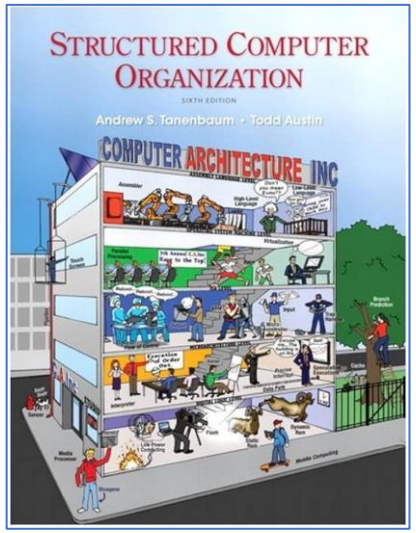
Operating Systems

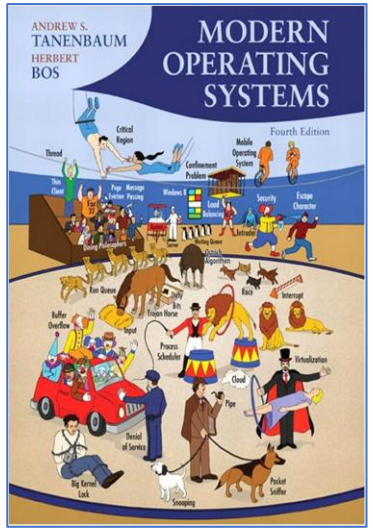
Ph.D. Ara Gevorgyan

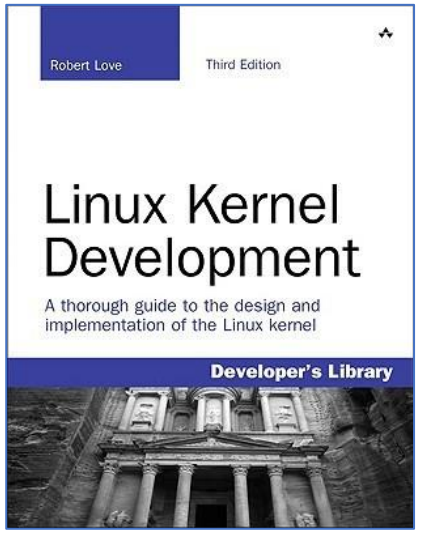
Syllabus

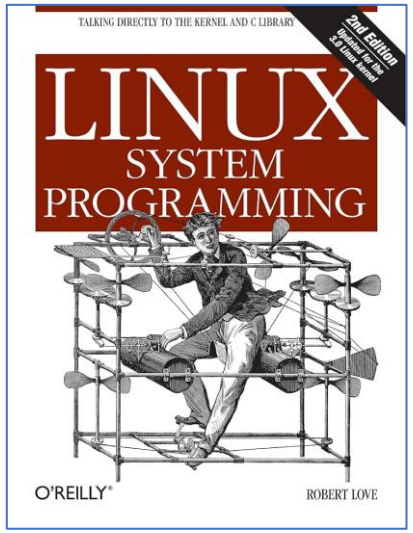
1. Computer Architecture Introduction
2. OS structure. Possible architectural approaches. OS responsibilities. Kernel space vs User space. System calls.
3. Processes. Program execution flow. Process' memory layout.
4. Process scheduling. Scheduler types and strategies.
5. Process management.
6. Memory management. Virtual vs physical memory addressing. Memory regions.
7. Inter-process communication mechanisms. Shared memory, message queues, pipes.
8. Files. File I/O. File operations.
9. Buffered I/O. Operations.
10. Concurrency and parallelism. Threading. Hardware level parallelism.
11. Concurrency and critical sections. Amdahl's law. Race conditions. Classical problems.
12. Synchronization techniques. Software approaches: Peterson's algorithm, Bakery algorithm. Spin locks. Mutexes.
13. Semaphores. Dining philosophers' problem. Deadlocks and priority inversion.
14. Network. Network communication techniques.
15. Course overview. Practicing. Q&A and preparation to the exams.

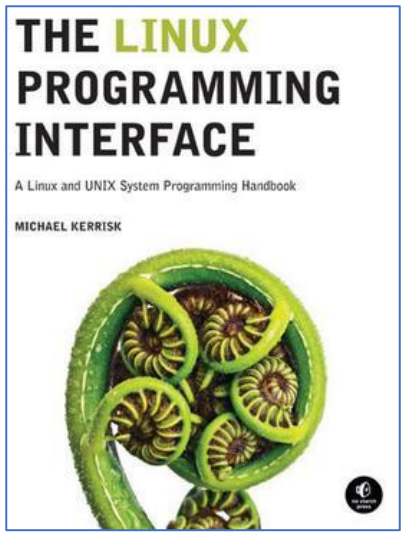
Books


STRUCTURED COMPUTER ORGANIZATION
SIXTH EDITION
Andrew S. Tanenbaum • Todd Austin
COMPUTER ARCHITECTURE INC


ANDREW S. TANENBAUM
HERBERT BOS
MODERN OPERATING SYSTEMS
Fourth Edition


Robert Love
Third Edition
Linux Kernel Development
A thorough guide to the design and implementation of the Linux kernel
Developer's Library


TALKING DIRECTLY TO THE KERNEL AND C LIBRARY
2nd Edition
LINUX SYSTEM PROGRAMMING
O'REILLY
ROBERT LOVE


THE LINUX PROGRAMMING INTERFACE
A Linux and UNIX System Programming Handbook
MICHAEL KERRISK

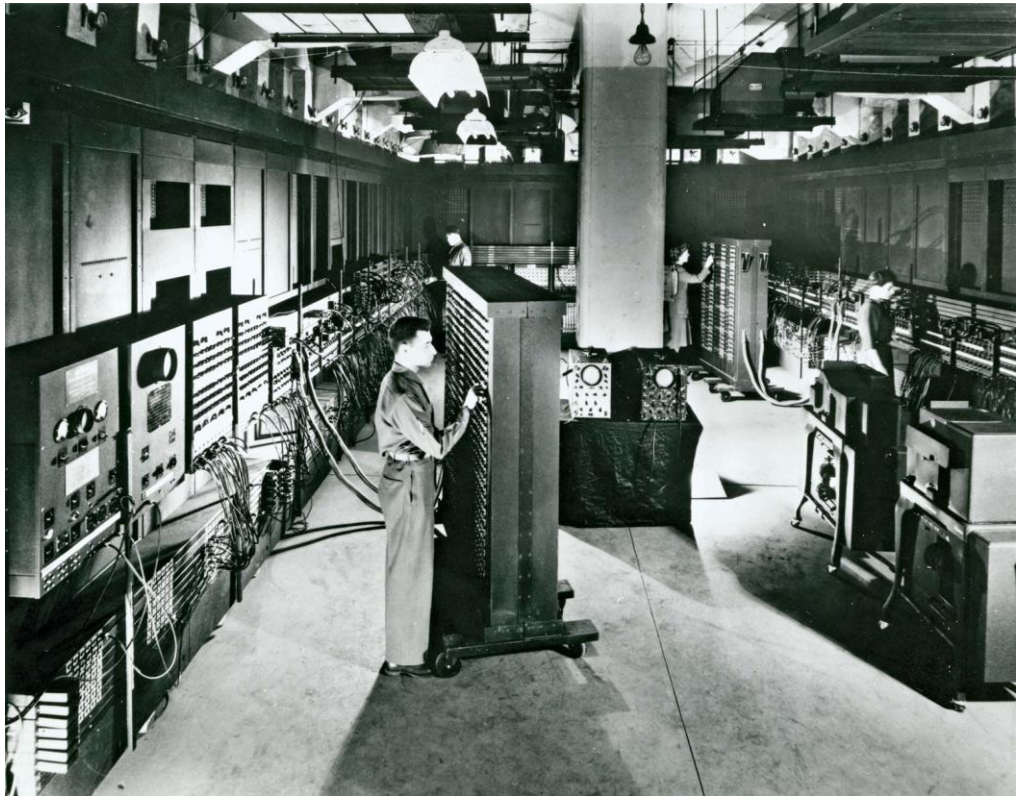
History

Main Generations of Computer Systems

1. First Generation (1940-1956): Vacuum Tubes
2. Second Generation (1956-1963): Transistors
3. Third Generation (1964-1971): Integrated Circuits (IC)
4. Fourth Generation (1971-Present): Microprocessors & Very Large IC
5. Fifth Generation (Present and Beyond): Artificial Intelligence

First Generation (1940-1956): Vacuum Tubes

- **Technology:** Vacuum tubes were used for circuitry and magnetic drums for memory.
- **Characteristics:**
  - Large size and high power consumption.
  - Limited programming capabilities, mainly machine language.
  - Example: ENIAC (Electronic Numerical Integrator and Computer).



Source: Britannica

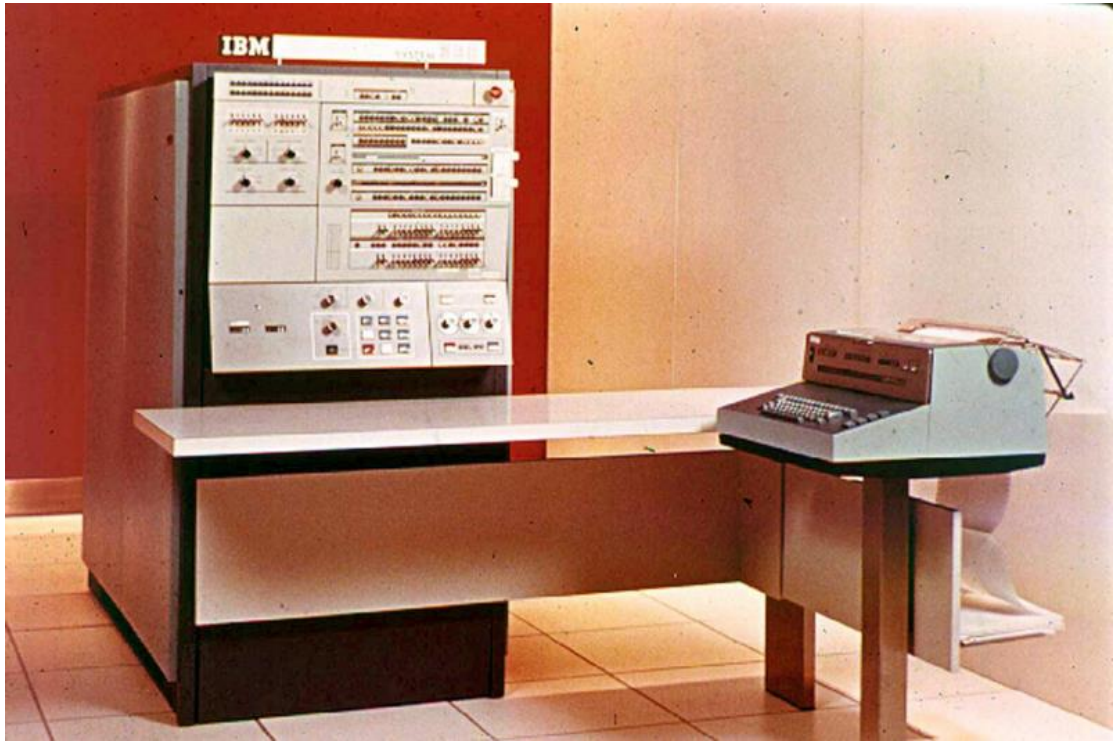
Second Generation (1956-1963): Transistors

- **Technology:** Transistors replaced vacuum tubes.
- **Characteristics:**
  - Smaller, faster, and more reliable than first-generation computers.
  - Lower power consumption.
  - Use of assembly language and high-level programming languages like COBOL and FORTRAN.
  - Example: IBM 7094.

Source: Columbia University

Third Generation (1964-1971): Integrated Circuits

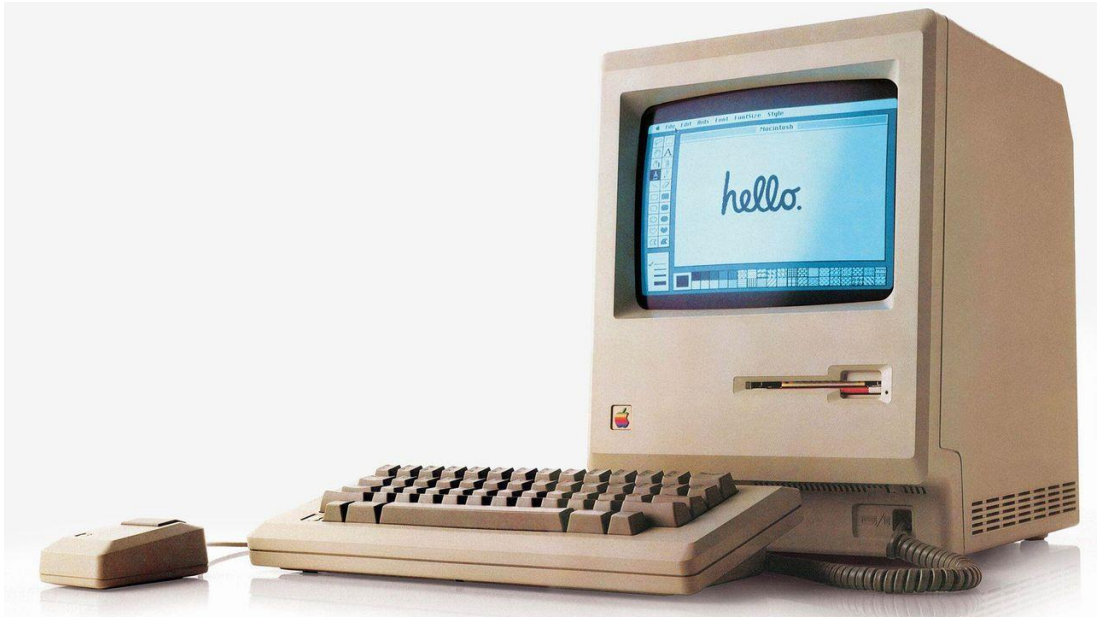
- **Technology:** Integrated Circuits (ICs) replaced transistors, with many transistors on a single chip.
- **Characteristics:**
  - Further size reduction, increased speed, and reliability.
  - Introduction of operating systems and multiprogramming.
  - More widespread use in businesses and scientific research.
  - Example: IBM System/360.



Source: https://sourcegraph.com/

Fourth Generation (1971-Present): Microprocessors

- **Technology:** Microprocessors, with thousands of integrated circuits on a single chip.
- **Characteristics:**
  - Personal computers became possible.
  - Development of GUIs, networking, and widespread use of the internet.
  - Dramatic improvements in processing power, storage, and cost-effectiveness.
  - Example: Intel 4004, Apple Macintosh, IBM PC.



Source: BBC

Fifth Generation (Present and Beyond): Artificial Intelligence

- **Technology:** Based on AI, quantum computing, and nanotechnology.
- **Characteristics:**
  - Development of intelligent systems capable of learning and decision-making.
  - Parallel processing, advanced AI algorithms, and massive storage capabilities.
  - Ongoing research in quantum computing, which could revolutionize computing with unprecedented speeds and power efficiency.
  - Example: IBM Watson, Google's AI systems, quantum computers.

Source: https://quantumai.google/discover/whatisqc

The Birth of Unix (1969-1970s)

- 1969: Unix developed at AT&T's Bell Labs by Ken Thompson, Dennis Ritchie, and others
- Originally created on a PDP-7 minicomputer
- Designed for simplicity, portability, and multitasking
- Introduction of the C programming language, allowing Unix to be rewritten and easily ported

Source: Wikipedia

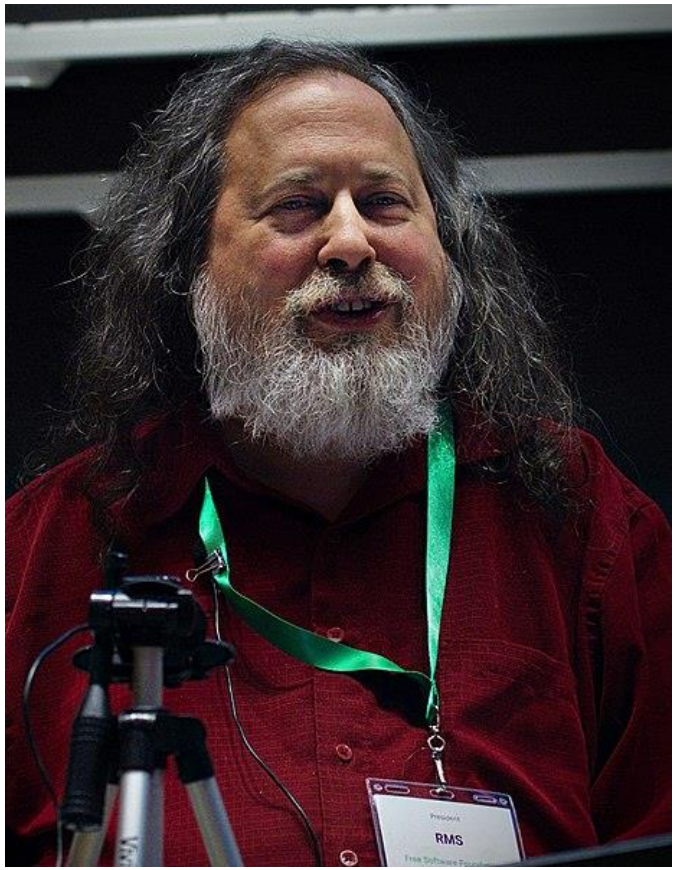
Unix Evolution and Expansion (1970s-1980s)

- 1973: Unix rewritten in C, making it easier to modify and port to different systems
- 1975: Unix Version 6 released, became widely adopted in academic and research institutions
- 1983: Birth of the "Unix Wars" due to multiple proprietary versions (e.g., AT&T's System V, BSD)
- The rise of different Unix variants like BSD, AIX, HP-UX, and Solaris

Source: Wikipedia

The GNU Project and Free Software Foundation

- 1983: Richard Stallman launches the GNU Project to create a free Unix-like operating system
- 1985: Formation of the Free Software Foundation (FSF) to support the GNU Project
- Importance of the GPL in promoting free and open-source software (FOSS)



Source: Wikipedia

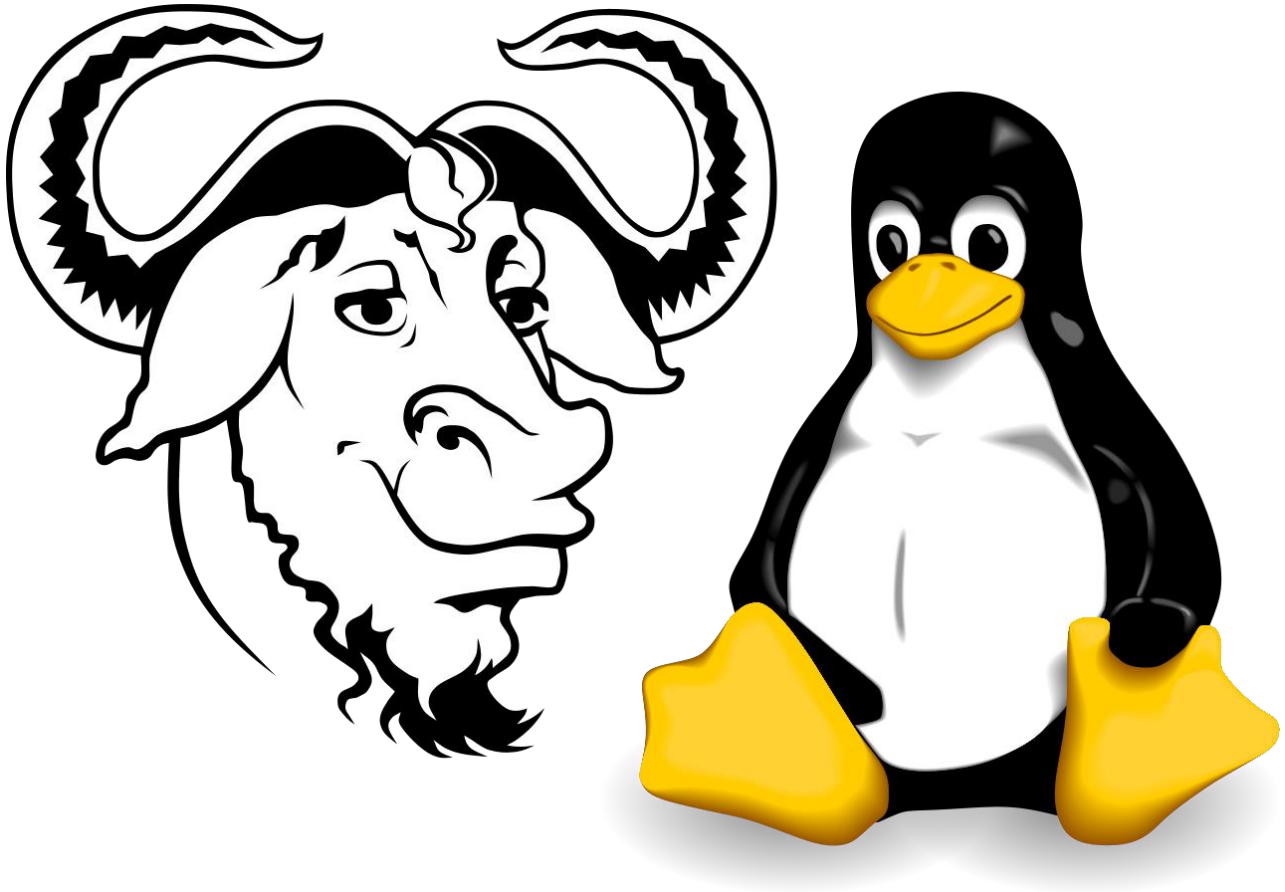
The Birth of Linux (1991)

- 1991: Linus Torvalds, a student from Finland, begins developing a Unix-like kernel
- Announced on the Usenet group comp.os.minix; initially a hobby project
- Released under the GNU General Public License (GPL), allowing free use and modification

Source: Wikipedia

GNU + Linux

Linux kernel combined with GNU tools creates a fully functional operating system



Source: Wikipedia

GNU + Linux

Foundation Key Differences Between Unix and Linux

- Unix: Proprietary, with various versions (System V, BSD)
- Linux: Open-source, with many distributions (Debian, Ubuntu, Red Hat)
- Unix primarily used in specialized, enterprise systems
- Linux has a broader range of applications from servers to desktops

Unix and Linux in the Modern Era

- Linux as the dominant force in servers, supercomputers, and cloud computing
- Role in Android OS, IoT devices, and embedded systems
- Unix's continued use in enterprise environments (e.g., AIX, Solaris)
- Influence on other modern operating systems like macOS and BSD derivatives

Overview of Microsoft Operating Systems

- Linux as the dominant force in servers, supercomputers, and cloud computing
- Role in Android OS, IoT devices, and embedded systems
- Unix's continued use in enterprise environments (e.g., AIX, Solaris)
- Influence on other modern operating systems like macOS and BSD derivatives

Unix and Linux in the Modern Era

- Linux as the dominant force in servers, supercomputers, and cloud computing
- Role in Android OS, IoT devices, and embedded systems
- Unix's continued use in enterprise environments (e.g., AIX, Solaris)
- Influence on other modern operating systems like macOS and BSD derivatives

