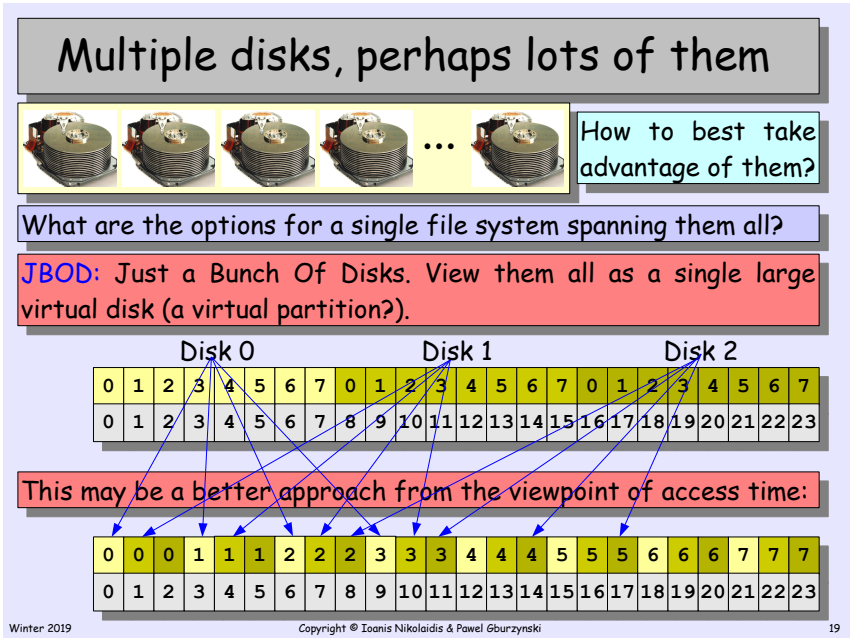
There are also other issues involved. Like optimizing other components of the disk access cost:

- The file system will try to allocate continuous ranges of blocks to files
- The hopscotch approach to numbering consecutive sectors/blocks
- Reading ahead.
- Merging multiple requests

Multiple disks in high performance systems:

- Disk arrays: for performance and reliability

Multiple Disks



RAID

With JBOD, if a single disk fails, the whole file system becomes essentially useless. It's if a hole suddenly popped up in your partition.

RAID 1

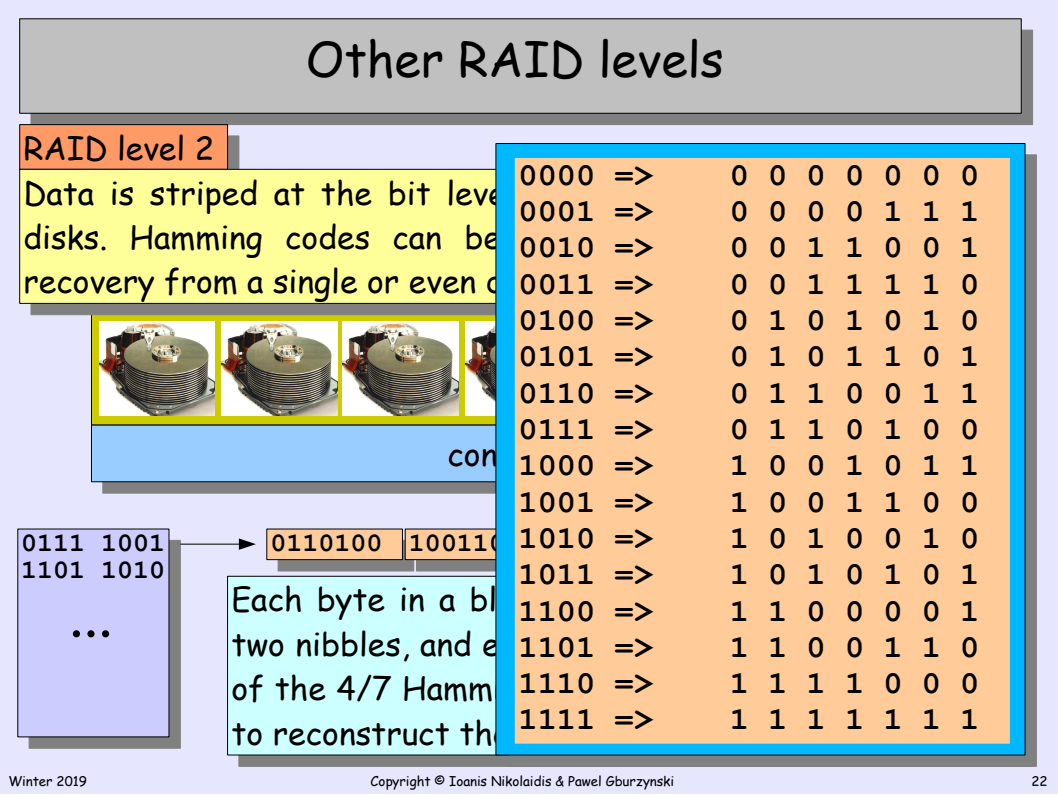
Creates a mirror on two or more disks. The multiple disks are for redundancy and reliability rather than larger size. If you have a lot of disks, you can group them into a JBOD and mirror them into a RAID 1 array.

Note that either set can be used for reading (alternately) to reduce average access time. Writing goes to the basic set, and then the mirror is updated.

RAID 2

Data is stripped at the bit level. The controller synchronizes all disks. Hamming codes can be used for error correction and recovery from a single or even dual disk failure

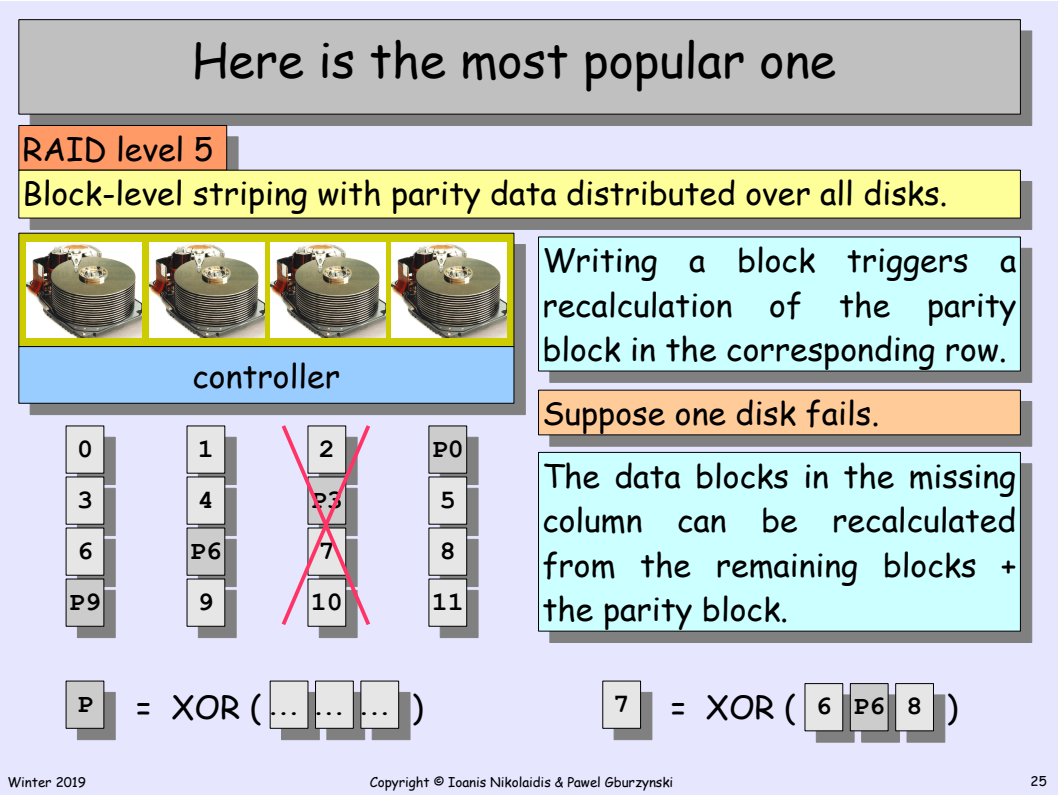
Each byte in a block visible by the FS is split into two half bytes, each half byte is transformed into the 7 bits of the 4/7 Hamming code. This allows the controller to reconstruct the half byte from any 5 of the 7 disks.