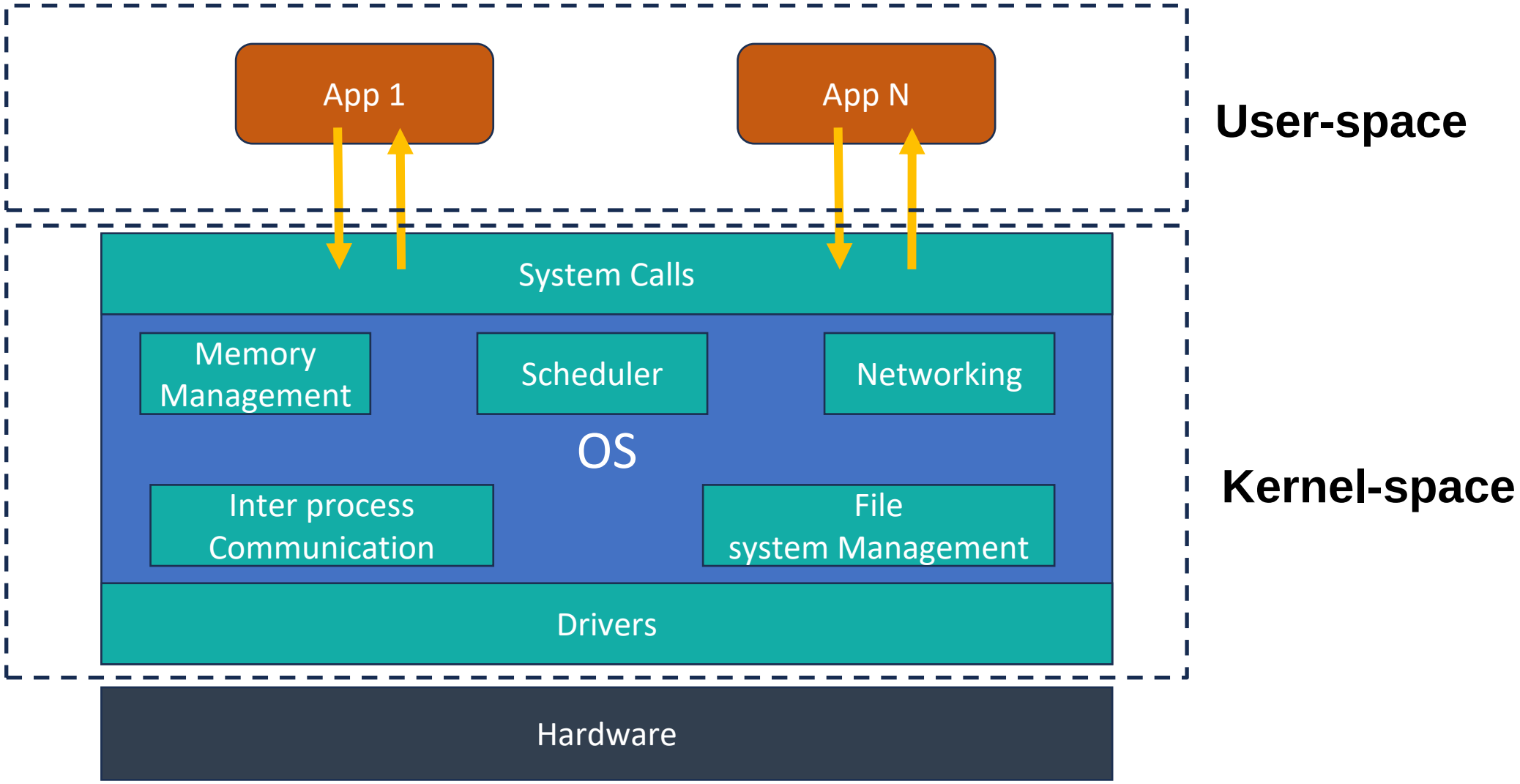
OS Types

- **Single-Tasking OS:** MS-DOS
- **Multi-Tasking OS:** GNU/Linux, Windows, macOS
- **Real-Time OS (RTOS):** VxWorks, QNX
- **Distributed OS:** Amoeba, Google's Fuchsia
- **Embedded OS:** FreeRTOS, embedded Linux distributions
- **Mobile OS:** Android, iOS

OS Responsibilities

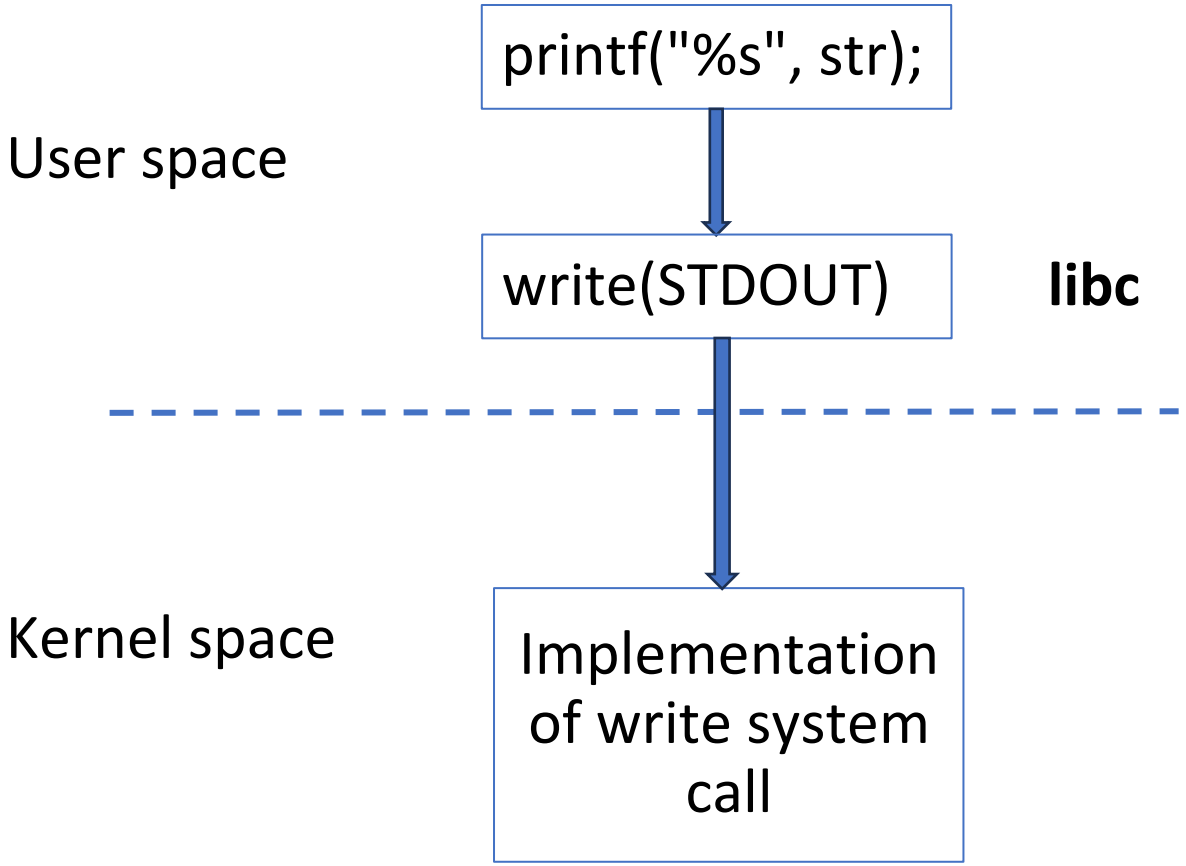
- **Process and Task Management**
- **Memory Management**
- **File System Management**
- **Device Management**
- **User Interface**
- **Security and Access Control**
- **Networking**
- **System Monitoring**

OS structure



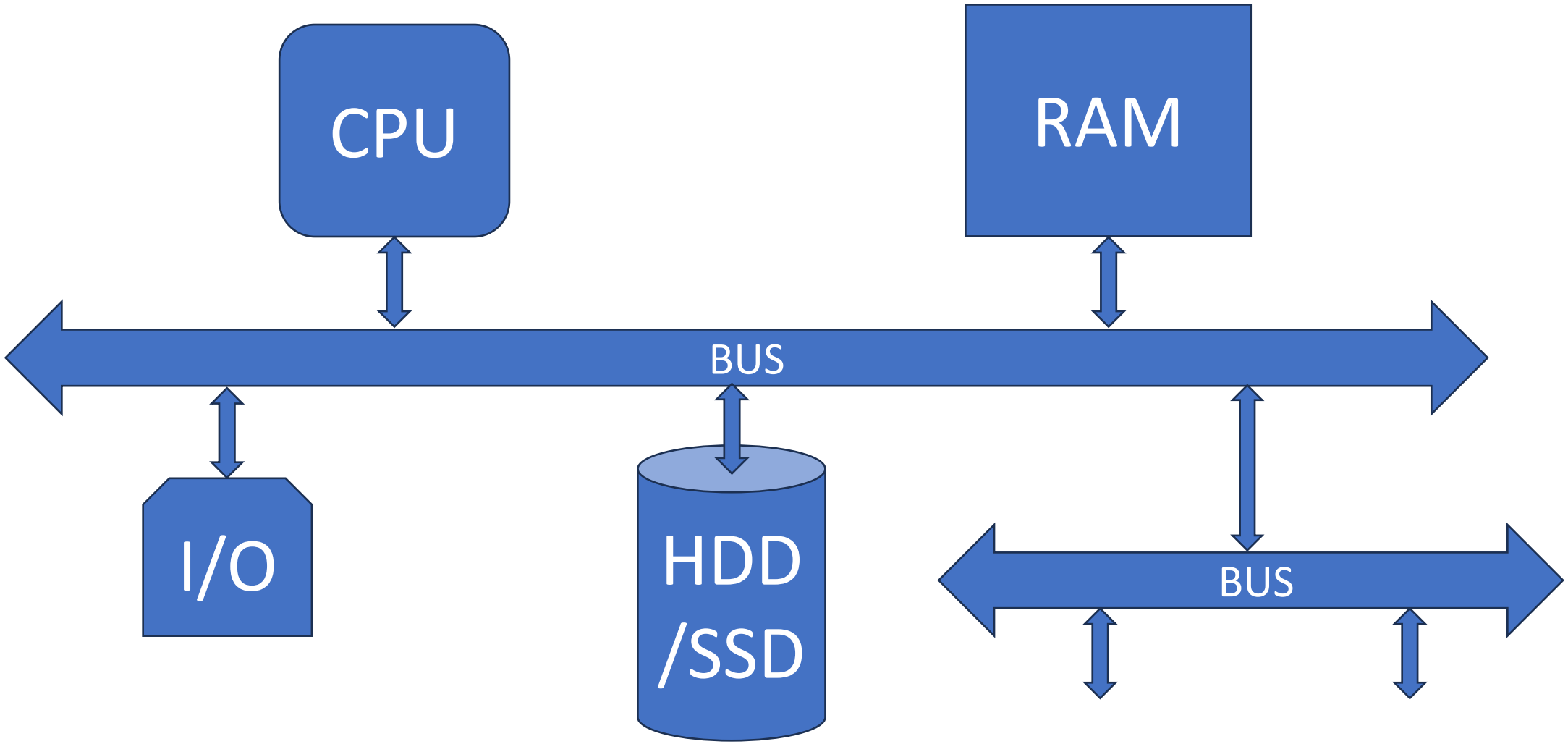
System calls

```
#include <stdio.h>
int main() {
    int x = 7;
    char str[] = "Hello\n";
    printf("%d %s",x, str);
}
```

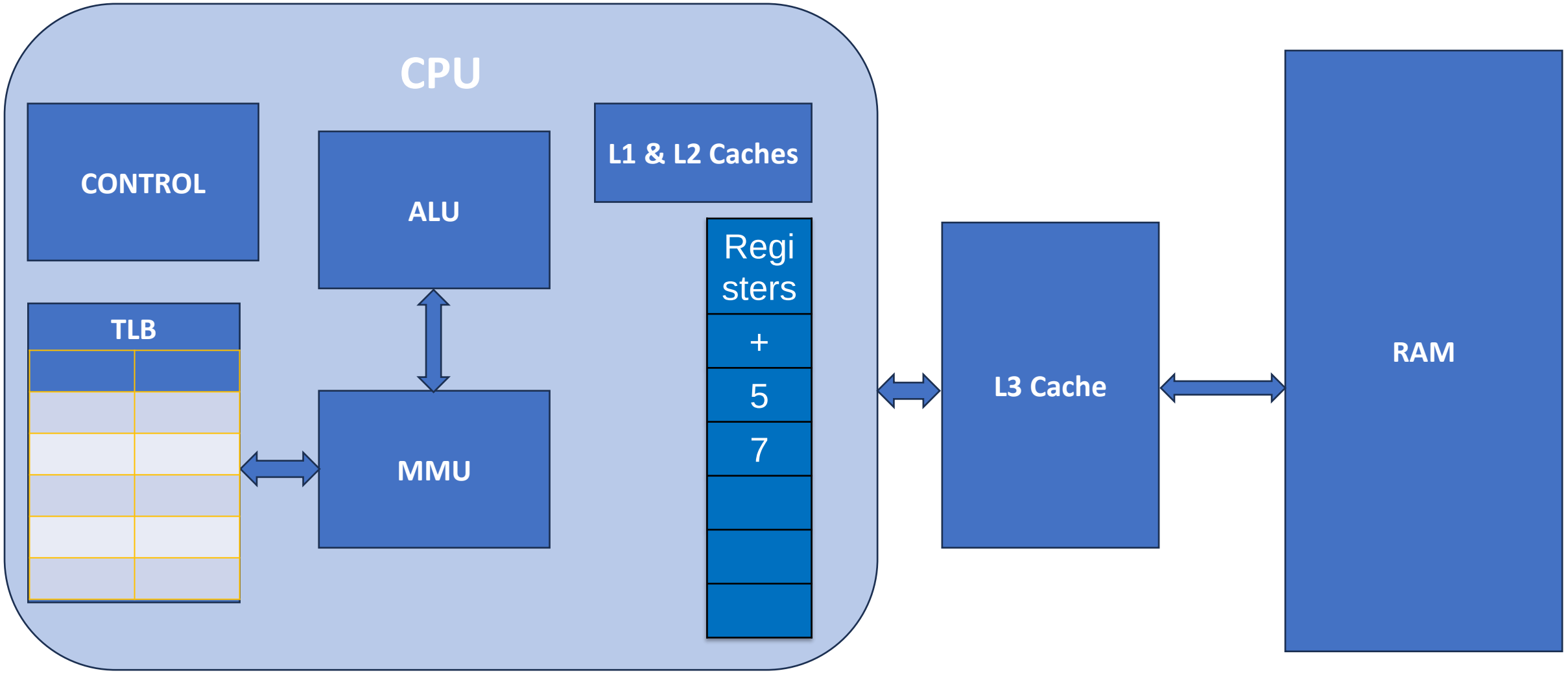


Computer Organization 101

Computer organization



Internal structure: CPU

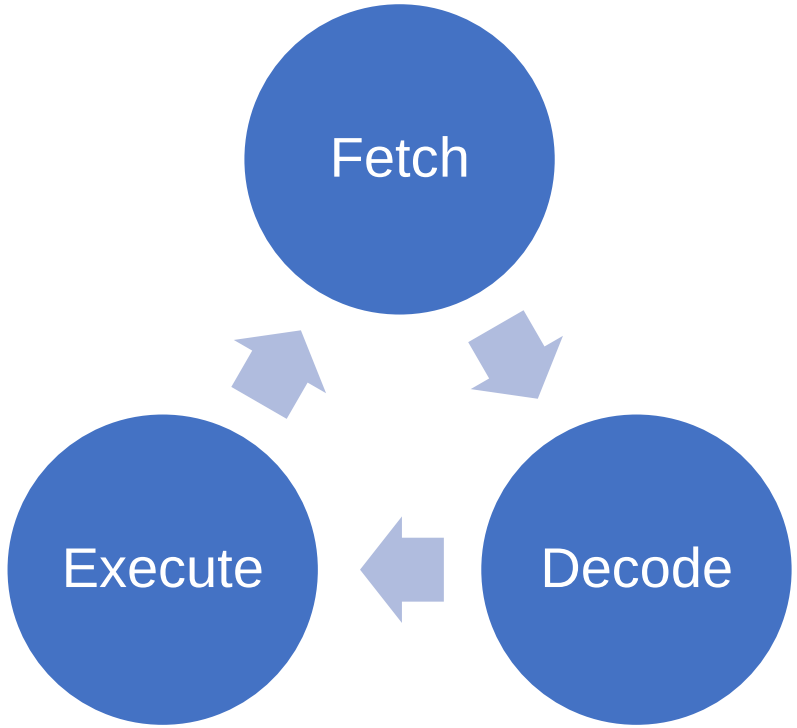


The CPU Fetch-Decode-Execute Cycle

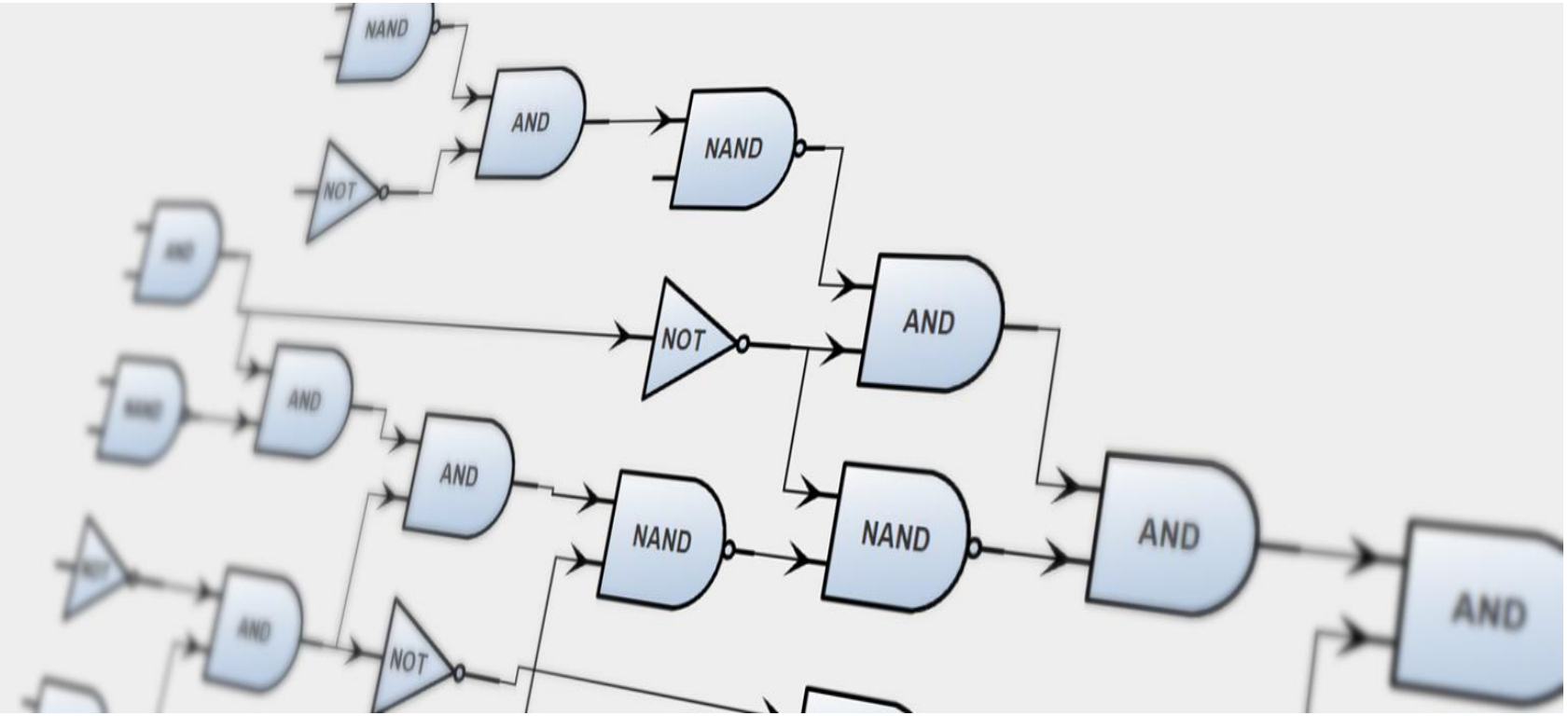
The CPU operates in a repeating cycle known as the **Fetch-Decode-Execute** cycle.

Key Phases of the Cycle:

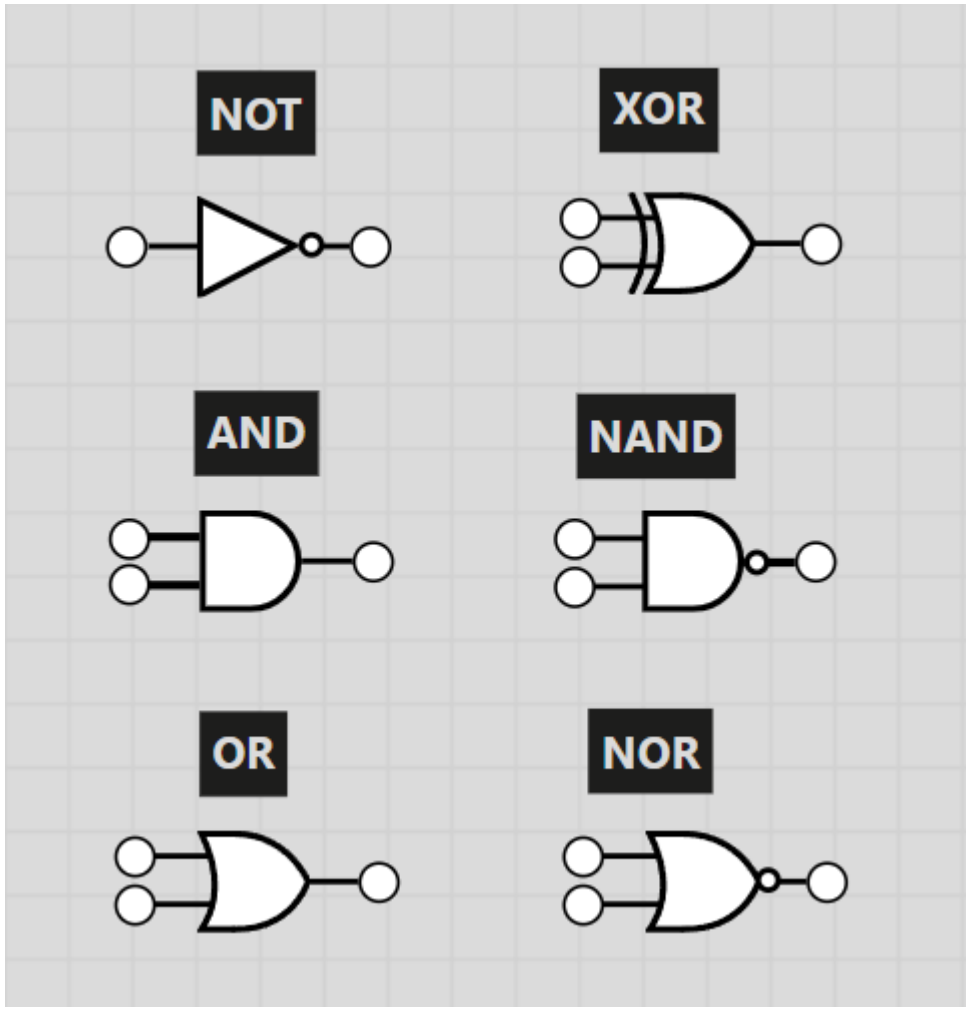
- **Fetch:** Retrieving the next instruction from memory.
- **Decode:** Interpreting the fetched instruction.
- **Execute:** Performing the operation dictated by the instruction.