RAID 5

Block-level striping with a parity data distributed over all disks. When a disk fails, the entire array can be reconstructed using the parity data and XORing the blocks. When you are re-silvering the array however, there is a performance hit as not only you are reading parity data and doing reconstruction operations but any data that is written onto the disk will force the parity to change. Causing a slowdown in the reconstruction process.

Moreover, if the data that is being written is block 1 and block 8 in the figure below then it would take much longer than if the data was written on block 0 and 4 as you are just updating the parity on the disk an nothing else. If block 1 and 8 were updated then you are updating block 1 and P8 which involves more write operations on that disk than on the disk that has block 8.



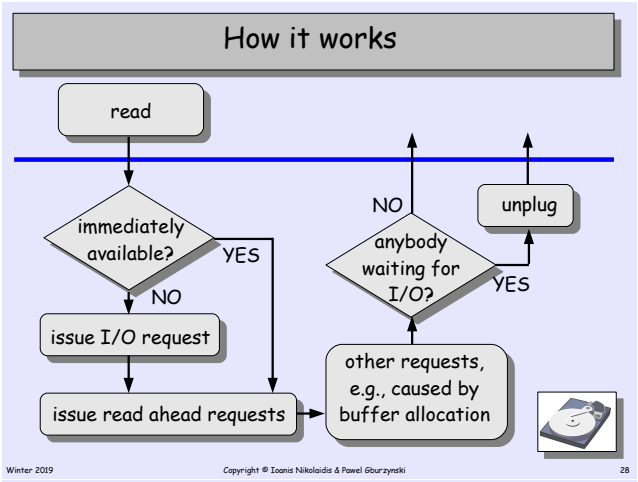
Reading ahead (for file operations)

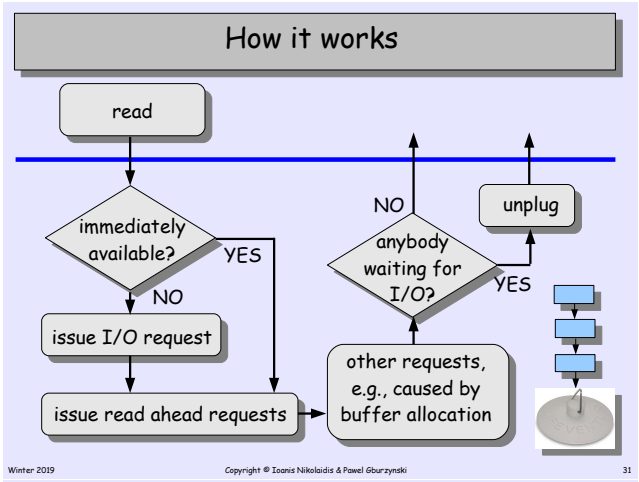
When a file is opened, the system tentatively sets the read ahead option for the file and assumes some window size specifying how much extra stuff should be read ahead after each actual read. If the subsequent read operation falls within the window, the system assumes that the read ahead option makes sense and it should remain on. Otherwise, the system assumes that the file is being read at random, and the read-ahead option is switched off. Unfortunately we can't have write-ahead.

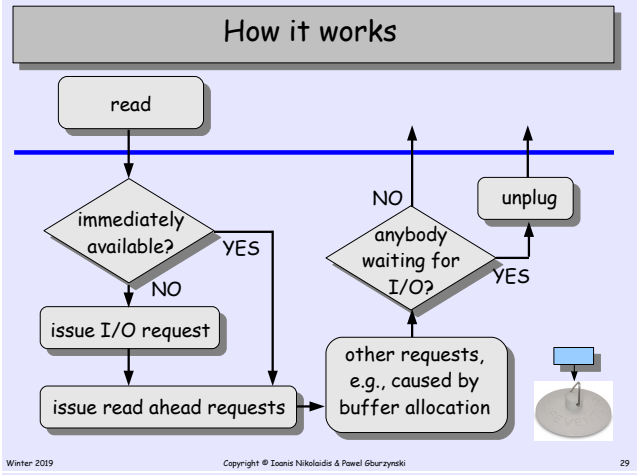
Plugging

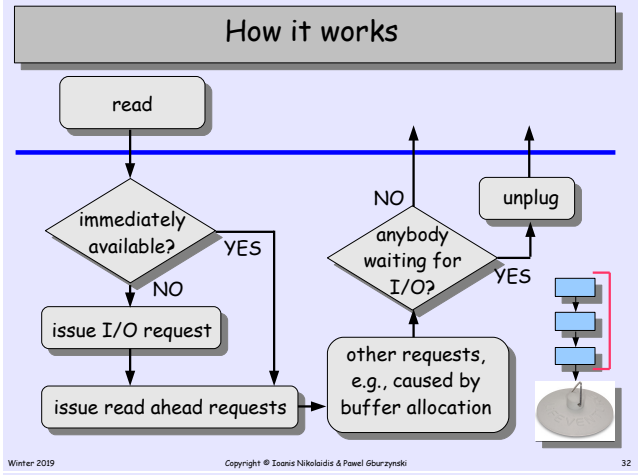
Whenever a new I/O request is added to the disk queue, the driver tries to merge it with another request already queued there. What you are able to do depends on the capabilities of the controller. Sometimes it makes sense to hold a request at the front of the queue waiting for merging opportunities

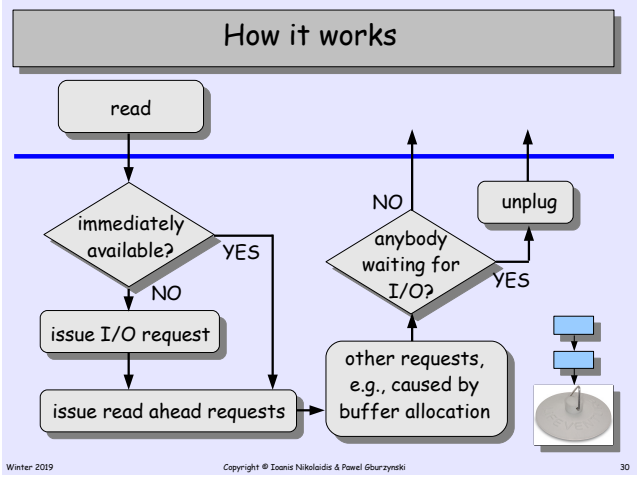
How it works

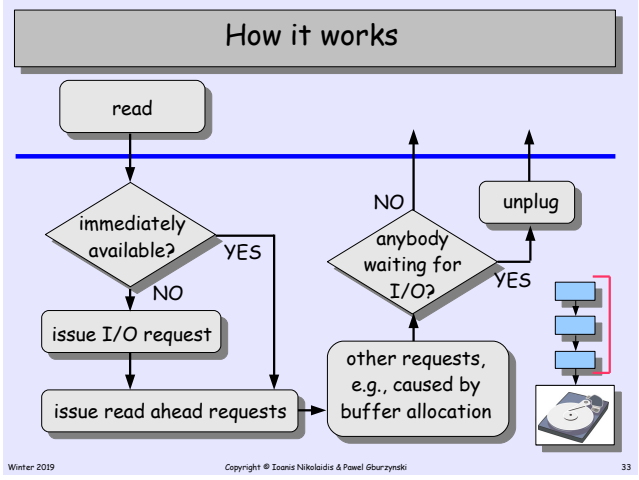












13: Virtual Machines

Virtualization is about independence from mundane aspects of reality: A program language makes programs independent of hardware, but still depend on libraries and syscalls.

You can have multiple OSes running on a machine either by dual booting or by running within a host OS.

Soft Virtualization

Java is a great example of a soft virtualization as you convert the source code from Java to its bytecode that can be executed on a JVM. The Java bytecode does not need to be recompiled again as the Javabyte code is system independent from the OS and machine. This is all handled by the JVM.

In fact interpreters are usually quite portable so it is easy to "rapidly" deploy the VM on different HW/OS platform.
Of course you can compile the Java byte code into machine code that can be executed naively on a machine for optimal performance.

Cygwin

Cygwin is a UNIX/Posix function and wrapper for system calls. Its not a perfect map as there are some syscalls that does behaves differently or does not exist at all.

Note
That the UNIX executables (compiled for the same ISA) cannot run under Cygwin (The program must be recompiled)

OS Virtual Machines

The guest OS goes though a hardware emulator that goes through the virtual machine monitor (a meta OS) that talks to the hardware.

Note
There are different types of virtualization architecture.