Logical gates



Logical gates

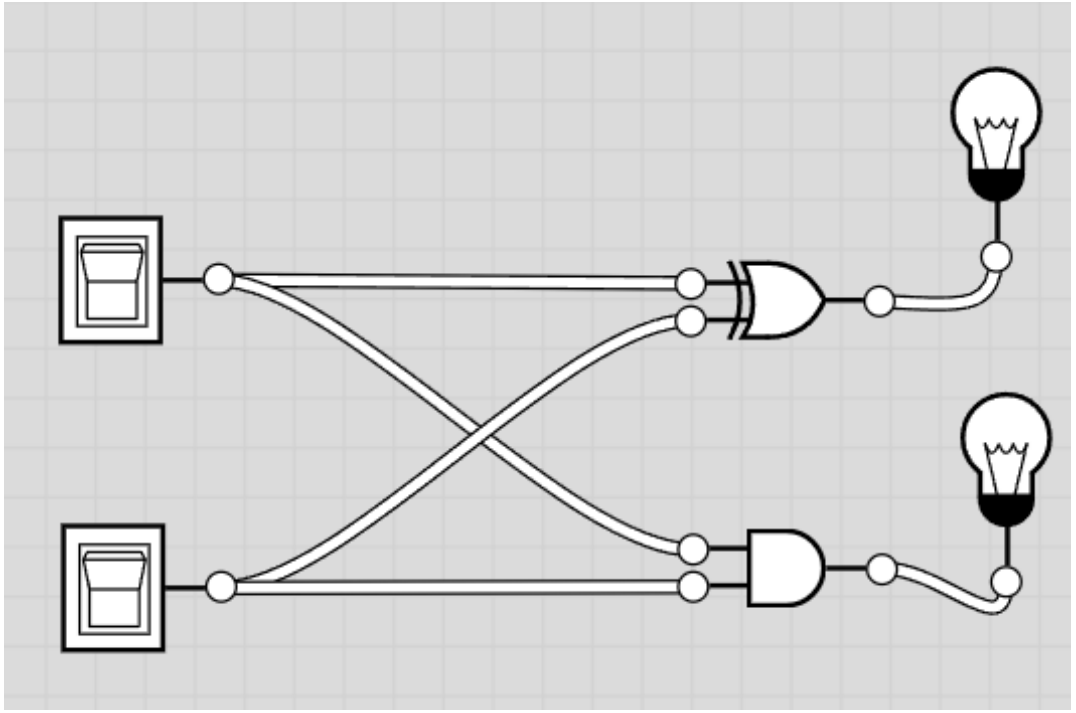


A	B	AND	NAND	OR	NOR	XOR
0	0	0	1	0	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	0	1	0	0

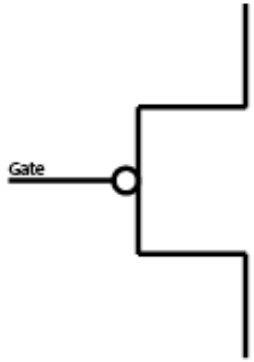
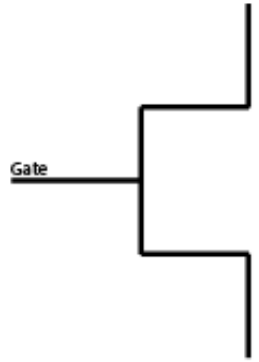
Half Adder

Z = X + Y

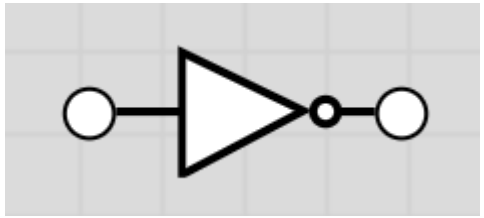
X	1	0	0	1	0	1	0	1
Y	0	1	0	0	1	1	1	0
Z								



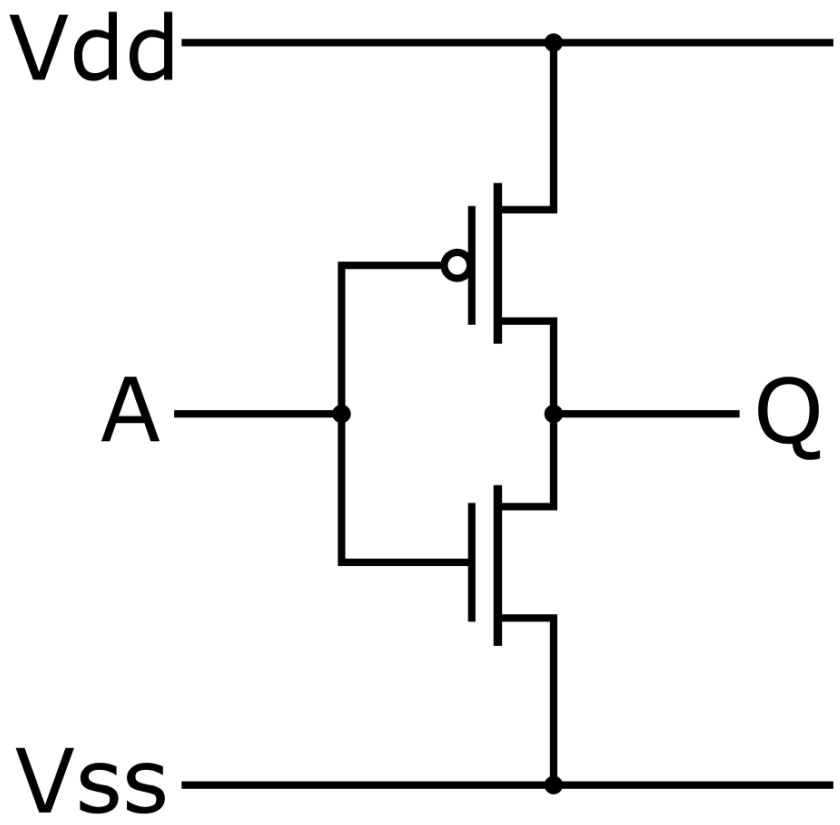
Transistor as a key

Signal on the Gate	PFET	NFET
		
0 (Logic Low, VDD)		
1 (Logic High, GND)		

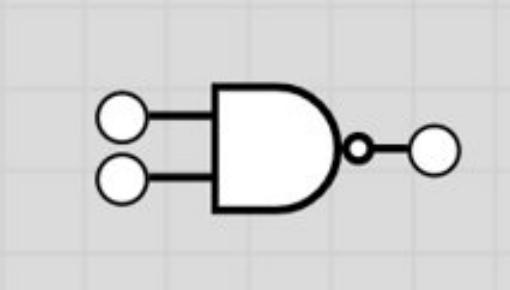
Gates internal: NOT



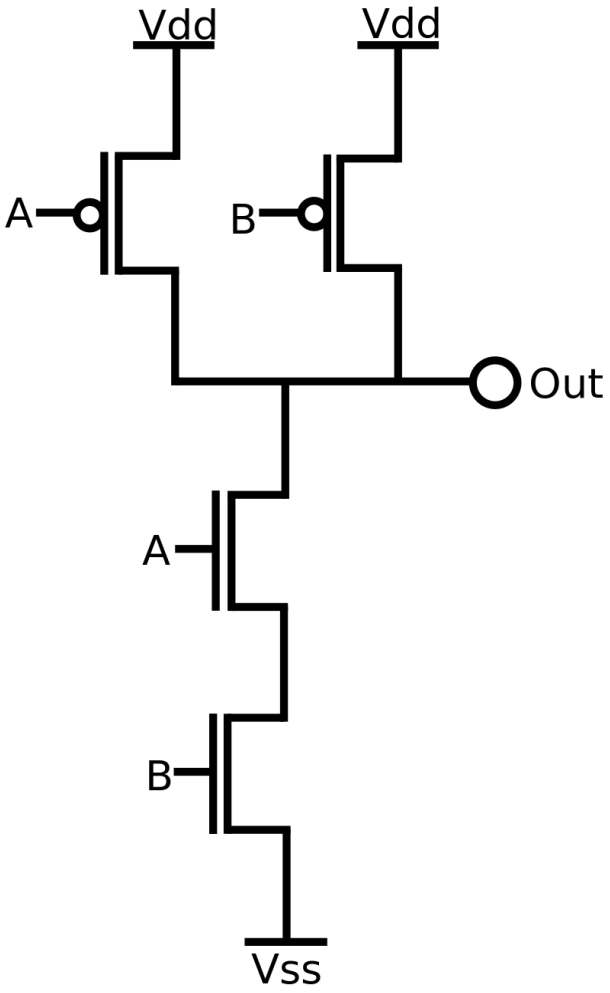
INPUT (A)	OUTPUT (Q)
0	1
1	0



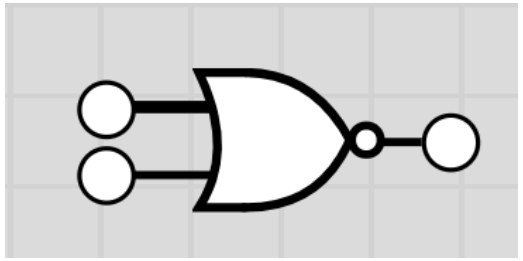
Gates internal: NAND



A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0



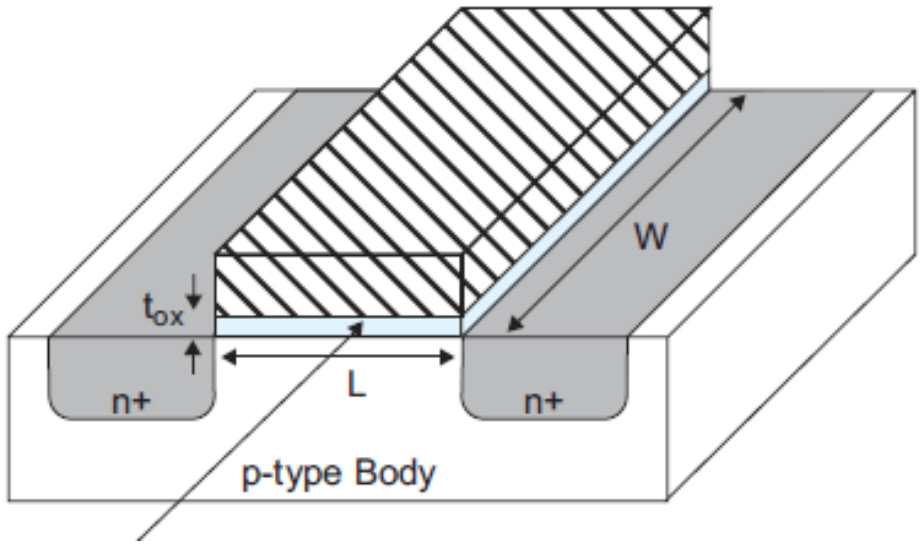
Gates internal: NOR

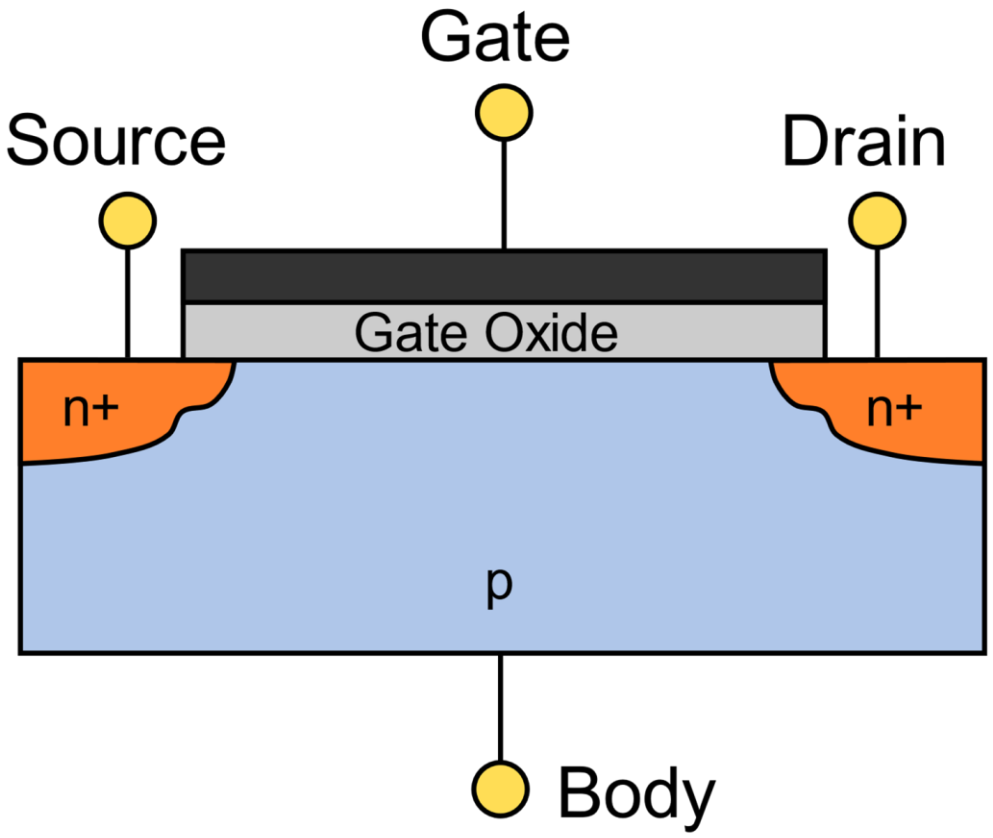


A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Schematics? ->**HOMEWORK**

Transistor internals





Interaction with devices

- **Polling**

CPU continuously checks the status of another device to determine if it needs attention.

- **Interrupts**

Interrupts are signals sent by devices to the CPU to indicate that they need attention.

Polling

Advantages

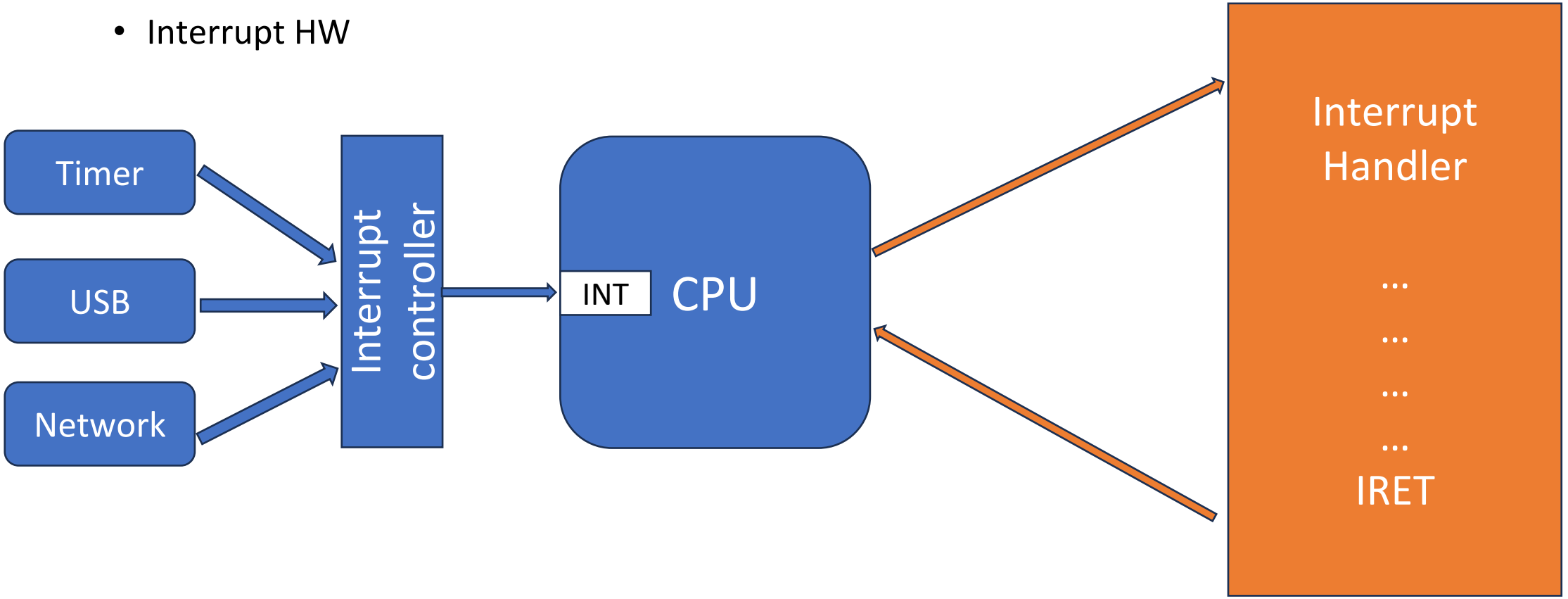
- Simplicity
- Compatibility
- Predictability

Disadvantages

- Resource Intensive
- Latency
- Complexity in Real-Time Systems

Interrupts

- Interrupt HW



Interrupts

Advantages

- Efficiency
- Reduced Latency
- Real-time Responsiveness
- Prioritization

Disadvantages

- Complexity
- Task Interference
- Race Conditions

Polling or Interrupts. Which to use?

Use Case	Polling	Interrupts
Frequency of Events	Useful for infrequent or irregular events	Efficient for frequent or regular events
System Responsiveness	Can introduce delays in response	Provides immediate response to events
CPU Utilization	May waste CPU cycles if events are rare	Efficient use of CPU cycles
Event Handling	Requires constant checking	Responds immediately to event occurrences

Simple C programs

```
#include <stdio.h>

int main()
{
        printf("Hello World\n");

        return 0;
}
```

```
#include <stdio.h>

int main()
{
        int a = 5, b = 7, c;
        c = a + b;
        printf("c = %d\n", c);

        return 0;
}
```

Simple C programs

```
#include <stdio.h>

int main()
{
        char str[] = “Hello World”;
        printf(“%s\n”, str);

        return 0;
}
```

```
#include <stdio.h>
struct Point
{
        double x;
        double y;
} ;
int main()
{
        Point p1;
        p1.x = 7.6;
        p1.y = 2.3;
        return 0;
}
```

Simple C programs

```
#include <stdio.h>

int main()
{
        int x = 5;
        int *p = &x;

        printf("%p\n", &x);
        printf("%p\n", p);

        printf("%d\n", x);
        printf("%d\n", *p);


        return 0;
}
```

```
#include <stdio.h>
struct Point
{
        double x;
        double y;
} ;
int main()
{
        Point p1;
        p1.x = 7.6;
        p1.y = 2.3;
        return 0;
}
```

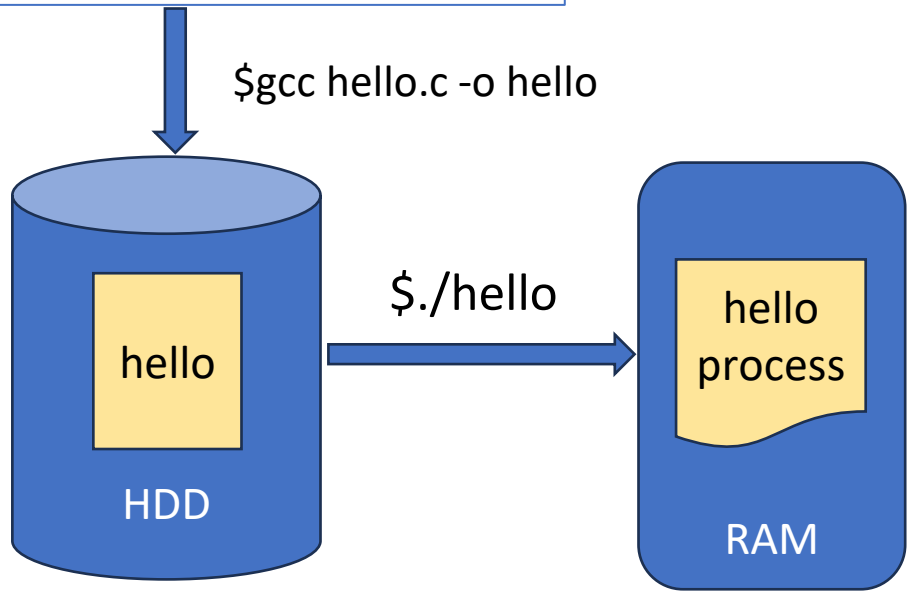
Processes

Process

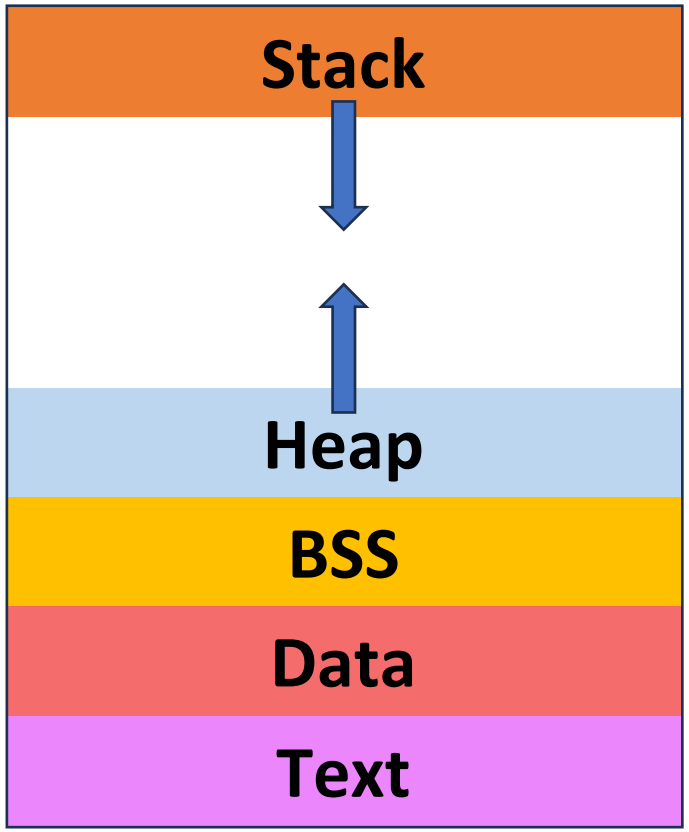
- A *process* is a program (object code stored on some media) in the middle of execution.
- Each thread includes a unique program counter, process stack, and set of processor registers.
- The kernel schedules individual threads, not processes. In traditional Unix systems, each process consists of one thread.
- Linux has a unique implementation of threads: It does not differentiate between threads and processes

Process

```
#include <stdio.h>
int main() {
    char str[] = "Hello\n";
    printf("%s", str);
}
```