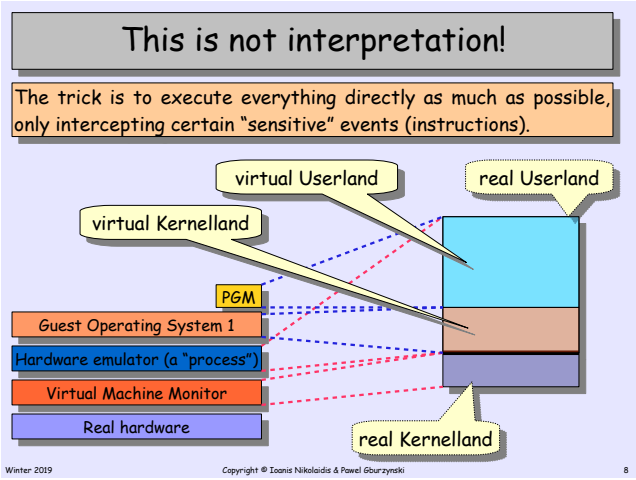
OS virtualization is useful as we can:

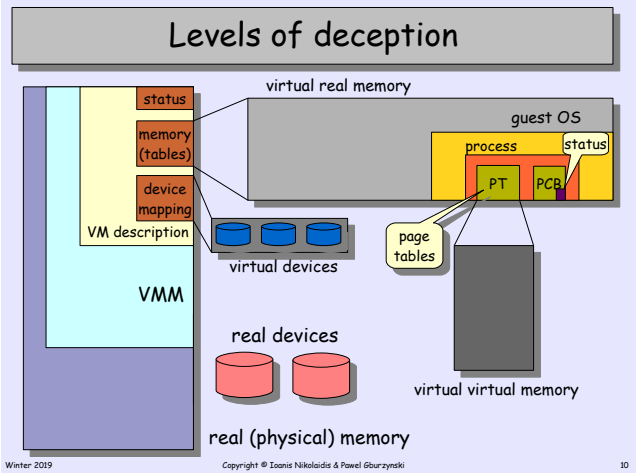
- Better utilize expensive resources more effectively, especially for multi-user systems.
- Better development as you don't need to hog the system and underutilize it
- Multiple OS as different OS have different syscalls and behaviors.
- Server consolidation and service isolation (increased security or sand-boxing)

Virtualization is not interpretation in this case as we want the guest OS to execute everything directly as much as possible with a few "sensitive" events (instructions) going towards the host OS.

Thus, the virtual computer set up by the VM monitor executes as a "process" in user space. First, note that it **has to** execute in non-privileged mode as it **cannot directly execute certain machine instructions**.

- What the guest OS perceives as "physical" memory is in fact a chunk of virtual memory setup by the monitor.
- When the guest OS think it is running privileged mode, it is in fact running in (real) user mode, with the monitor knowing that the mode is "privileged" (**virtual privileged**)
- So when the guest OS executes a privileged instructions (in its fake privileged mode), it will cause a fault. This is exactly what should happen.





The idea is very simple. When the CPU is running regular instructions, there is no need to do anything (regardless of the level). This assumes that anything that must be emulated requires an interrupt. E.g., **suppose that the guest OS wants to switch the privilege level (to user)**

1. It executes a privilege instruction (thinking it has a right to do so, as it is running in "privileged" mode)
2. But this is an illegal instruction (the OS is in fact running in non-privileged mode at the host OS level), so this causes an interrupt
3. The monitor receives it and changes the virtual privilege for the emulated machine (and emulates the necessary effects)
4. Then, it gives back the CPU to the virtual machine.

Example: System call

If a process in the guest OS executes a system call:

1. it is captured by the host OS.
2. The host OS (the virtual machine monitor) will see the syscall coming from a process which it knows to be an OS
3. bounce back the system call back to the kernel space of the guest OS. Where it will take care of it.

I/O request

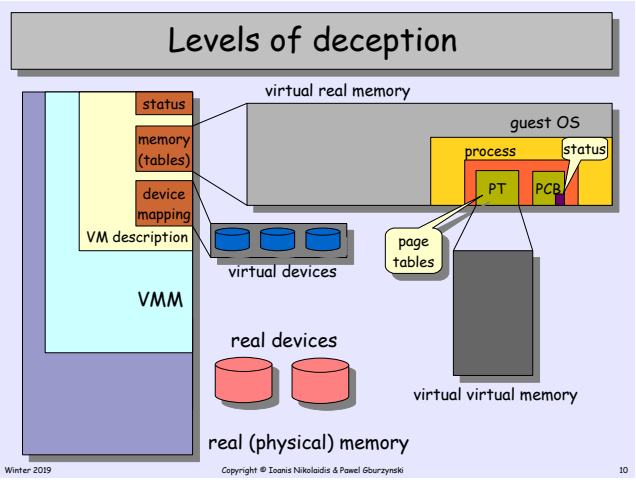
If a process in the guest OS is trying to do an I/O operation then the guest OS

1. is trying to do something "sensitive" like referencing the fake device registers, which are not formally mapped.
2. This will interrupt the virtual machine monitor which will emulate the I/O operations
3. and passes the outcome to the guest OS.
4. Eventually, the guest OS will pass the results to the process (all the time thinking it is doing the real job)

The real job is in fact being done by the virtual machine monitor, which, in particular, will emulate all the requisite interrupts in the guest OS, and so on.

Memory mapping

When the guest OS tries to create its own page table as it wants to run a new program it thinks it has access to physical memory. In reality it does not, in reality the guest OS is mapping the its logical address to the logical address of the host OS, created by the virtual machine monitor. In order to aid this translation the VMM will create and maintain a shadow table that combines the table of the two.



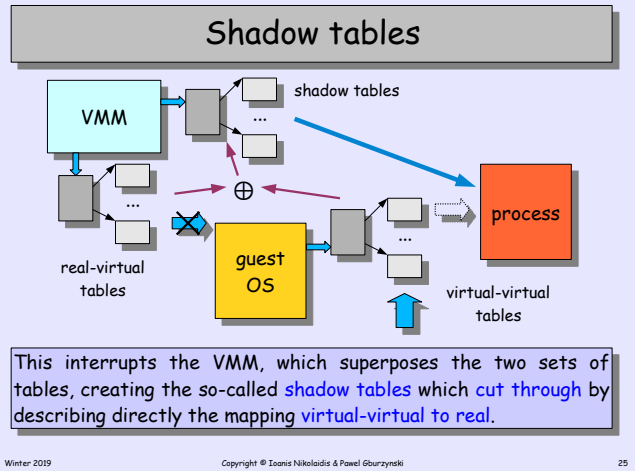


Table switching/updating

Note that when the virtual-virtual process leaves it context, there is always an interrupt which (necessarily) goes to the VMM, so it can replace the tables as required - to keep the guest OS fooled. A tricky bit is suppose that the guest OS changes something in the V-V tables, How does the VMM learn about it?
It involves the update to the respective shadow tables.
Unfortunately, updating tables involves just writing to memory (nothing "sensitive"), so the VMM is not interrupted.

One way out would be to keep the V-V tables in read-only area (the VMM knows where they are). But it isn't really needed! Remember the TLB? The guest OS has no right to assume that a change in the tables becomes effective, until it invalidates the TLB!
So long as the instruction to do that is privileged, we are in the clear.