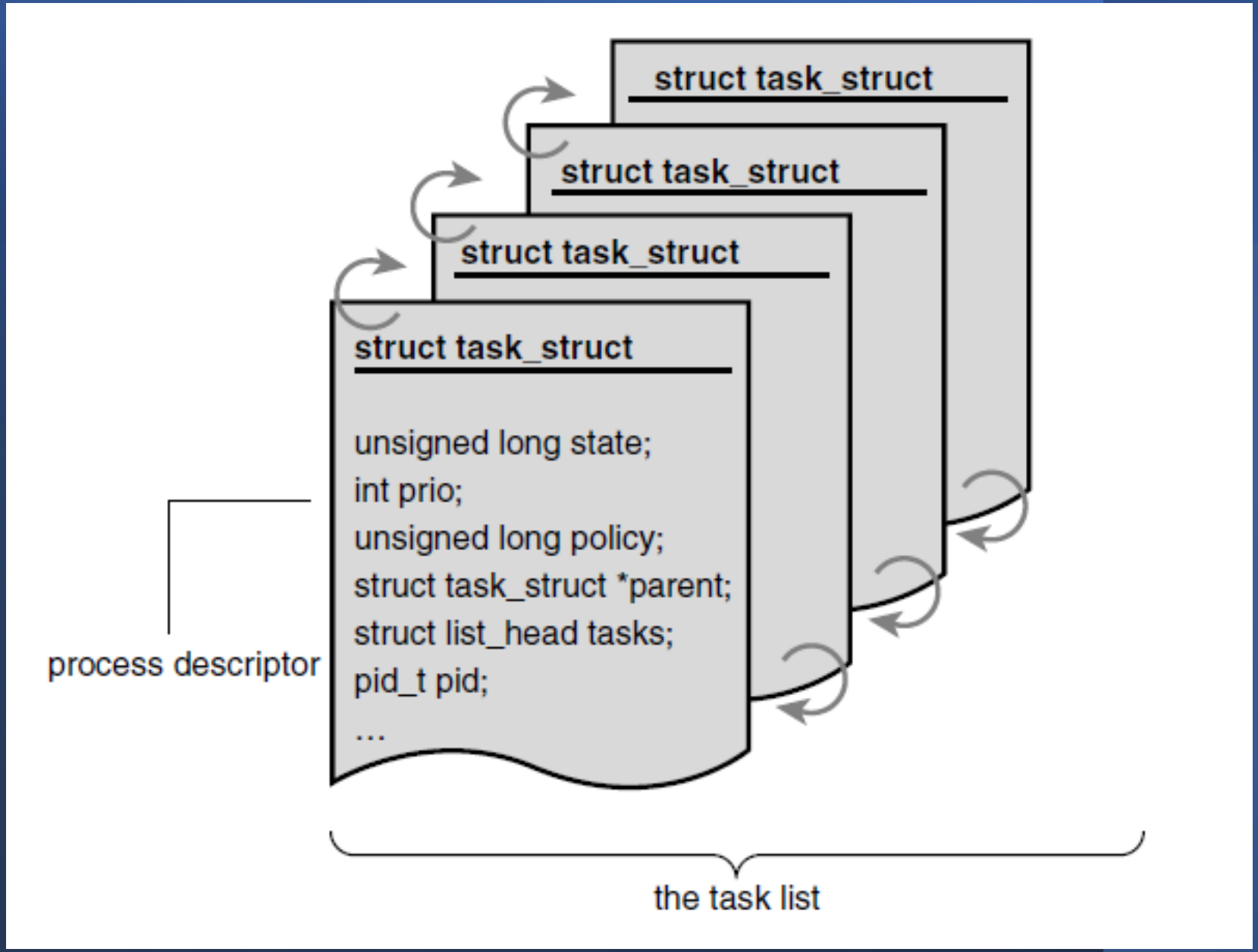


Memory layout of the process

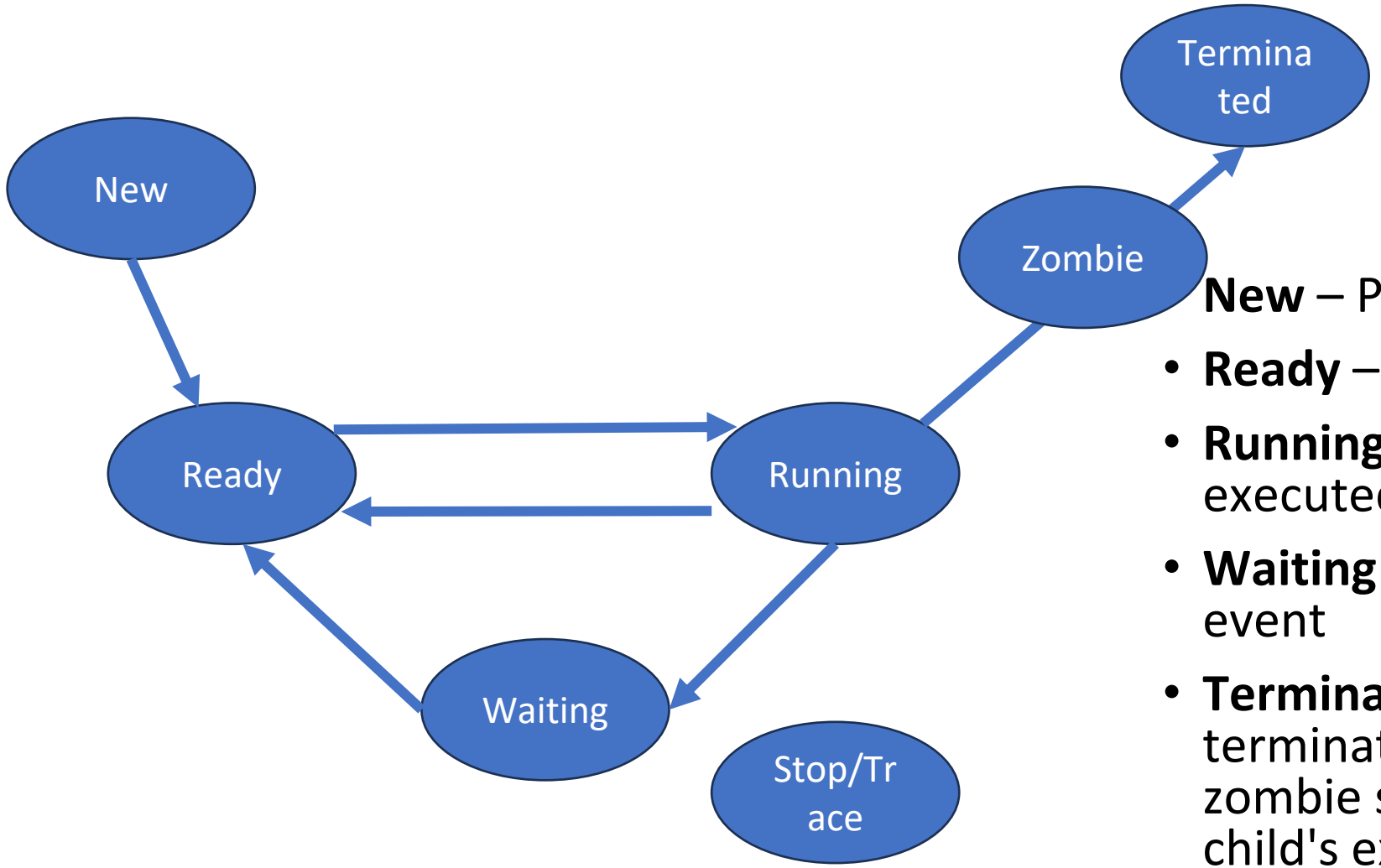


TASK_STRUCT

- The kernel stores the list of processes in a circular doubly linked list called the *task list*
- Each element in the task list is a *process descriptor* of the type struct **task_struct**, which is defined in <linux/sched.h>
- The task_struct is a relatively large data structure, at around 1.7 kilobytes on a 32-bit machine
- The process descriptor contains the data that describes the executing program—open files, the process's address space, pending signals, the process's state & etc.


struct task_struct
struct task_struct
struct task_struct
struct task_struct
unsigned long state;
int prio;
unsigned long policy;
struct task_struct *parent;
struct list_head tasks;
pid_t pid;
…
process descriptor
the task list

Process states



New – PID is assigned to the process

- **Ready** – The process is in ready queue
- **Running** – The process is currently executed
- **Waiting** – The process is waiting for an event
- **Terminated/Zombie** - The process is terminated. At first, it becomes a zombie so that the parent can read the child's exit status.

The Process ID

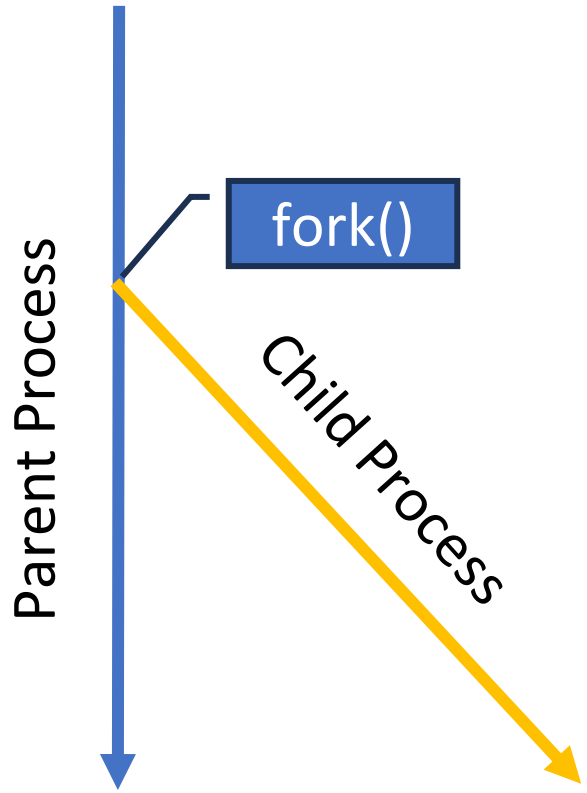
- Each process is represented by a unique identifier, the process ID (frequently shortened to pid)
- The pid is guaranteed to be unique at any *single point in time*.
- By default, the kernel imposes a maximum process ID value of 32768.
- The kernel allocates process IDs to processes in a strictly linear fashion. If pid 17 is the highest number currently allocated, pid 18 will be allocated next, even if the process last assigned pid 17 is no longer running when the new process starts.
- The kernel does not reuse process ID values until it wraps around from the top—that is, earlier values will not be reused until the value in */proc/sys/kernel/pid_max* is allocated

The Process Hierarchy

- The process that spawns a new process is known as the parent; the new process is known as the child.
- Every process is spawned from another process (except, of course, the init process).
- Therefore, every child has a parent.
- This relationship is recorded in each process's *parent process ID* (ppid), which is the pid of the child's parent.
- Each process is owned by a *user* and a *group*.

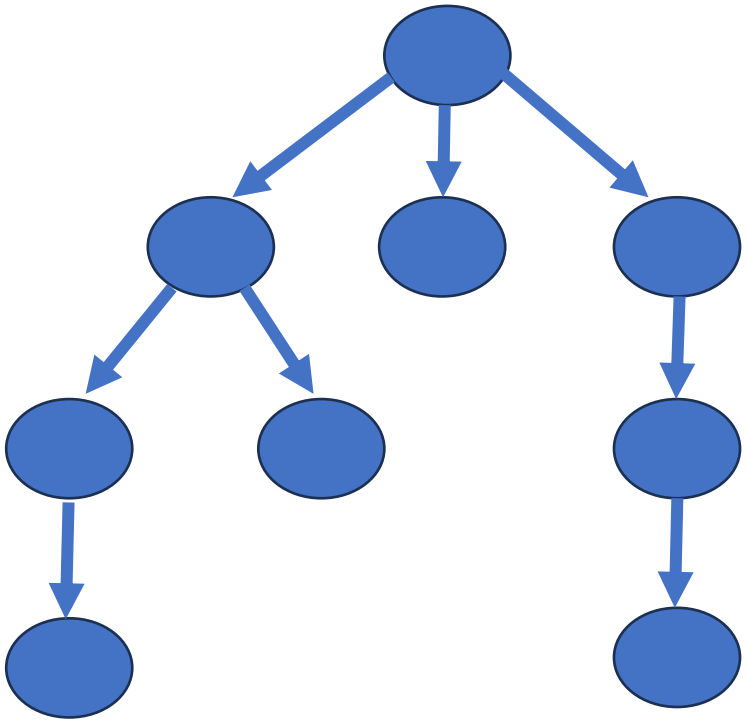
Process creation

- The child process is a replica of the parent
- In the beginning they share:
  - Memory pages
  - Kernel stack

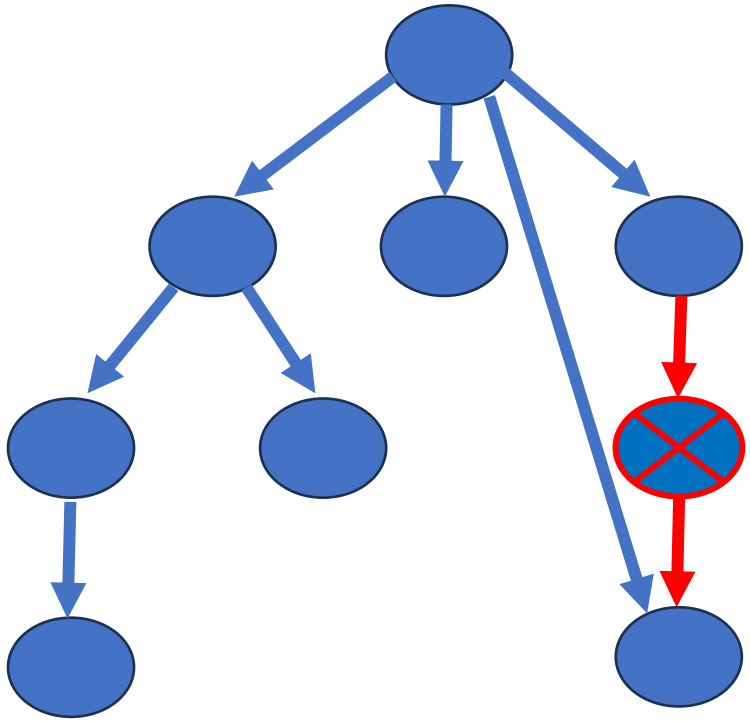


Process tree

Process tree



Orphans



The Process Hierarchy

- Programmatically, the process ID is represented by the pid_t type, which is defined in the header file <sys/types.h>.
- The exact backing C type is architecture-specific and not defined by any C standard. On Linux, however, pid_t is generally a typedef to the C int type.

Obtaining the Process ID and Parent Process ID

- The getpid() system call returns the process ID of the invoking process:
  - #include <sys/types.h>
  - #include <unistd.h>
  - pid_t getpid (**void**);
- The getppid() system call returns the process ID of the invoking process's parent:
  - #include <sys/types.h>
  - #include <unistd.h>
  - pid_t getppid (**void**);

Process Creation

- **fork()**

Creates a child process that is a copy of the current task.

It differs from the parent only in its PID (which is unique), its PPID (parent's PID, which is set to the original process), and certain resources and statistics, such as pending signals, which are not inherited.

- **exec()**

Loads a new executable into the address space and begins executing it.

Forking

- Linux implements fork() via the **clone()** system call.
- The clone() system call, in turn, calls do_fork().
- The bulk of the work in forking is handled by do_fork(), which is defined in kernel/fork.c

Forking experiment 1

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

int main()
{
        // make two process which run same program after this instruction
        fork();

        printf("Hello world!\n");
        return 0;
}
```

Forking

- In the child, a successful invocation of fork() returns 0.
- In the parent, fork() returns the pid of the child.
- Resource statistics are reset to zero in the child.
- Any pending signals are cleared and not inherited by the child

On error, a child process is not created, fork() returns −1, and errno is set appropriately.

- EAGAIN

The kernel failed to allocate certain resources, such as a new pid.

- ENOMEM

nsufficient kernel memory was available to complete the request.

Forking experiment 2

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        if (fork() == 0)                // child process because return value zero
                printf("Hello from Child!\n");
        else                            // parent process because return value non-zero.
                printf("Hello from Parent!\n");

        return 0;
}
```

exec

- A call to execl() replaces the current process image with a new one by loading into memory the program pointed at by path.
- The execl() function is *variadic*
- The list of arguments must be NULL-terminated

Example 1

```
int ret;
ret = execl ("/bin/vi", "vi", NULL);
if (ret == −1)
        perror ("execl");
```

Example 2

```
int ret;
ret = execl ("/bin/vi", "vi",
        "/home/kidd/hooks.txt",
        NULL);
if (ret == −1)
        perror ("execl");
```

exec

- Normally, execl() does not return.
- A successful invocation ends by jumping to the entry point of the new program, and the just-executed code no longer exists in the process's address space.
- On error, however, execl() returns −1 and sets errno to indicate the problem.

exec

A successful execl() call changes not only the address space and process image, but certain other attributes of the process

- Any pending signals are lost.
- Any signals that the process is catching are returned to their default behavior, as the signal handlers no longer exist in the process's address space.
- Any memory locks are dropped.
- Most thread attributes are returned to the default values.
- Most process statistics are reset.
- Anything related to the process's memory address space, including any mapped files, is cleared.
- Anything that exists solely in user space, including features of the C library, such as atexit() behavior, is cleared.

exec experimenting

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

int main()
{
   int ret;
   ret = execl("/usr/bin/vim", "vim", NULL);
   if (ret == -1)
   {
      perror("execl");
   }
   printf(“After exec\n”);
   return 0;
}
```

The rest of the family

- #include <unistd.h>
- **int** execlp (**const char** *file, **const char** *arg, ...);
- **int** execle (**const char** *path, **const char** *arg, ..., **char** * **const** envp[]);
- **int** execv (**const char** *path, **char** ***const** argv[]);
- **int** execvp (**const char** *file, **char** ***const** argv[]);
- **int** execve (**const char** *filename, **char** ***const** argv[], **char** ***const** envp[]);

- The l and v delineate whether the arguments are provided via a *list* or an array (*vector*).
- The p denotes that the user's full *path* is searched for the given file. Commands using the p variants can specify just a filename, so long as it is located in the user's path.

Terminating a Process

- POSIX and C89 both define a standard function for terminating the current process:

```
#include <stdlib.h>
void exit (int status);
```

- A call to exit() performs some basic shutdown steps, then instructs the kernel to terminate the process.
- This function has no way of returning an error—in fact, it never returns at all.
- Therefore, it does not make sense for any instructions to follow the exit() call.

- EXIT_SUCCESS and EXIT_FAILURE are defined as portable ways to represent success and failure.
- Consequently, a successful exit is as simple as this one-liner: `exit (EXIT_SUCCESS);`

Terminating a Process

Before terminating the process, the C library performs the following shutdown steps, in order:

1. Call any functions registered with atexit() or on_exit(), in the reverse order of their registration.
2. Flush all open standard I/O streams.
3. Remove any temporary files created with the tmpfile() function.

These steps finish all the work the process needs to do in user space, so exit() invokes the system call _exit() to let the kernel handle the rest of the termination process:

- `#include <unistd.h>`
- `void _exit (int status);`

Other Ways to Terminate

- The classic way to end a program is not via an explicit system call, but by simply “falling off the end” of the program.
- In the case of C or C++, this happens when the main() function returns.
- The “falling off the end” approach, however, still invokes a system call: the compiler simply inserts an implicit call to exit() after its own shutdown code.
- It is good coding practice to explicitly return an exit status, either via exit(), or by returning a value from main().
- The shell uses the exit value for evaluating the success or failure of commands. Note that a successful return is exit(0), or a return from main() of 0.
- A process can also terminate if it is sent a signal whose default action is to terminate the process. Such signals include SIGTERM and SIGKILL
- The kernel can kill a process for executing an illegal instruction, causing a segmentation violation, running out of memory, consuming more resources that allowed, and so on.

atexit()

The atexit() library call, used to register functions to be invoked upon process termination:

```
#include <stdlib.h>

int atexit (void (*function)(void));
```

- A successful invocation of atexit() registers the given function to run during normal process termination, that is, when a process is terminated via either exit() or a return from main().
- If a process invokes an exec function, the list of registered functions is cleared (as the functions no longer exist in the new process's address space).

The given function takes no parameters, and returns no value. A prototype has the form:

```
void my_function (void);
```

Functions are invoked in the reverse order that they are registered(LIFO).

atexit() experiemnting

```
#include <stdio.h>
#include <stdlib.h>
void out (void)
{
        printf ("atexit() succeeded!\n");
}
int main (void)
{
        if (atexit (out))
                fprintf(stderr, "atexit() failed!\n");
Ddhfgahguifh
dshailusdagfik
        return 0;
}
```

Waiting for Terminated Child Processes

- When a child dies before its parent, the kernel should put the child into a special process state.
- A process in this state is known as a *zombie*.
- A process in this state waits for its parent to inquire about its status (a procedure known as *waiting on* the zombie process). Only after the parent obtains the information preserved about the terminated child does the process formally exit and cease to exist even as a zombie.
- The Linux kernel provides several interfaces for obtaining information about terminated children.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
```

What is the value of num

```
int num = 5;
pid_t pid = fork();

if (pid == 0) { // Child process
    printf("Child num = %d\n", num);
} else { // Parent process
    wait(NULL);
    printf("Parent num = %d\n", num);
}
```

Output

```
Child num = 5 / 0 / err
Parent num = 5
```

```
int num = 5;
pid_t pid = fork();

if (pid == 0) { // Child process
    num++;
    printf("Child num = %d\n", num);
} else { // Parent process
    wait(NULL);
    printf("Parent num = %d\n", num);
}
```

Output

```
Child num = 6
Parent num = 6 / 5/
```

Building executable

Build Stages

1. Preprocessing
   - Handles macro expansions, file inclusions, and conditional compilation.
2. Compilation
   - Converts preprocessed code to assembly language.
3. Assembly
   - Converts assembly language to machine code, producing object files.
4. Linking
   - Combines multiple object files, resolves symbols, and includes libraries to produce a complete executable.

Build Stages

1. Preprocessing
   - gcc -E hello.c -o hello.i
2. Compilation
   - gcc -S hello.i -o hello.s
3. Assembly
   - as hello.s -o hello.o
4. Linking
   - gcc hello.o -o hello
   - ld hello.o -o hello -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2

Memory Management

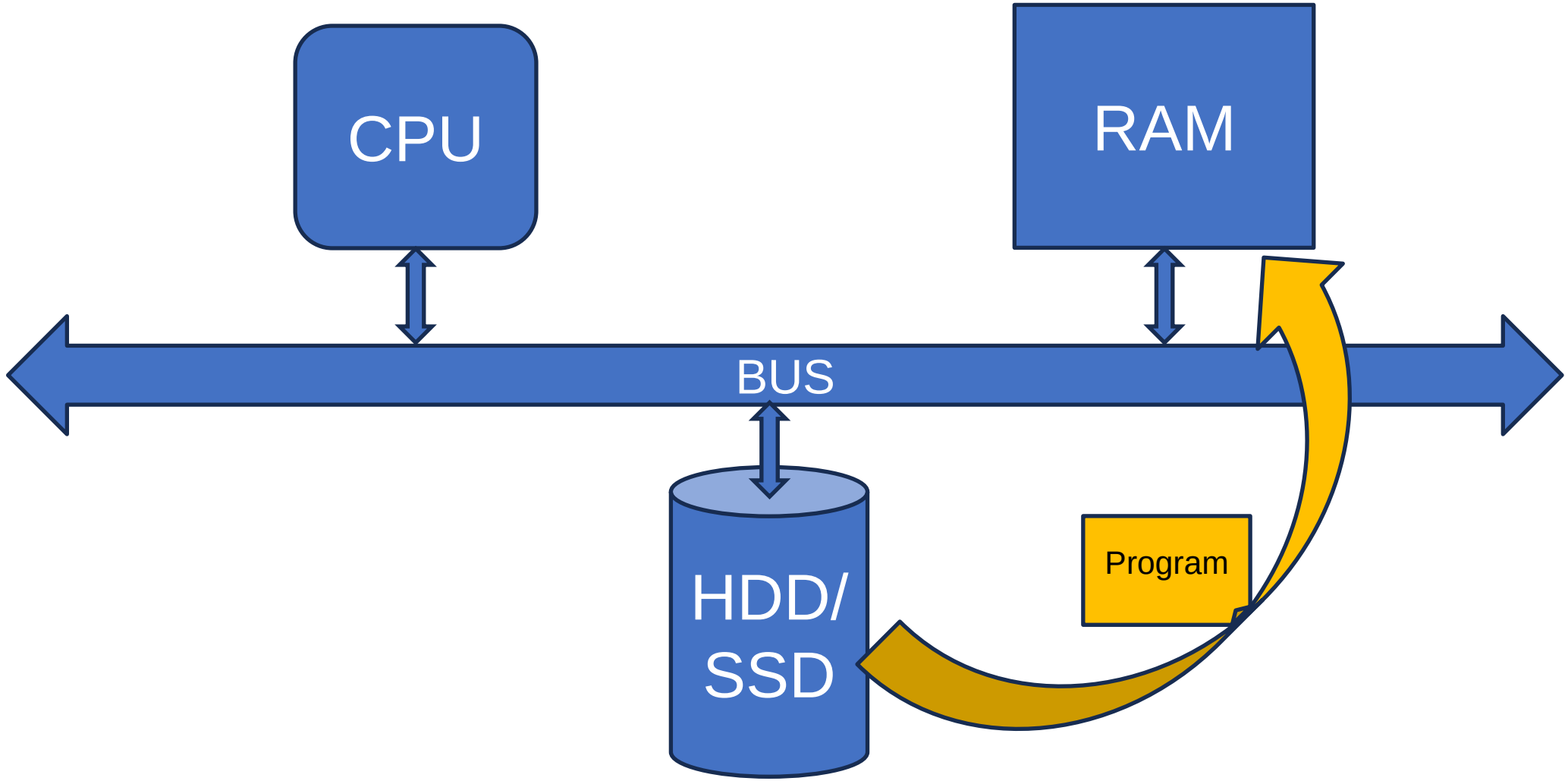
How your program really works: Memory management.

The Linux Way

A "simple" program

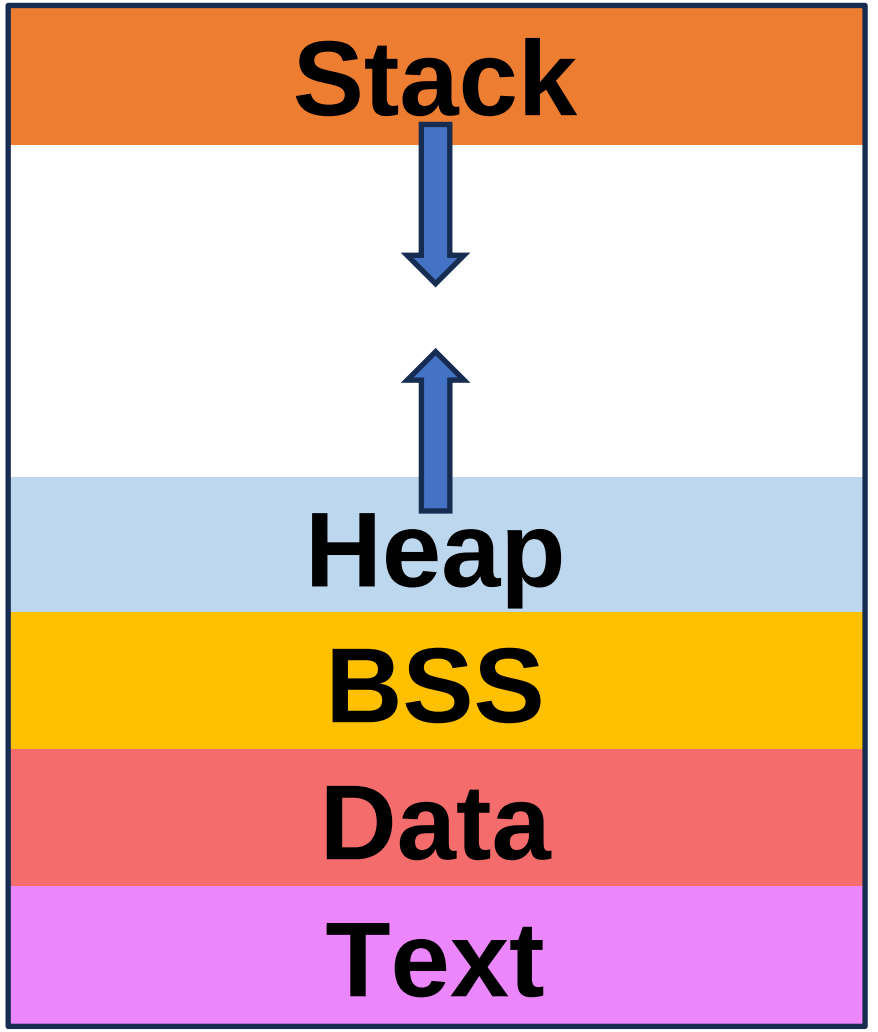
```
#include <stdio.h>

int globalBSSVariable;
int globalDataVariable = 42;

int main() {
    int stackVariable = 10;
    int* heapVariable = malloc(sizeof(int));
    free(heapVariable);
    return 0;
}
```

Hardware Architecture 101



The Process Address Space

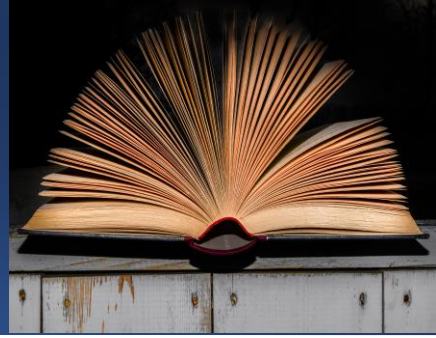
- **Stack**: Automatic variables
  int **stackVariable** = 7;
- **Heap**: Runtime dynamic allocations
  int *ptr = new malloc(10 * sizeof(int);
- **BSS**: Uninitialized Global and Static Variables
  int globalVariable;
- **Data**: Initialized global and static variables
  int **globalDataVariable** = 42;
- **Text**: Process's program code



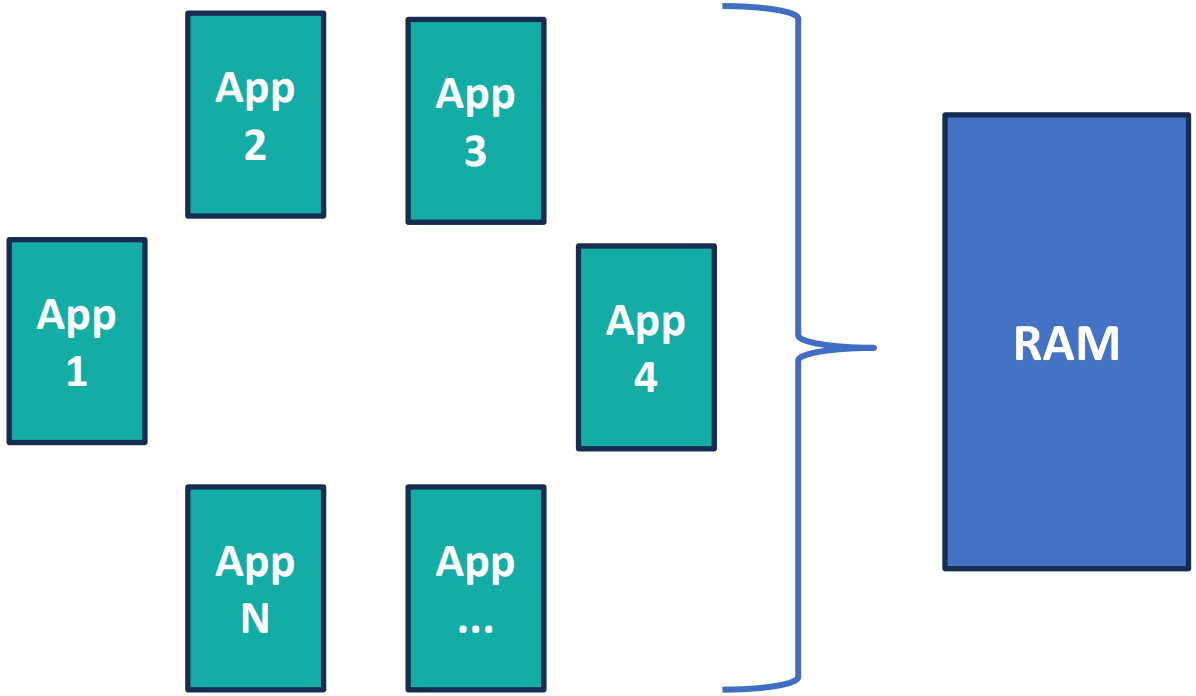
Memory Regions of the Process

- The ***text*** *segment* contains a process's program code, string literals, constant variables, and other read-only data.
- The ***stack*** contains the process's execution stack, which grows and shrinks dynamically as the stack depth increases and decreases. The execution stack contains local variables and function return data.
- The ***data*** *segment*, or *heap*, contains a process's dynamic memory.
- The ***bss*** *segment* contains uninitialized global variables. These variables contain special values (essentially, all zeros), per the C standard.

The idea of paging



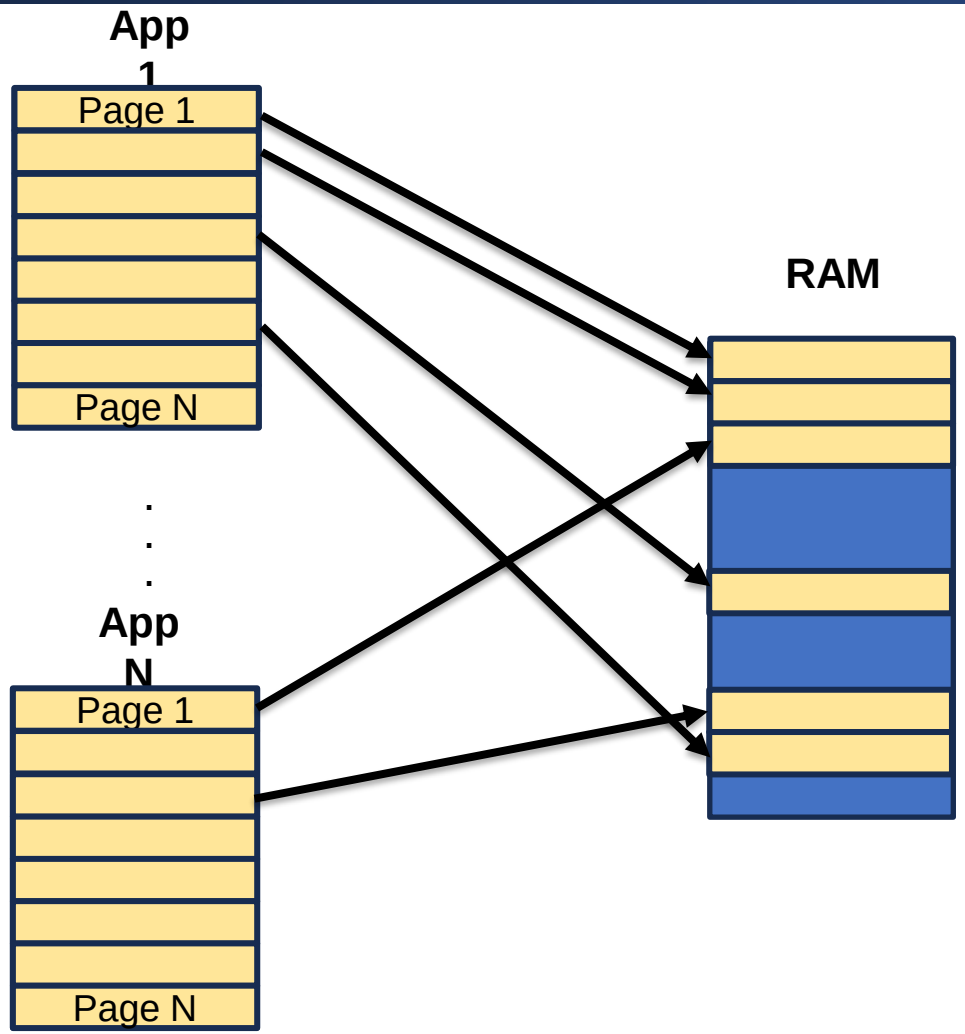
What if the total amount of required memory exceeds available physical memory?

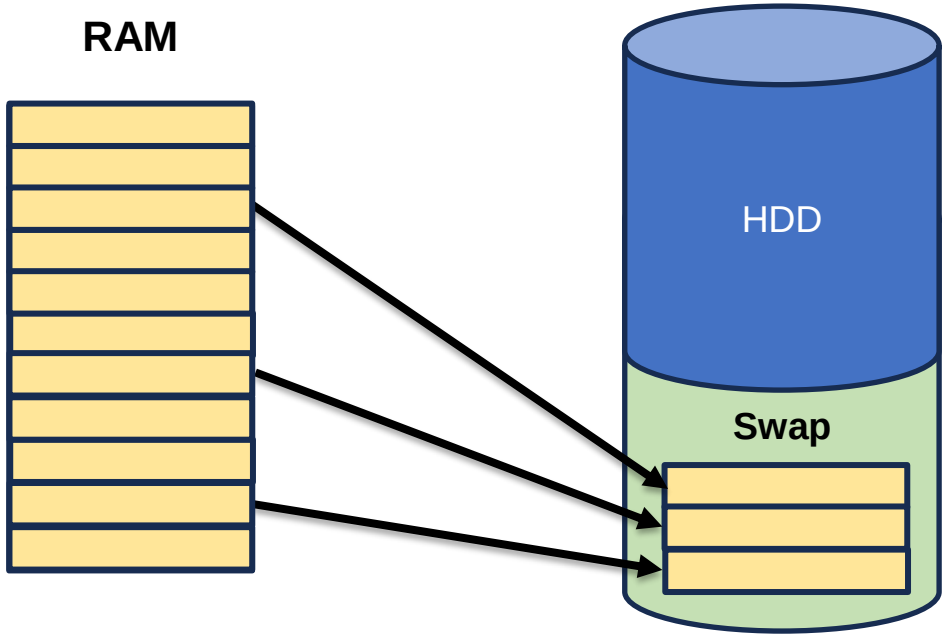


- **Division of memory into "pages"**
- **We can refer to memory by the Page Frame Number (RFN) + offset into the page**
- **Common size is 4KB**
- **Number_of_pages = Total_memory_size / page_size**

➢ **<u>Load and keep in memory only the required pages</u>**

The idea of paging: swap



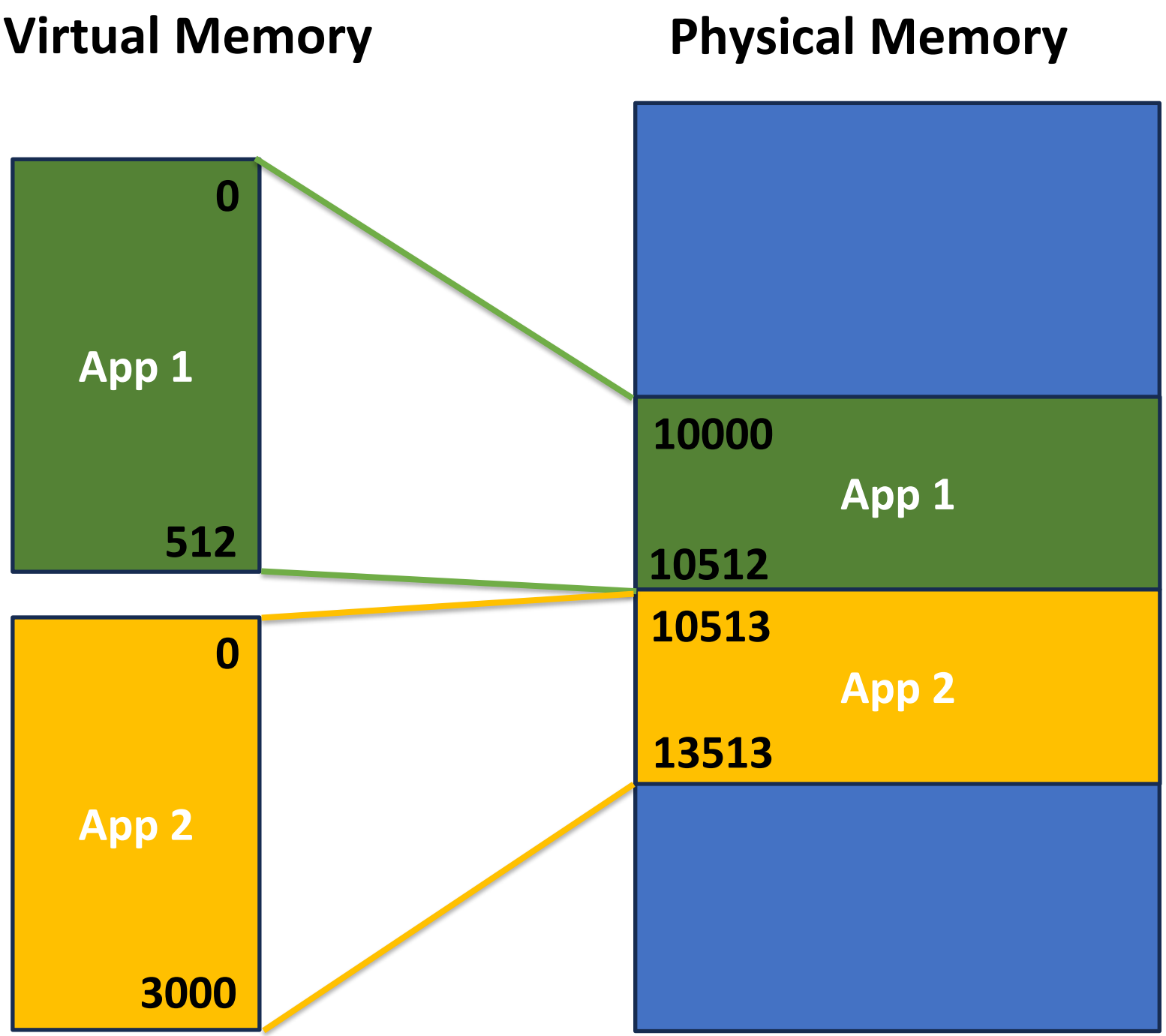


Variable's Address

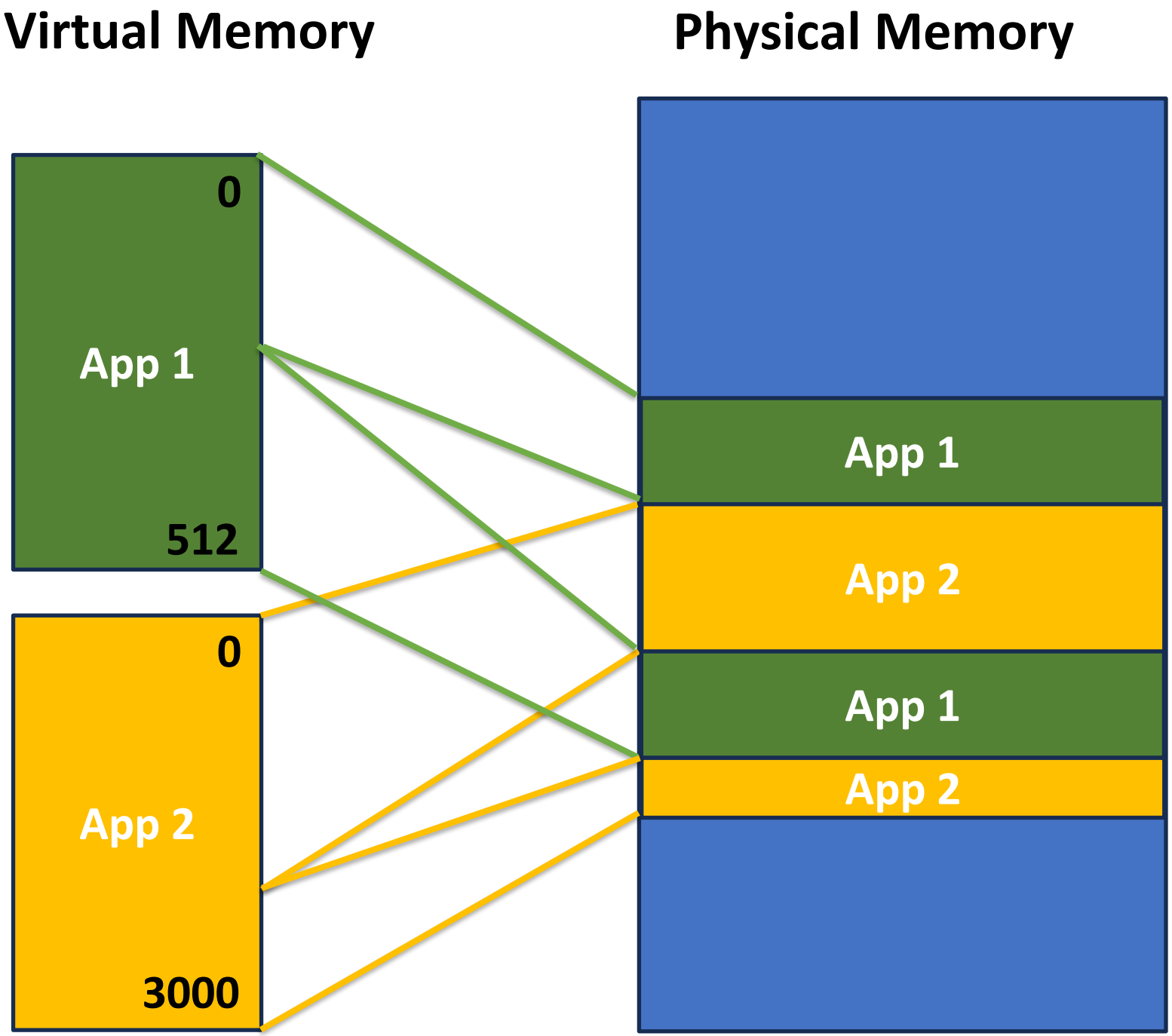
```
#include <stdio.h>

int main() {
    int X = 42;

    printf("Address of X: %p\n“, &X);

    return 0;
}
```