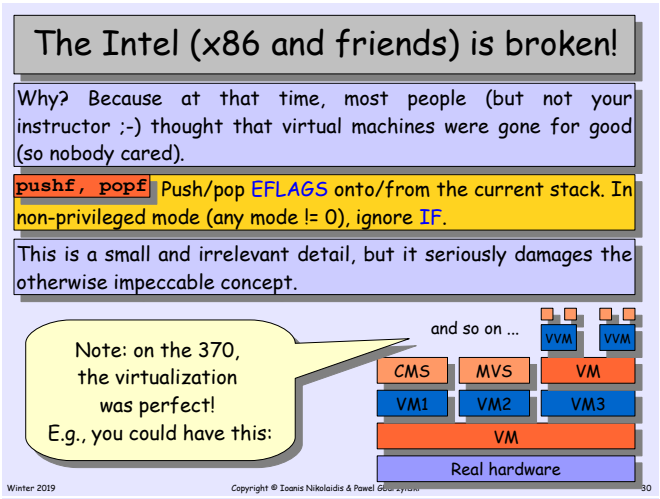
The goal of virtualizing OSes

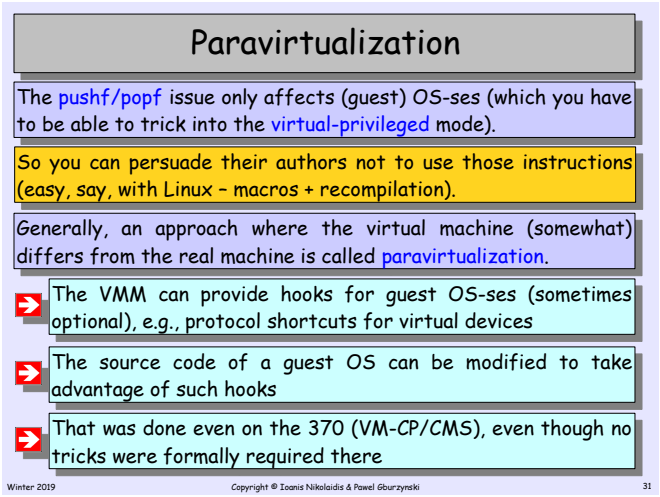
Smooth virtualization is possible if:

- All "sensitive" instructions are privileged.
- There are no instructions that silently behave in a different manner depending on the state (privileged/user). In other words: you either execute the same way in both states, or are privileged.

If you want to be formal, then you can (theoretically) emulate everything by declaring all memory "absent" and intercepting ALL instructions (at a tremendous performance hit).

Fucking Intel





Binary rewriting

Sometime the source code may not be available; then you can try to modify the binaries:

- initial preprocessing. Generally, you cannot grow the code, but you may be able to replace the opcode (with one of the unused/illegal opcodes).
- On demand. recall lazy loading. Generally, you know when a code page id demanded (and about to be executed).

Note that you need this for the OS code only. The translated code can then be cached (VMWare). Generally, the VMM will try to give preference to the pages belonging to the guest OS image, so they are not victimized (at least not too often)

Hardware assist (Intel's VT-x)

- A new state (root) added - to be used by the VMM
- Hardware data structure to keep track of the state of VMs
- Sensitive instructions cause VM exits, i.e., automatic transitions to the VMM Device virtualization hooks. i.e, devices like NICs can be "context switched" to a particular guest OS, with DMA and interrupt redirection.

Professor's Villains and Credits

C Minion

Medium Website psychic, neutral evil

Armor Class 12-15
Hit Points 100
Speed 1 Month

STR	DEX	CON	INT	WIS	CHA
29 (+9)	10 (+0)	17 (+3)	12 (+1)	11 (+0)	15 (+2)

Senses —
Languages C
Challenge 1 (200 XP)

Coverage

Everything that is being discussed in class during the existence of the minion. Including the readings

Composition

Programming 1. You create an client to client messenger using signals.

Programming 2. Creating a file deduplicator.

Programming 3. Simulating a TLB and analyzing the performance.

Midterm Champion

Large paper psychic, neutral evil

Armor Class 20
Hit Points ?
Speed 70 minutes

STR	DEX	CON	INT	WIS	CHA
23 (+6)	14 (+2)	21 (+5)	16 (+3)	13 (+1)	20 (+5)

Senses —
Languages C, Unix
Challenge 8 (3,900 XP)

Coverage

Everything from the beginning to Lecture 10

Composition

Multiple Choice. Most of the exam is MC

Short Answer. Some short answer in a specific format

Grade Slayer Tiamat

Gargantuan paper psychic, Lathrain of grades, chaotic evil

Armor Class 38
Hit Points ?
Speed 180 minutes

STR	DEX	CON	INT	WIS	CHA
30 (+10)	10 (+0)	30 (+10)	26 (+8)	26 (+8)	28 (+9)

Senses —
Languages C, Unix
Challenge 30 (155,000 XP)

Coverage

- Compiling, linking, and loading
- Processes
- Process scheduling
- Synchronization
- Deadlock
- Memory/Virtual Memory
- Disk scheduling
- File system
- VM

Composition

It going to be like the midterm but with 19 questions

Credits

General

- Created by u/DnD_Notes, April 26, 2019
- Lecture Slides by Ioanis Nikolaidis & Pawel Gburzynski
- Typesetting engine: LaTeX
- Dungeon and Dragon (5e) LaTeX Template

Art

- Figures for the cover art is from D&D Beyond
- Andy the D&D Ampersand is from Dungeon and Dragons
- Cover art formatting and design done in Photoshop CC 2019

Disclaimer