 # Address of X: 0x7fff58125a48

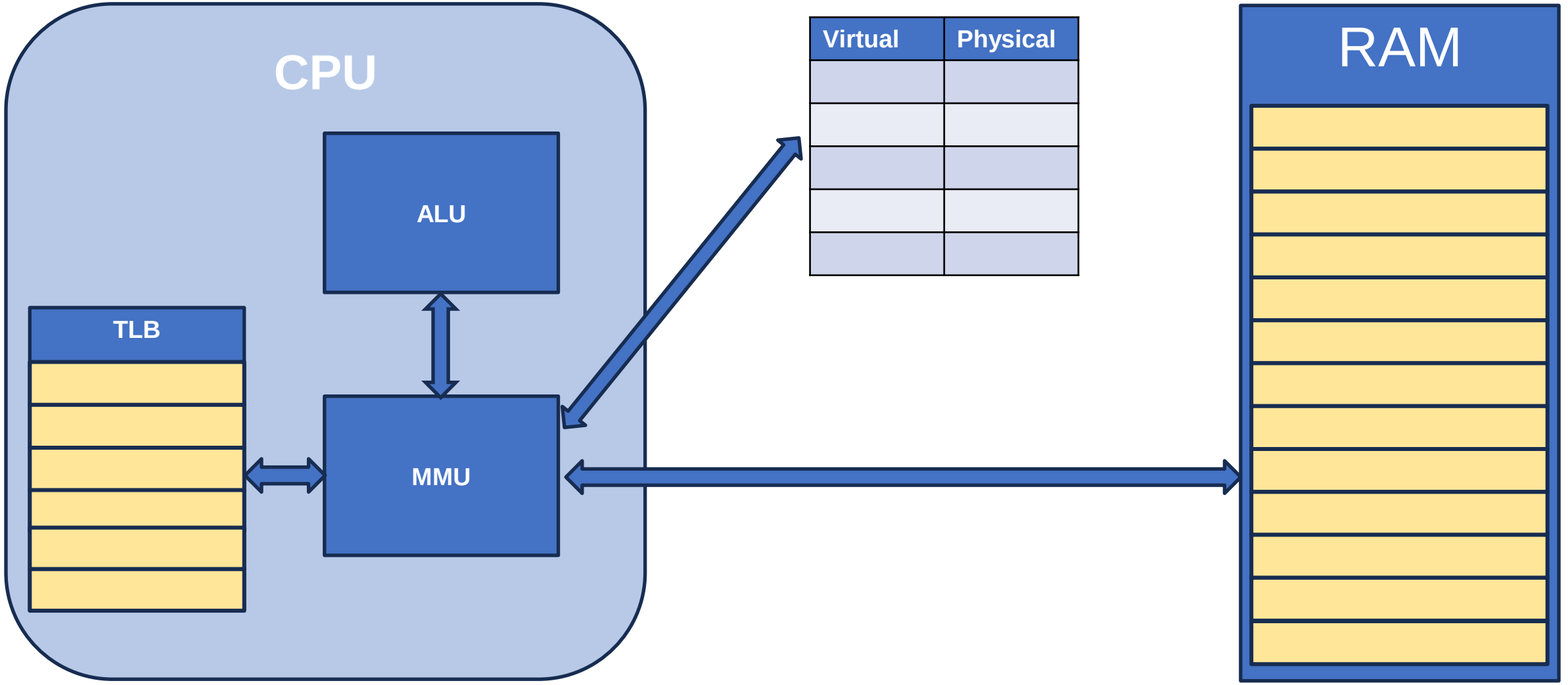
Virtual Memory



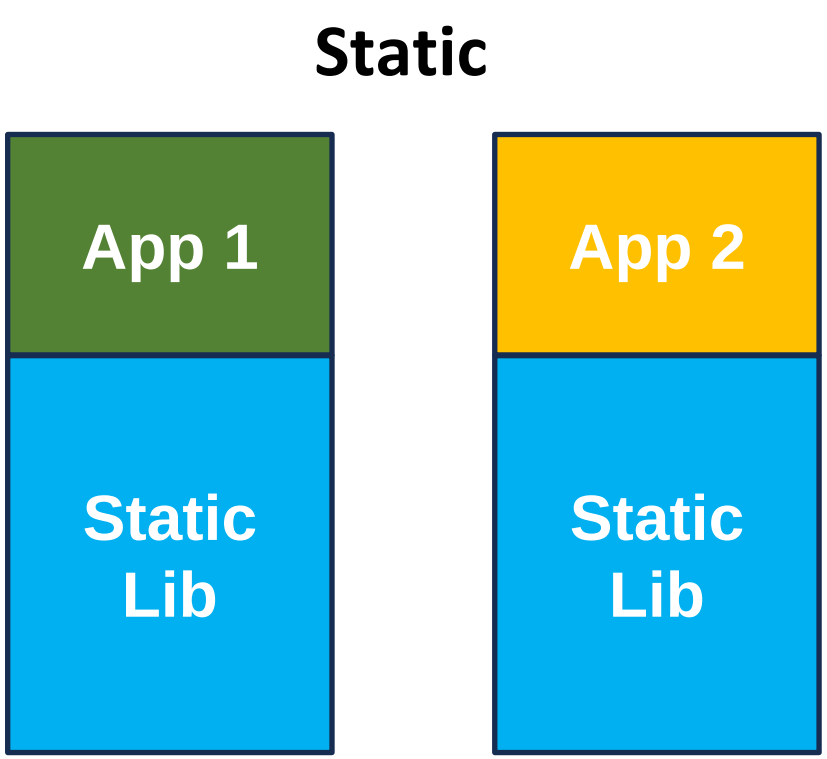
Variable's Address

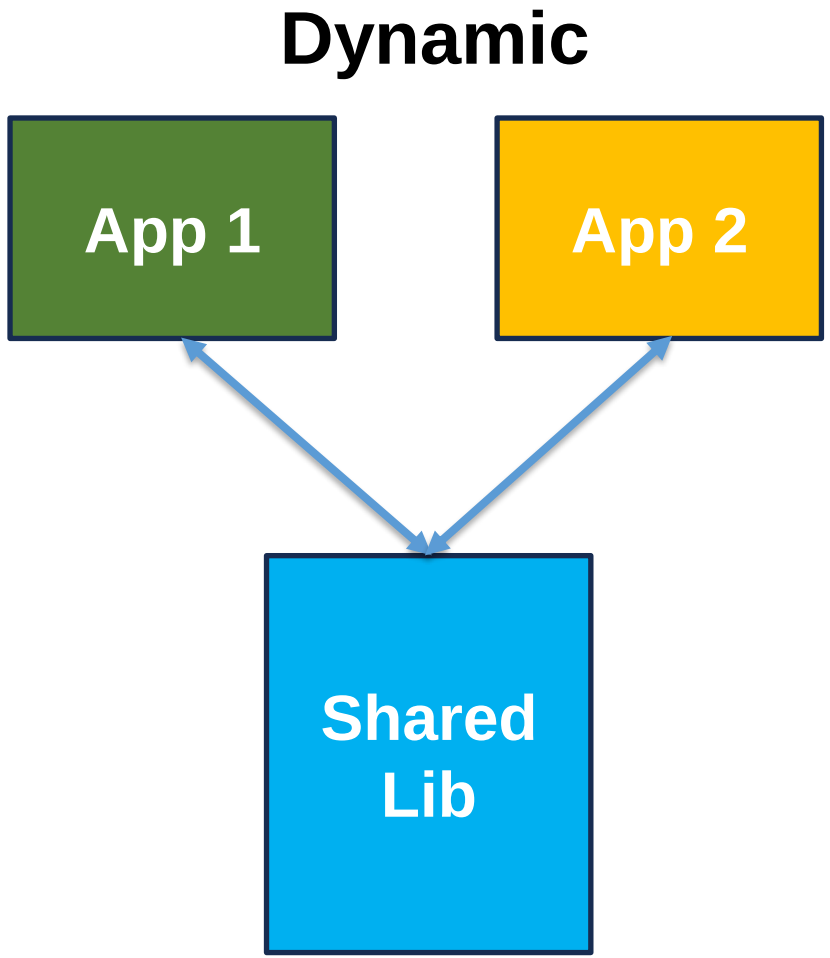


Hardware Architecture 101



Static vs Dynamic (Shared) libraries



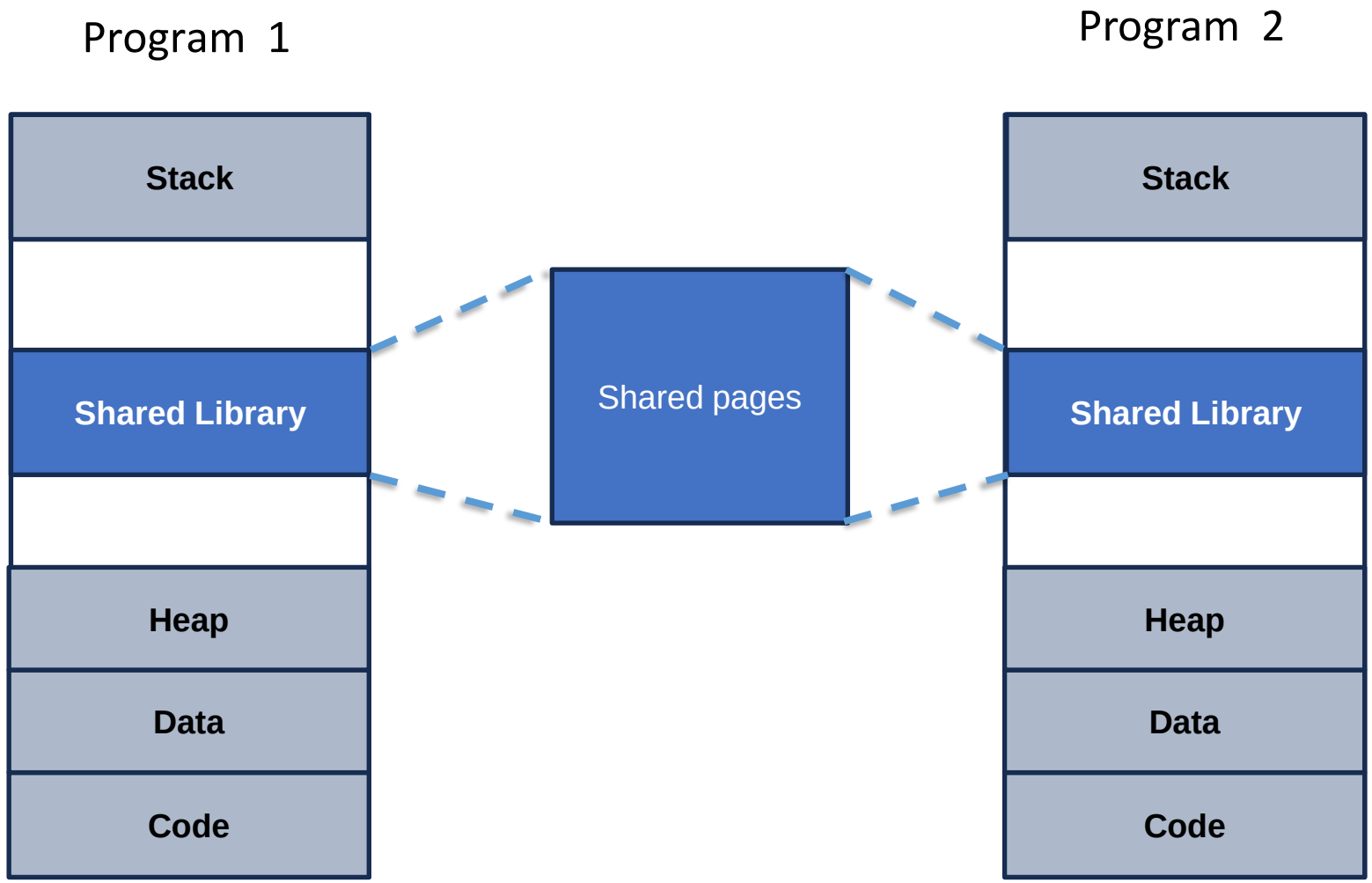


Dynamic (shared) libraries

Dynamic linker **ld.so**

Responsible for:
- loading shared libraries
- resolving symbols

Shared library's location:
- /lib
- /usr/lib



Page size

- **On x86_64 the typical page size is 4K**

Check Page Size from the shell:

$ getconf PAGESIZE

$ getconf PAGE_SIZE

Check Page Size from the *C* code:

```
#include <unistd.h>
#include <stdio.h>
int main() {
        long page_size = sysconf(_SC_PAGESIZE);
        printf("Page size: %ld bytes\n", page_size);
        return 0;
}
```

Virtual Memory Pros and Cons

- **Expanded Addressable Memory Space**
- **Isolation and Memory Protection**
- **On Demand Paging**
- **Efficient Use of RAM**
- **Memory Mapping and Sharing**

- **Performance Overhead**
- **Page Faults**
- **Storage Requirements**
- **Complexity**

Copy On Write

- Traditionally, upon fork(), all resources owned by the parent are duplicated and the copy is given to the child.
- In Linux, fork() is implemented through the use of ***copy-on-write*** pages.
- Rather than duplicate the process address space, the parent and the child can share a single copy
- The data, however, is marked in such a way that if it is written to, a duplicate is made and each process receives a unique copy.

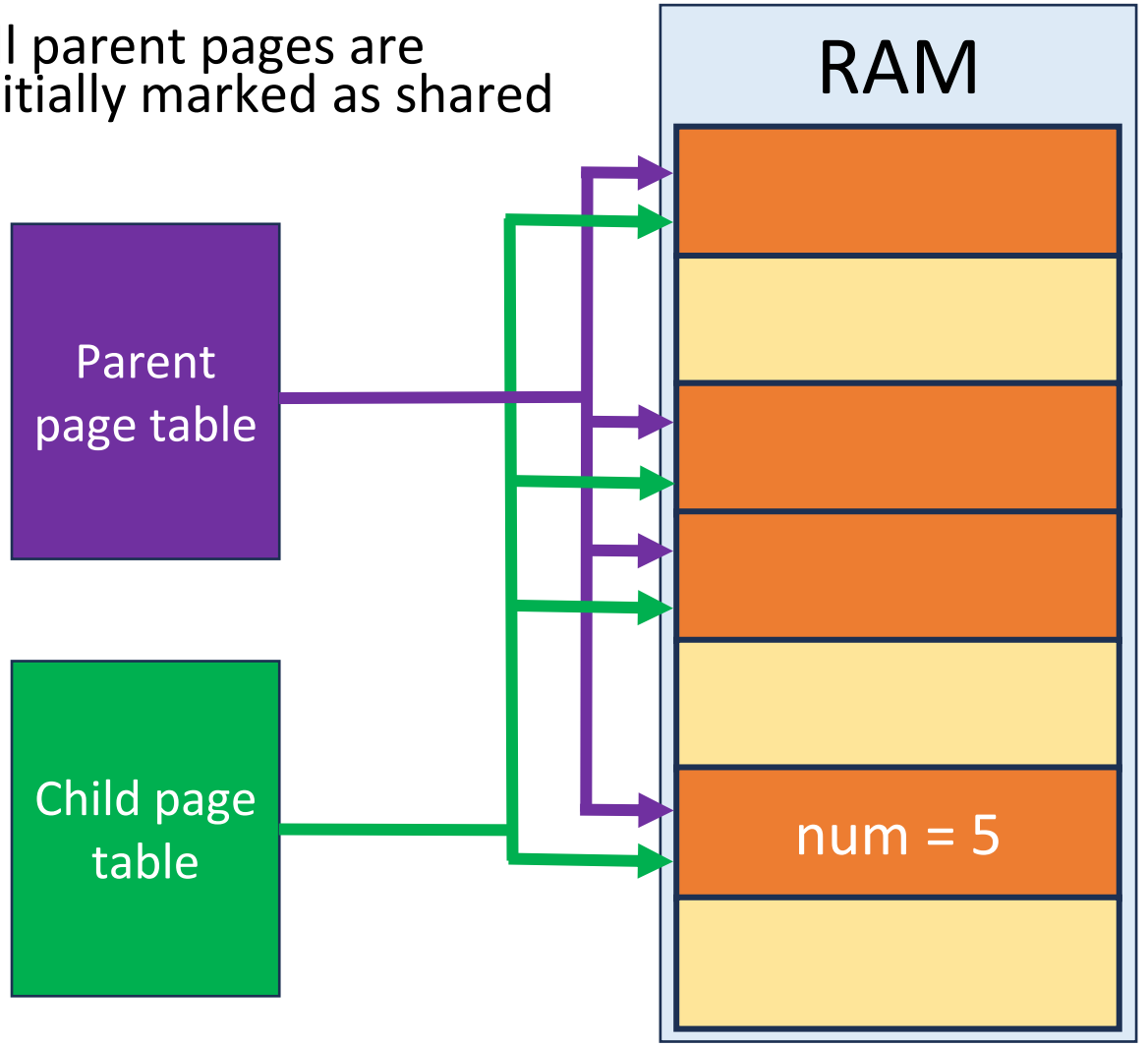
Copy On Write

```
int num = 5;
pid_t pid = fork();

if (pid == 0) { // Child process
    printf("Child num = %d\n", num);
} else { // Parent process
    printf("Parent num = %d\n", num);
}
```

Output

```
Child num = 5
Parent num = 5
```

- All parent pages are initially marked as shared



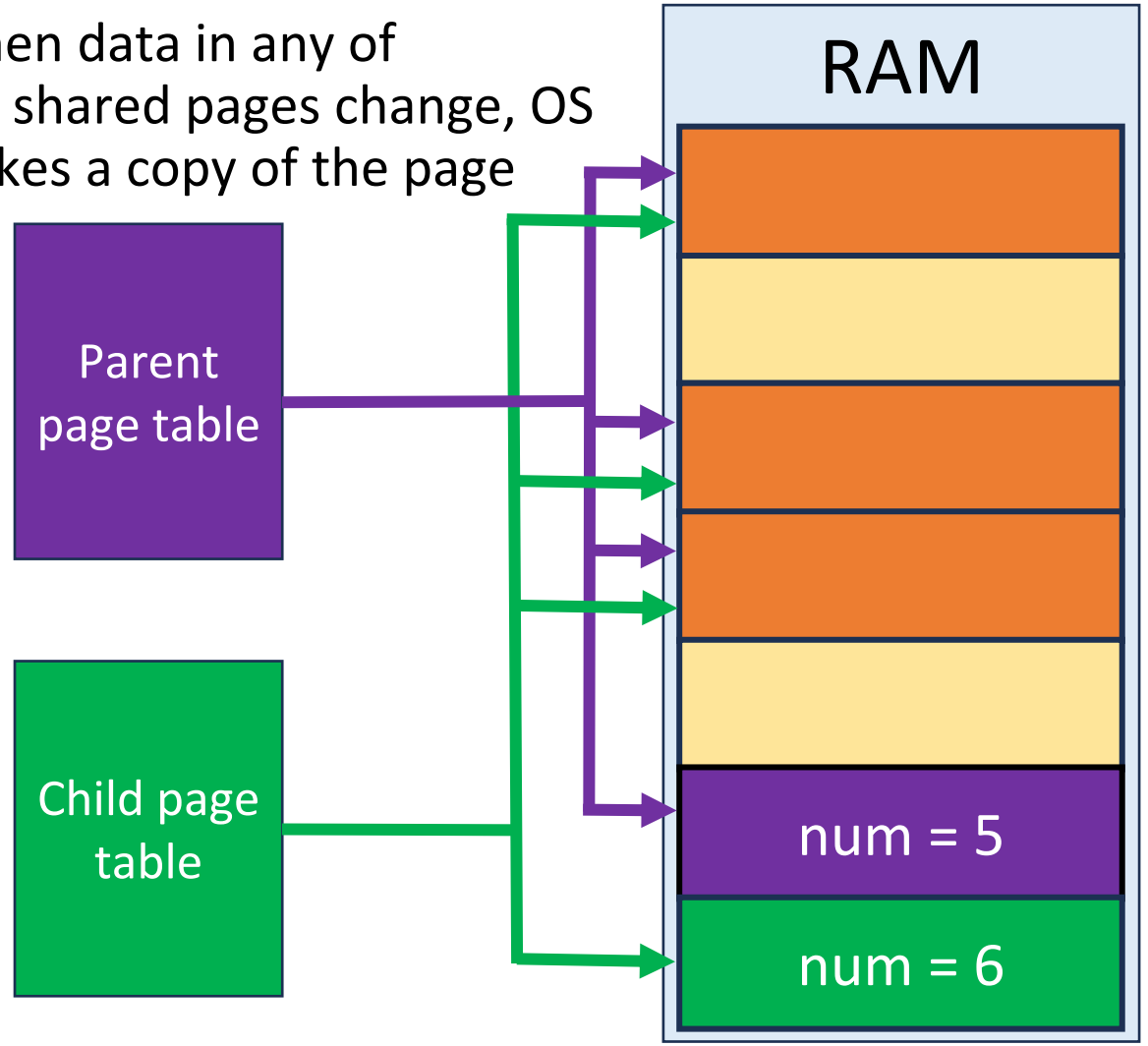
Copy On Write

```
int num = 5;
pid_t pid = fork();

if (pid == 0) { // Child process
    num++;
    printf("Child num = %d\n", num);
} else { // Parent process
    wait(NULL);
    printf("Parent num = %d\n", num);
}
```

Output

```
Child num = 6
Parent num = 5
```

- When data in any of the shared pages change, OS makes a copy of the page



Allocating Dynamic Memory

- The classic C interface for obtaining dynamic memory is malloc():

```
#include <stdlib.h>
void* malloc (size_t size);
```

Example:

```
char *p;
/* give me 4 KB! */
p = (char*)malloc (4096);
if (!p)
        perror ("malloc");
```

Allocating Dynamic Memory

- The classic C interface for obtaining dynamic memory is malloc():

```
#include <stdlib.h>
void * malloc (size_t size);
```

Example:

```
 struct my_struct* s;

 // allocate enough memory to hold a my_struct stucture
 s = (struct my_struct * )malloc (sizeof (struct my_struct));
 if (!s)
         perror ("malloc");
```

Allocating Arrays

```
int *x;
x = malloc (50 * sizeof (int));
if (!x) {
        perror ("malloc");
        return –1;
}
…
…
free(x);
x = NULL;
```

```
#include <stdlib.h>
void * calloc (size_t nr, size_t size);

int *y;
y = calloc (50, sizeof (int));
if (!y) {
        perror ("calloc");
        return –1;
}
…
…
free(y)
```

Freeing Dynamic Memory

- Unlike automatic allocations, dynamic allocations stay in the process's address space until manually freed.
- It's the programmer's job to free them.
- Both static and dynamic allocations are cleared when the process exits.

```
#include <stdlib.h>
void free (void *ptr);
```

Resizing Allocations

- A successful call to realloc() resizes the region of memory pointed at by ptr to a new size of size bytes.

- If ptr is NULL, the result of the operation is the same as a fresh malloc()
- If size is 0, the effect is the same as an invocation of free() on ptr

```
#include <stdlib.h>
void * realloc (void *ptr, size_t size);
```

Resizing Allocations

```
int *arr = malloc (3 * sizeof (int)); // Allocate memory for 3 ints
if (! arr) {
        perror ("malloc");
        return –1;
}
arr[0] = 0; arr[1] = 1; arr[2] = 2;

// Resize memory to hold 5 integers
arr = realloc (arr, 5 * sizeof (int));
if (!arr) {
        perror ("realloc");
        return –1;
}
arr[3] = 3; arr[4] = 4;

free(arr); // Free allocated memory
```

Alignment

- Data alignment refers to the way data is arranged in memory
- A memory address A is said to be n-byte aligned when
  - ✓ n is a power-of-2
  - ✓ A is a multiple of n.
- Most processors operate on words and can only access memory addresses that are word-size-aligned
- Similarly, memory management units deal only in page-size-aligned addresses.

Alignment

- A variable located at a memory address that is a multiple of its size is said to be *naturally aligned*.
- For example, a 32-bit variable is naturally aligned if it is located in memory at an address that is a multiple of 4—in other words, if the address's lowest two bits are 0.
- Rules pertaining to alignment derive from hardware and thus differ from system to system.
- Some machine architectures have very stringent requirements on the alignment of data. Others are more lenient.
- This incurs a performance hit and sacrifices atomicity, but at least the process isn't terminated.

Struct Alignment

Complex data types possess alignment requirements

1. The alignment requirement of a structure is that of its largest constituent type.

For example, if a structure's largest type is a 32-bit integer that is aligned along a 4-byte boundary, the structure must be aligned along at least a 4-byte boundary as well.

2. Structures introduce the need for padding, which is used to ensure that each constituent type is properly aligned to that type's own requirement.

For example, if a char (with a probable alignment of 1 byte) finds itself followed by an int (with a probable alignment of 4 bytes), the compiler will insert 3 bytes of padding between the two types to ensure that the int lives on a 4-byte boundary.

Struct Alignment

3. The alignment requirement of a union is that of the largest unionized type.
4. The alignment requirement of an array is that of the base type. Thus, arrays carry no requirement beyond a single instance of their type. This behavior results in the natural alignment of all members of an array.

Allocating aligned memory

- For the most part, the compiler and the C library transparently handle alignment concerns.
- POSIX decrees that the memory returned via malloc(), calloc(), and realloc() be properly aligned for use with any of the standard C types.
- On Linux, these functions always return memory that is aligned along an 8-byte boundary on 32-bit systems and along a 16-byte boundary on 64-bit systems

Allocating aligned memory

Occasionally, programmers require dynamic memory aligned along a larger boundary, such as a page.

```
#include <stdlib.h>
int posix_memalign (void **memptr,
                    size_t alignment,
                    size_t size);
```

- A successful call to posix_memalign() allocates size bytes of dynamic memory, ensuring it is aligned along a memory address that is a multiple of alignment.

Allocating aligned memory

```
char *buf;
int ret;
/* allocate 1 KB along a 256-byte boundary */
ret = posix_memalign (&buf, 256, 1024);
if (ret) {
        fprintf (stderr, "posix_memalign: %s\n",
        strerror (ret));
        return -1;
}
/* use 'buf'... */

free (buf);
```

Stack-Based Allocations

- There is no reason, that a programmer cannot use the stack for dynamic memory allocations.

```
#include <alloca.h>
void * alloca (size_t size);
```

- Usage is identical to malloc(), but you do not need to (indeed, must not) free the allocated memory.
- Upon return, the memory allocated with alloca() is automatically freed as the stack unwinds back to the invoking function.
- This means you cannot use this memory once the function that calls alloca() returns!

Allocating aligned memory

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

int main() {
    int n = 5;
    int *arr = alloca(n * sizeof(int));  // Allocate memory on the stack

    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;  // Initialize array
    }

    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);  // Print array
    }

    // No need to free, memory is automatically released when function exits
    return 0;
}
```

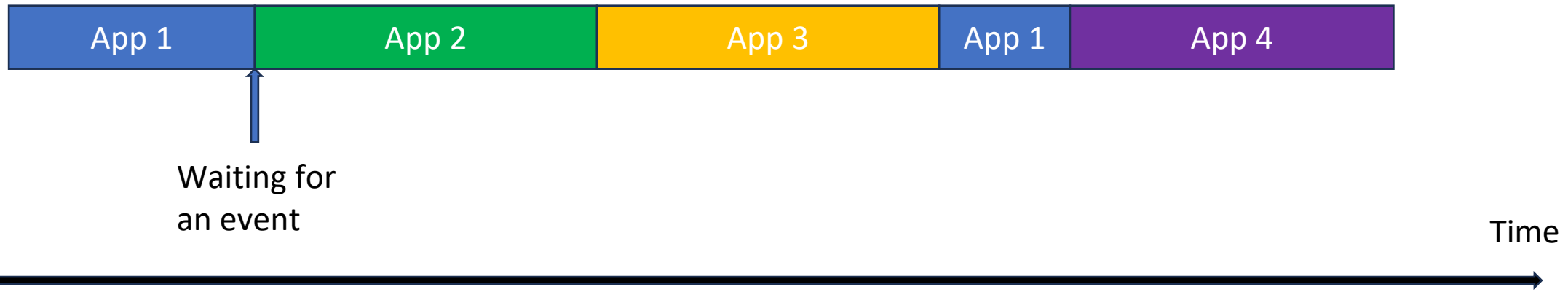
Choosing a Memory Allocation Mechanism

Allocation approach	Pros	Cons
malloc()	Easy, simple, common.	Returned memory not necessarily zeroed.
calloc()	Makes allocating arrays simple, zeros returned memory.	Convoluted interface if not allocating arrays.
realloc()	Resizes existing allocations.	Useful only for resizing existing allocations.
brk() and sbrk()	Provides intimate control over the heap.	Much too low-level for most users.
Anonymous memory mappings	Easy to work with, sharable, allow developer to adjust protection level and provide advice; optimal for large mappings.	Suboptimal for small allocations; malloc() automatically uses anonymous memory mappings when optimal.
alloca()	Very fast allocation, no need to explicitly free memory; great for small allocations.	Unable to return error, no good for large allocations, broken on some Unix systems.

Multitasking

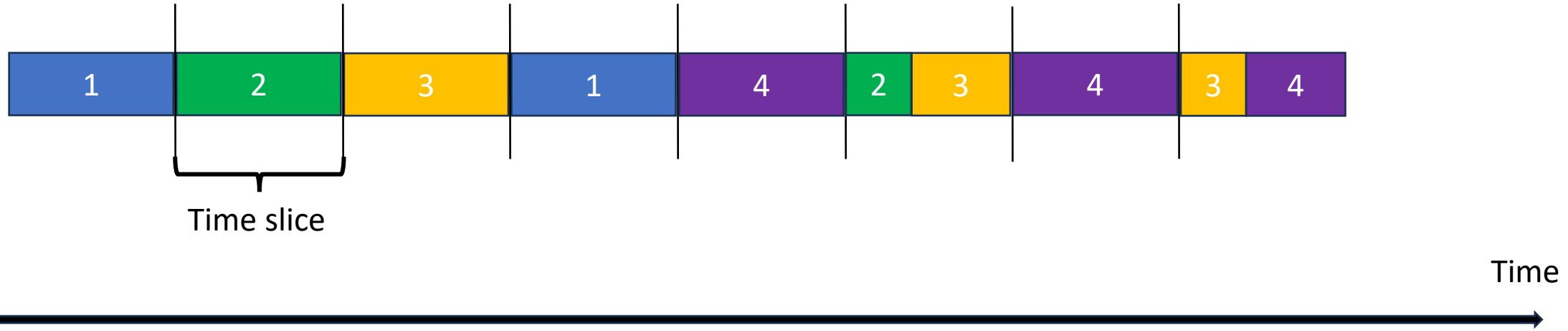
Multiprogramming: Single task

- Simple
- Processes run one after another
- Each process executes till the end
- When the CPU is idle, switch to another task



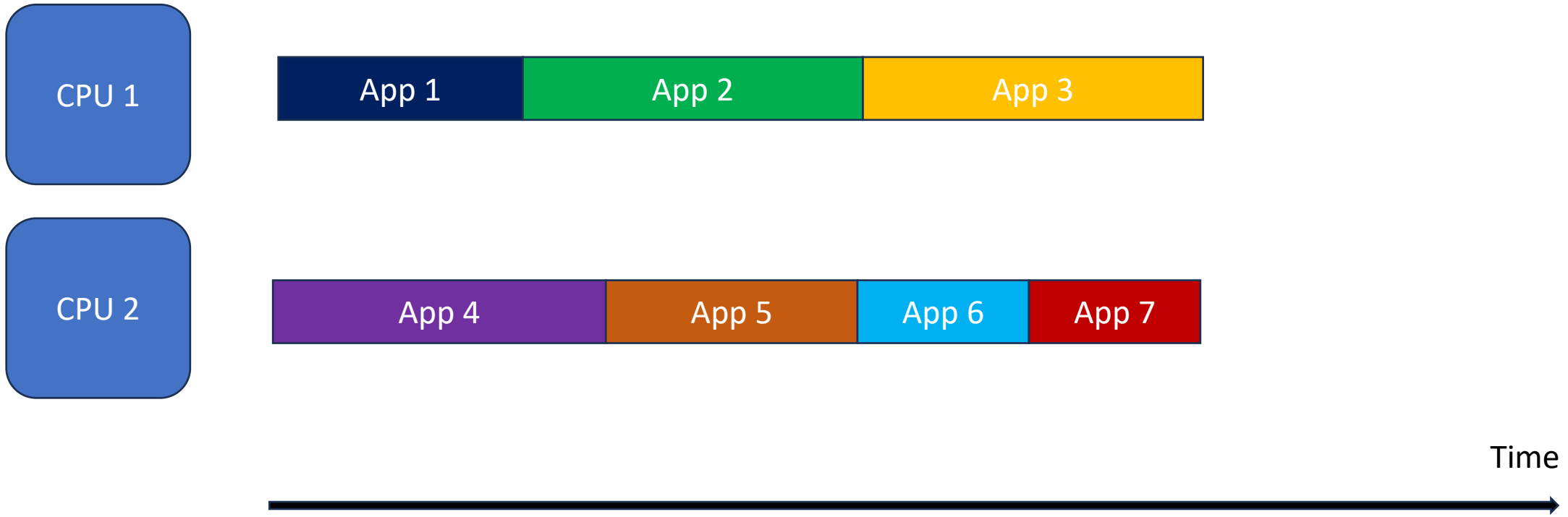
Multitasking: Time sharing

- Time sliced
- Each process executes within a slice
- Impression of concurrently running apps
- No starvation

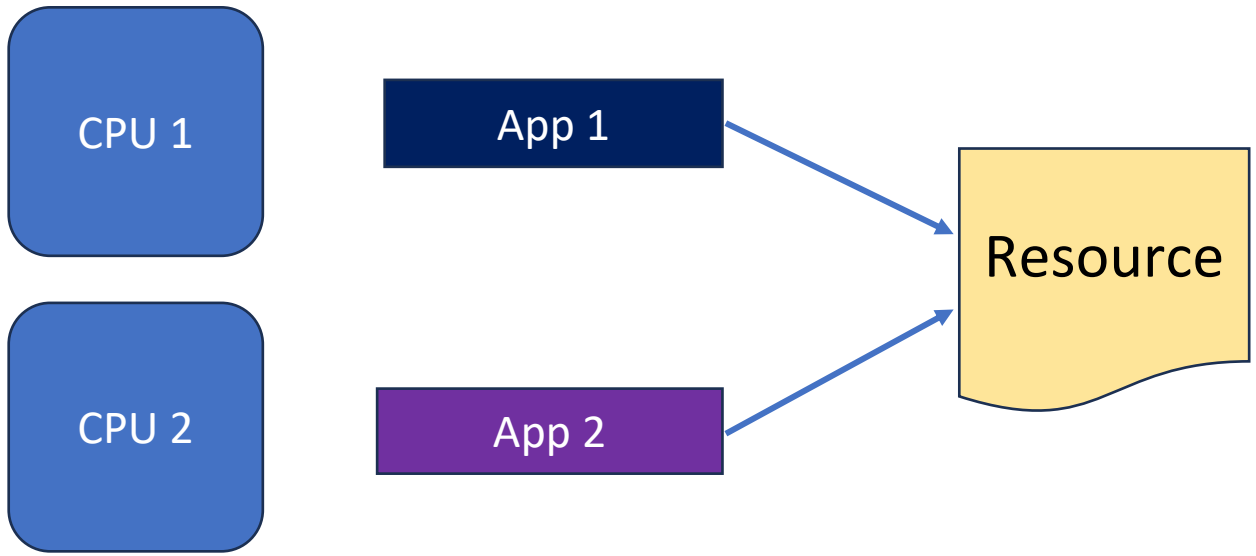


Multiprocessor

- Each processor executes independent apps



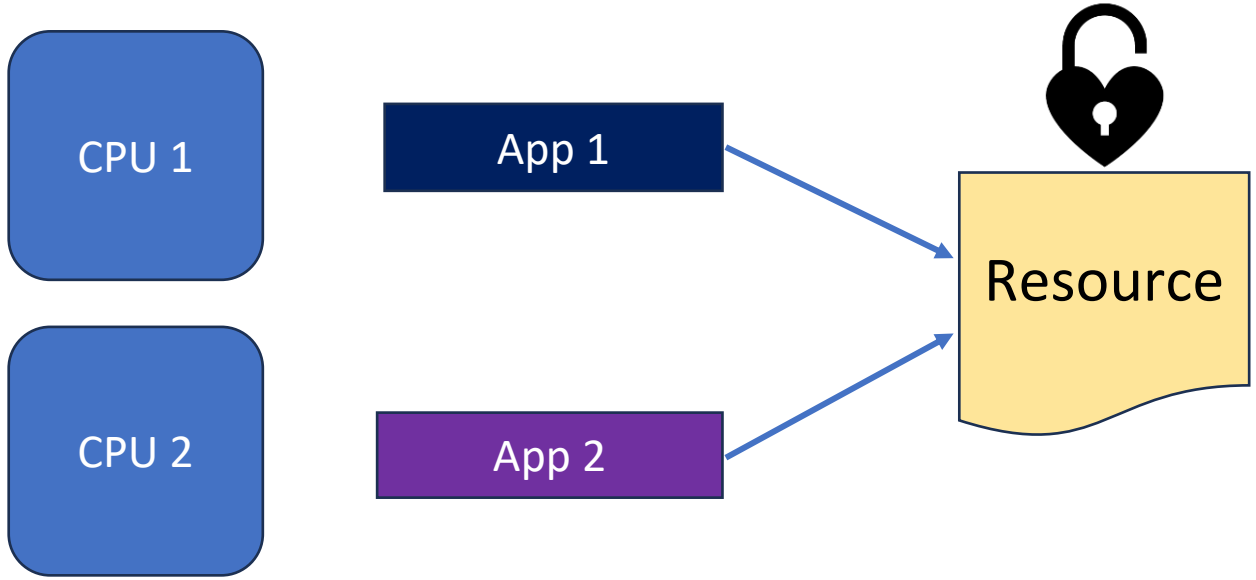
Race conditions



Example

- App1 and App4 want to write to the same file simultaneously
- This is a **race condition**
- Need to **synchronize**

Race conditions



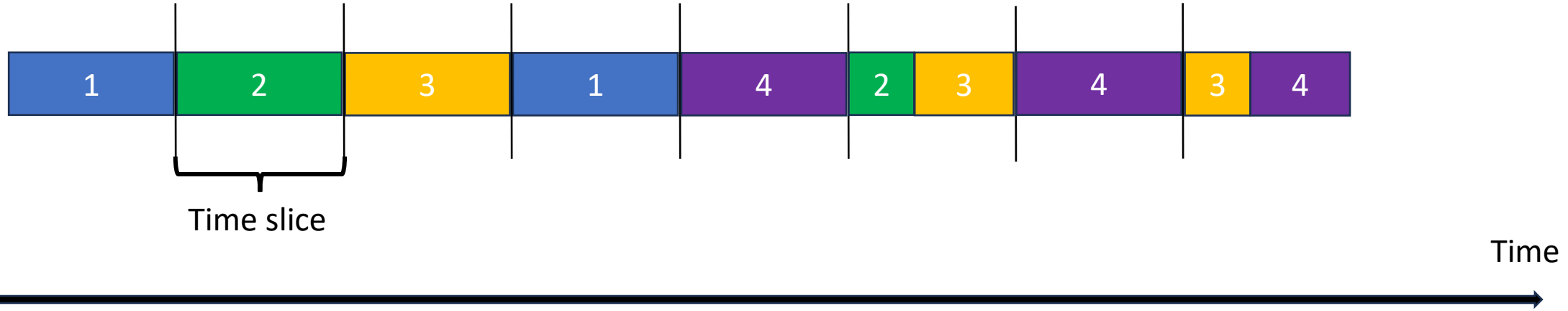
Solution

Shared resource is associated with a lock

Steps

1. App1 locks the resource
2. App1 accesses the resource
3. App2 waits
4. App1 unlocks the resource
5. App2 can lock and access the resource

What if there is only one CPU in the mode of Multitasking? Can a race condition happen?



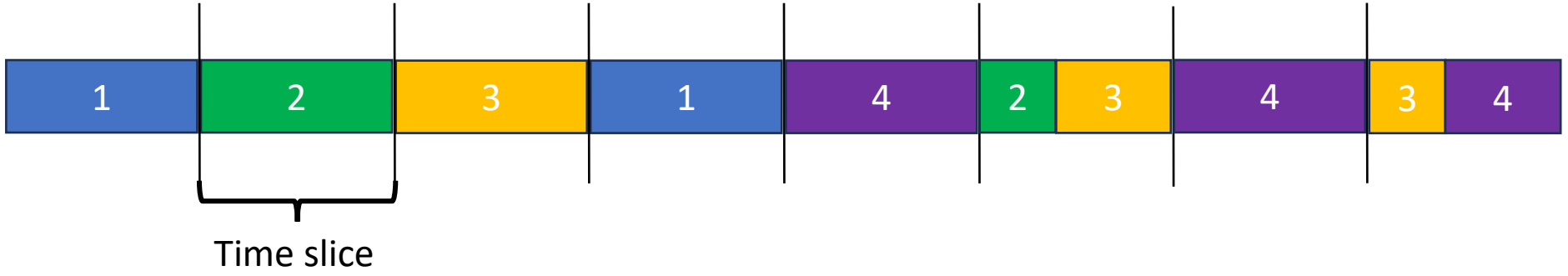
The Linux implementation of Threads

- Each thread has a unique task_struct and appears to the kernel as a normal process
- Threads just happen to share resources, such as an address space, with other processes

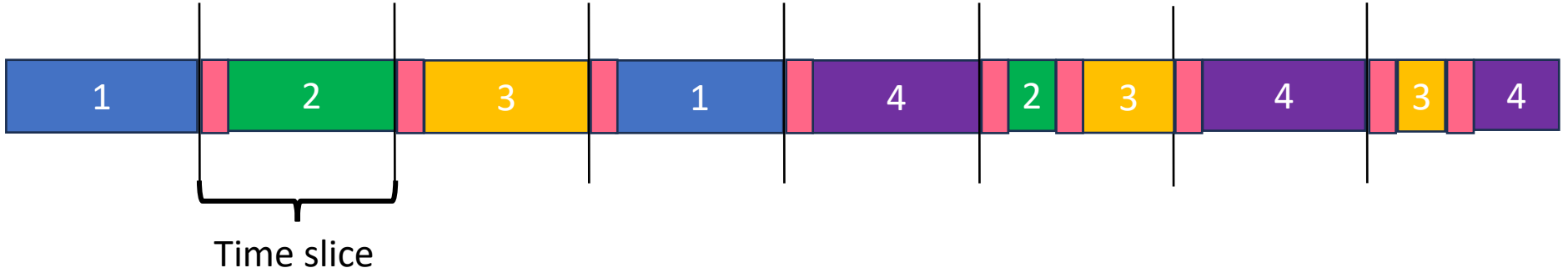
- clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

Context switch

- Ideal multitasking



- Real multitasking



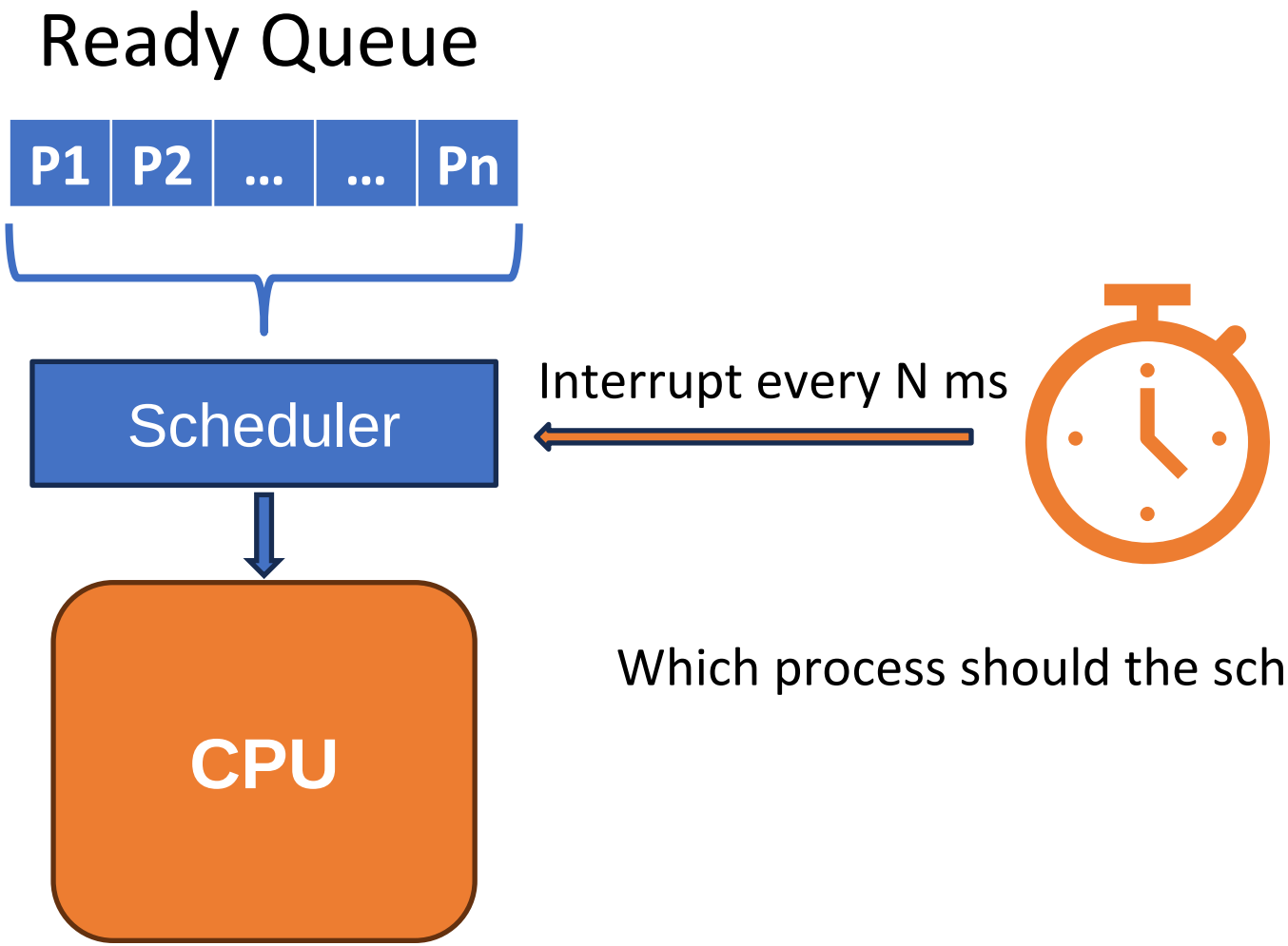
Time

Context Switching Overheads

- Direct factors
  - Timer interrupts handling latency
  - Saving/restoring context
  - Finding the next process to execute
- Indirect factors
  - Loss of cache locality
  - Processor pipeline flush

Schedulers

Choosing the next process



Which process should the scheduler choose next?

Types of processes

- I/O bound
  - Has small bursts of CPU activity and then waits for I/O (e.g. Text editor)
  - Affects user interaction. We want these processes to have highest priority

- CPU bound
  - Hardly any I/O, mostly CPU activity (e.g. gcc, 3D rendering, calculating)
  - Could do with lower priorities

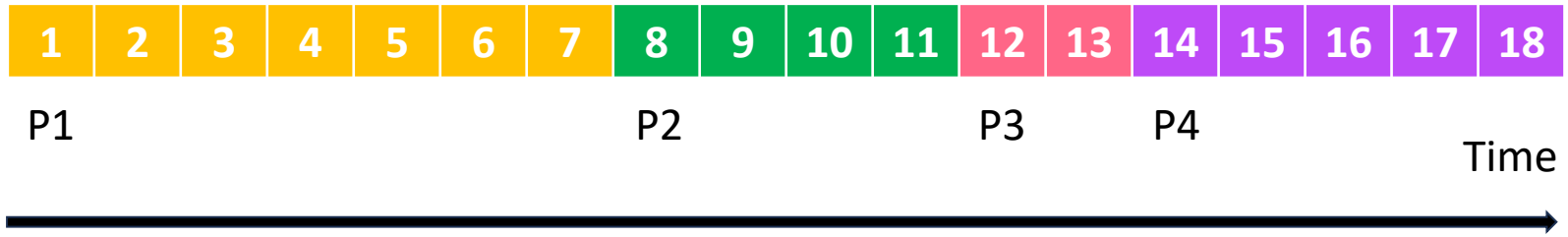
Scheduling Criteria

Maximize CPU utilization	• CPU should not be idle
Maximize throughput	• Complete as many process as possible per unit time
Minimize turnaround time	• Time from start to completion
Minimize response time	• CPU should response quickly (from ready queue -> to execution)
Minimize waiting time	• Process should not wait long in the ready queue (sum of all time waiting in ready queue)
Fairness	• Give each process a fair share of CPU

First Come First Serve - FCFS

- First job that requires CPU gets it
- Non preemptive
  - Process continues till the burst cycle ends

Process	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	0	2
P4	0	5

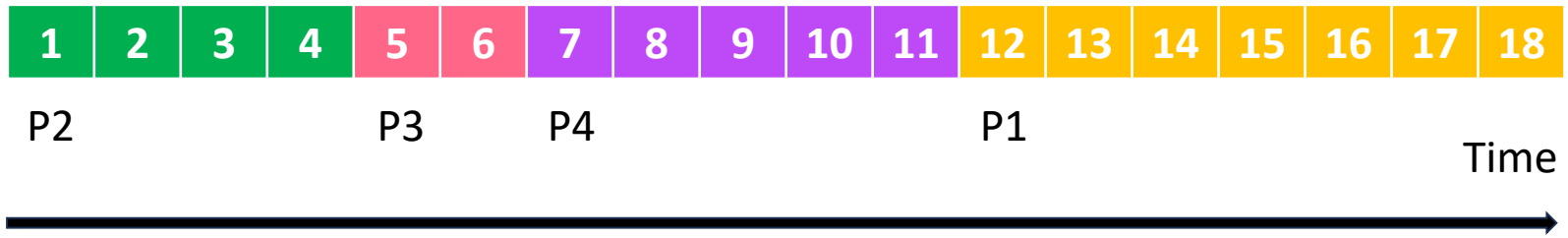


- Average Waiting Time = (0 + 7 + 11 + 13) / 4 = 7.75
- Average Response Time = (0 + 7 + 11 + 13) / 4 = 7.75

First Come First Serve - FCFS

- Order of scheduling matters

Process	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	0	2
P4	0	5



- Average Waiting Time = (11 + 0 + 4 + 6) / 4 = 5.25
- Average Response Time = (11 + 0 + 4 + 6) / 4 = 5.25

FCFS Pros & Cons

Advantages

- Simplicity
- Fair (if process is not endless)
- Predictability

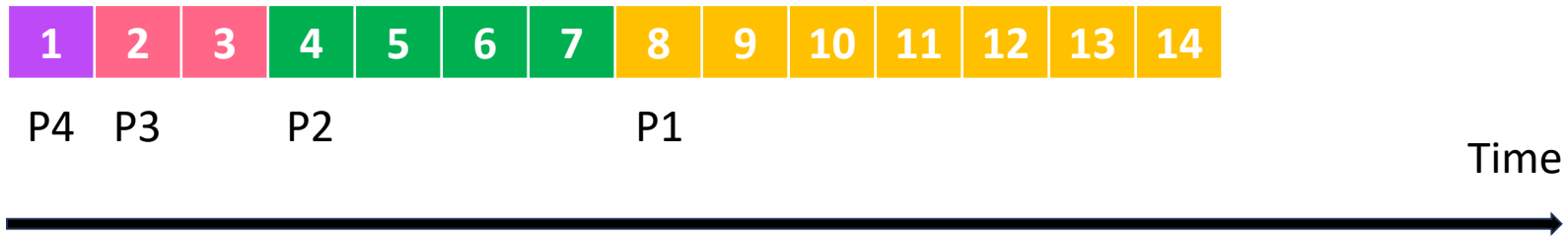
Disadvantages

- Waiting time depends on arrival order
- Convoy effect

Shortest Job First - SJF

- Schedule process with the shortest burst time
- Without preemption
  - Process continues till the burst cycle ends

Process	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	0	2
P4	0	1

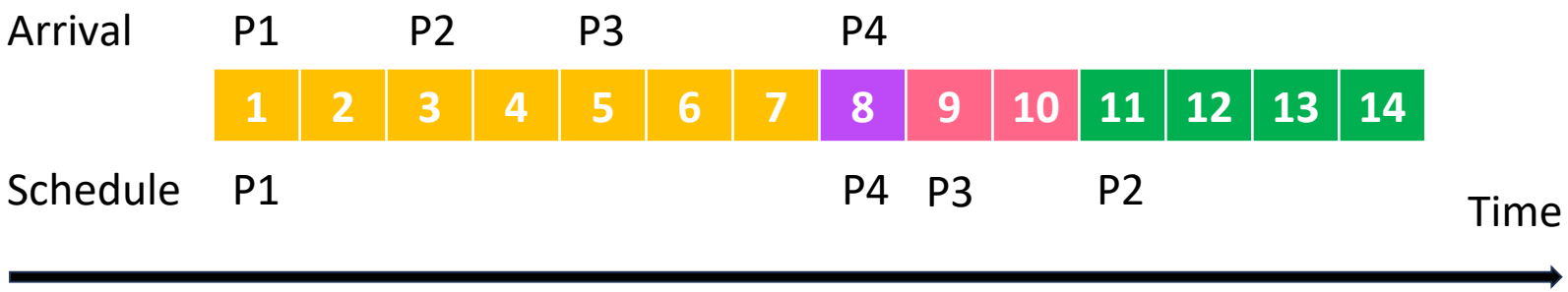


- Average Waiting Time = (7 + 3 + 1 + 0) / 4 = 2.75
- Average Response Time = Average Waiting Time

Shortest Job First - SJF

- Schedule process with the shortest burst time
- Without preemption
  - Process continues till the burst cycle ends

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	2
P4	7	1



- Average Waiting Time = (0 + 8 + 4 + 0) / 4 = 3
- Average Response Time = (0 + 0 + 8 + 4) / 4 = 3

SJF Pros & Cons

Advantages

- Minimum average wait time
- Average response time decreases

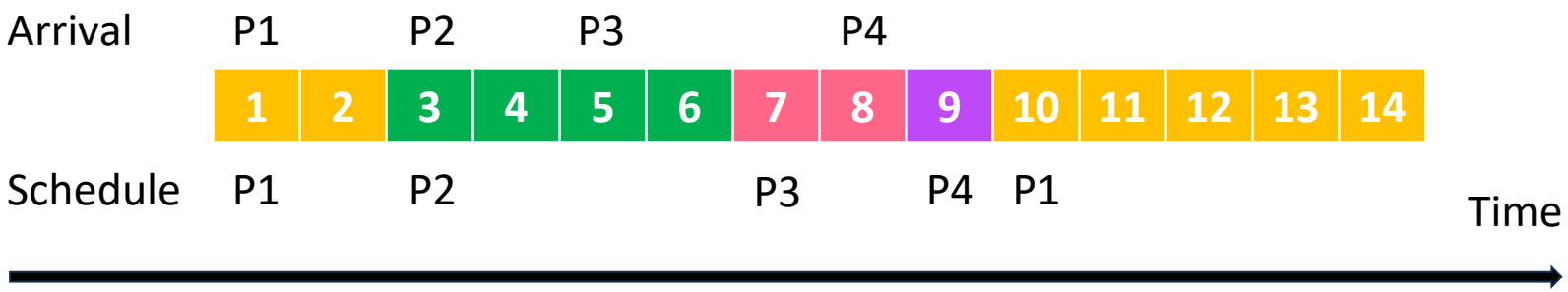
Disadvantages

- Not practical: difficult to predict burst time
- Long jobs may starve

Shortest Remaining Time First - SRTF

- Schedule process with the shortest remaining time
- With preemption

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	2
P4	7	1

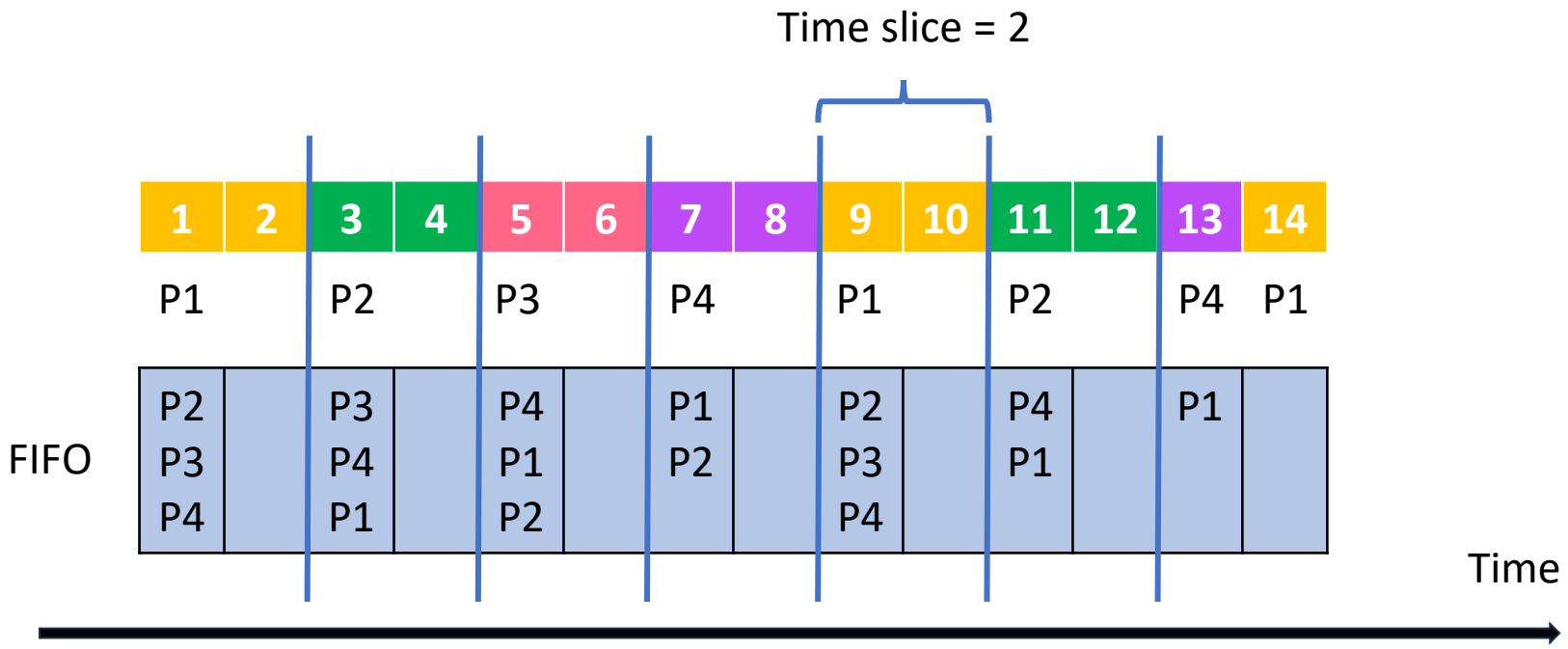


- Average Waiting Time = (7 + 0 + 2 + 1) / 4 = 2.5
- Average Response Time = (0 + 0 + 2 + 1) / 4 = 0.75

Round Robin - RR

- Schedule process with a period "Time Slice"

Process	Arrival Time	Burst Time
P1	0	5
P2	0	4
P3	0	2
P4	0	3

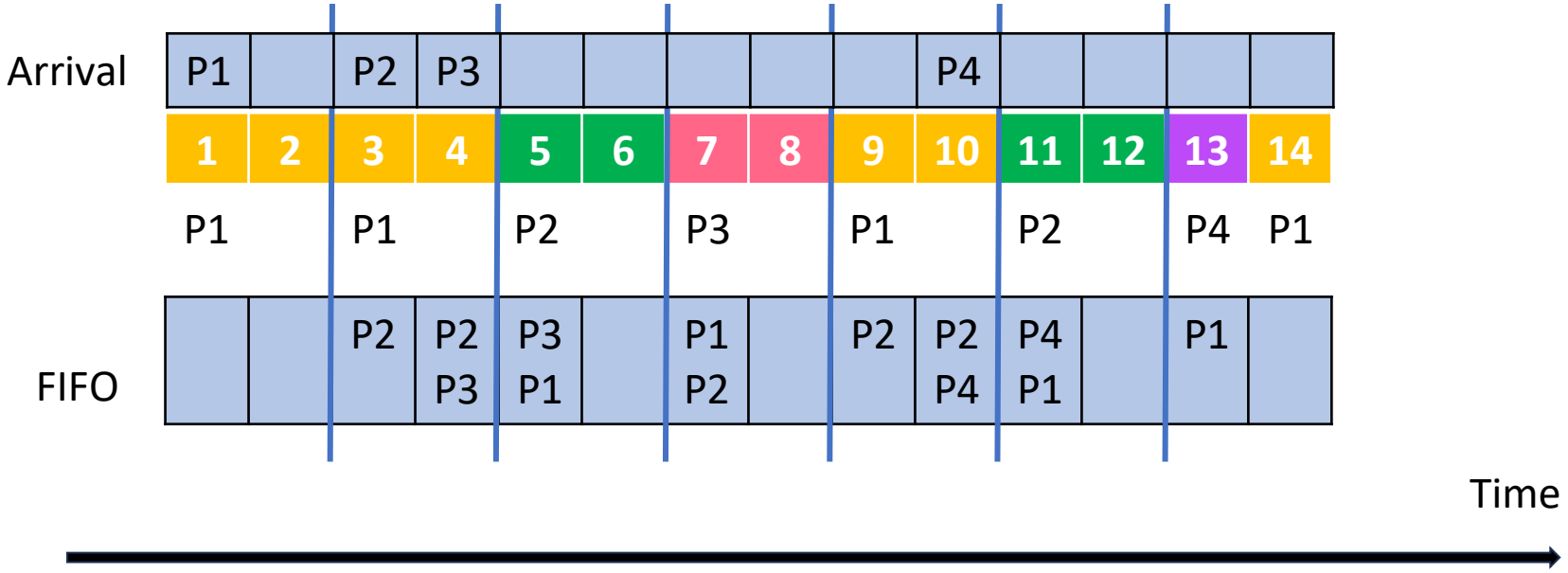


- Average Waiting Time = (9 + 8 + 4 + 10) / 4 = 7.75
- Average Response Time = (0 + 2 + 4 + 6) / 4 = 3

Round Robin - RR

- Schedule process with a period "Time Slice"

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

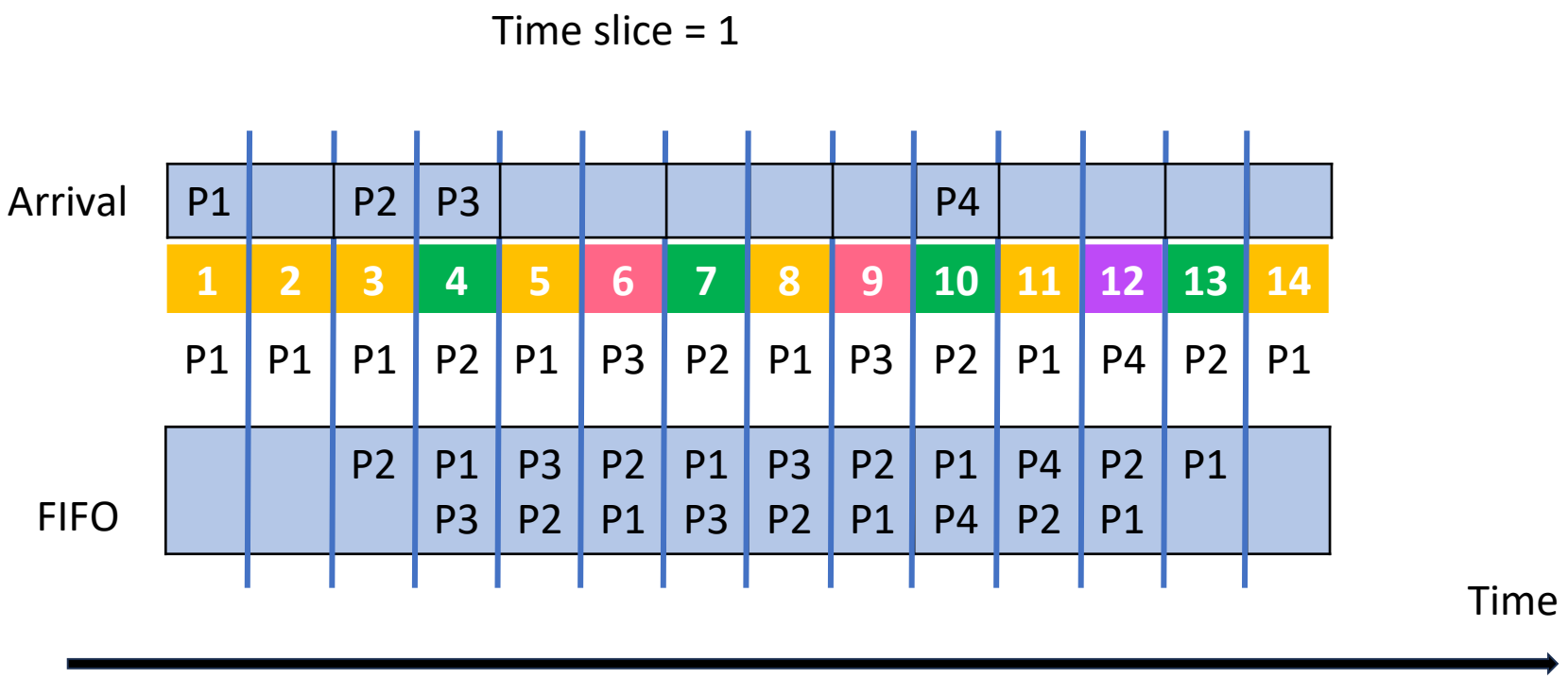


- Average Waiting Time = (7 + 6 + 3 + 3) / 4 = 4.75
- Average Response Time = (0 + 2 + 3 + 3) / 4 = 2

Round Robin - RR: Smaller time slice

- Schedule process with a period "Time Slice"

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

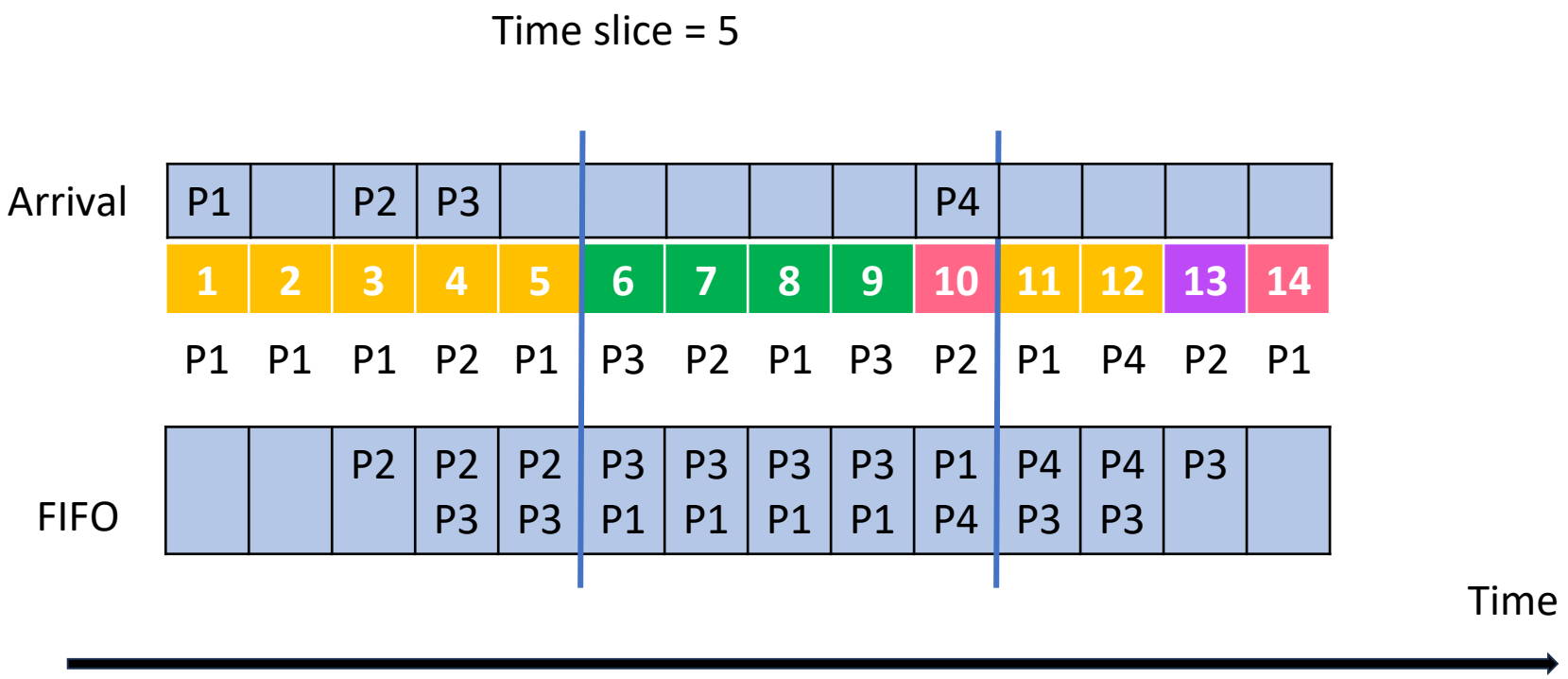


- Average Waiting Time = (7 + 7 + 4 + 2) / 4 = 5
- Average Response Time = (0 + 1 + 2 + 2) / 4 = 1.25

Round Robin - RR: Smaller time slice

- Schedule process with a period "Time Slice"

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1



- Average Waiting Time = (5 + 3 + 9 + 3) / 4 = 5
- Average Response Time = (0 + 3 + 6 + 2) / 4 = 2.25

RR Time slice duration

- Short
  - Good – processes need not wait long before they are scheduled
  - Bad – context switches overhead increase

- Long
  - Bad – processes no longer appear to execute concurrently
  - Bad – may degrade system performance

RR Pros & Cons

Advantages

- Fair
- Low avg wait time
- Faster response time

Disadvantages

- Increase context switch
- High avg wait time, when burst times have equal lengths

Priority Based Scheduling

- Each process is assigned a priority
  - A priority is a number in a range (for instance 0 to 255)
  - A small number would mean high priority while a large number would mean low priority
- Scheduling policy: pick the process in ready queue having the highest priority
- **Advantage:** mechanism to provide relative importance to processes
- **Disadvantage:** could lead to starvation of low priority processes

Starvation

Process	Arrival time	Burst Time
P1	0	8
P2	2	4
P3	3	2
P4	9	1

- Time slice = 1
- Processes re-arrive every 15 cycles
- P4 is a low priority process

- **Low priority process may never get a chance to execute**

Dealing with Starvation

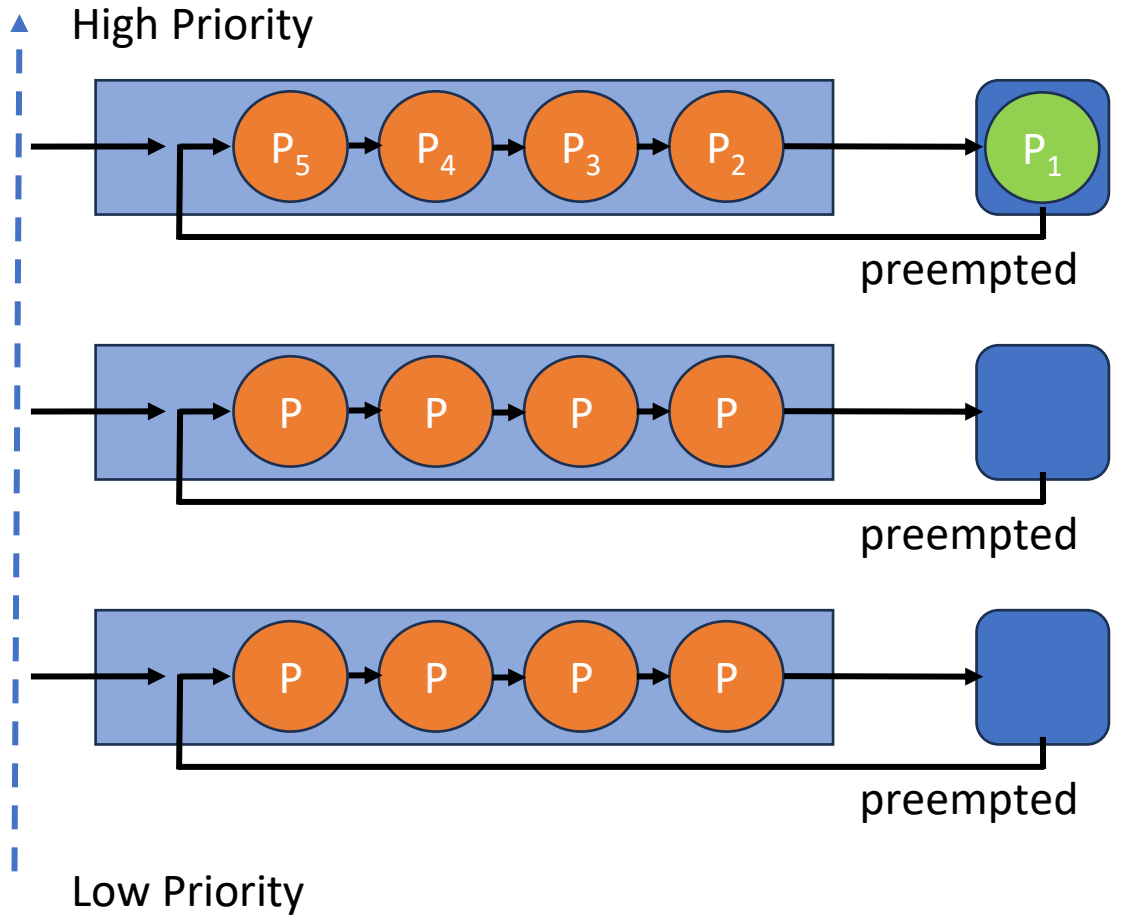
- Scheduler dynamically adjust priority of processes to ensure that they all eventually execute
- Several techniques are possible. For example:
  - Every process is given a base priority
  - After every time slot increase the priority of all other processes
    - This ensures that even a low priority process will eventually execute
  - After a process executes, its priority reset

Priority Types

- Static Priority
  - Typically set at start of execution
  - If not set by user, there is a default value (base priority)
- Dynamic Priority
  - Scheduler can change the process priority
  - Decrease priority of a process to give another process a chance to execute
  - Increase priority for I/O bound processes

Multilevel Queues

- Process is assigned to a priority classes
- Each class has it’s own ready queue
- Scheduler picks the highest priority class which has at least one ready process
- Selection of a process within the class could have it’s own policy
  - Typically round robin (but can be changed)
  - High priority classes can implement first come first serve in order to ensure quick response time for critical tasks



Multilevel Queues

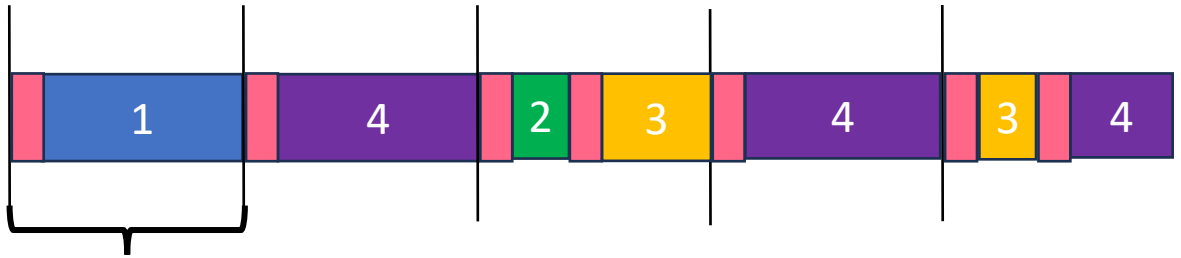
- Scheduler can adjust time slice based on the class picked
  - I/O bound processes can be assigned to higher priority class with longer time slice
  - CPU bound processes can be assigned to lower priority classes with shorter time slices

- Disadvantages:
  - Class of a process must be assigned apriori

Multilevel Feedback Queues

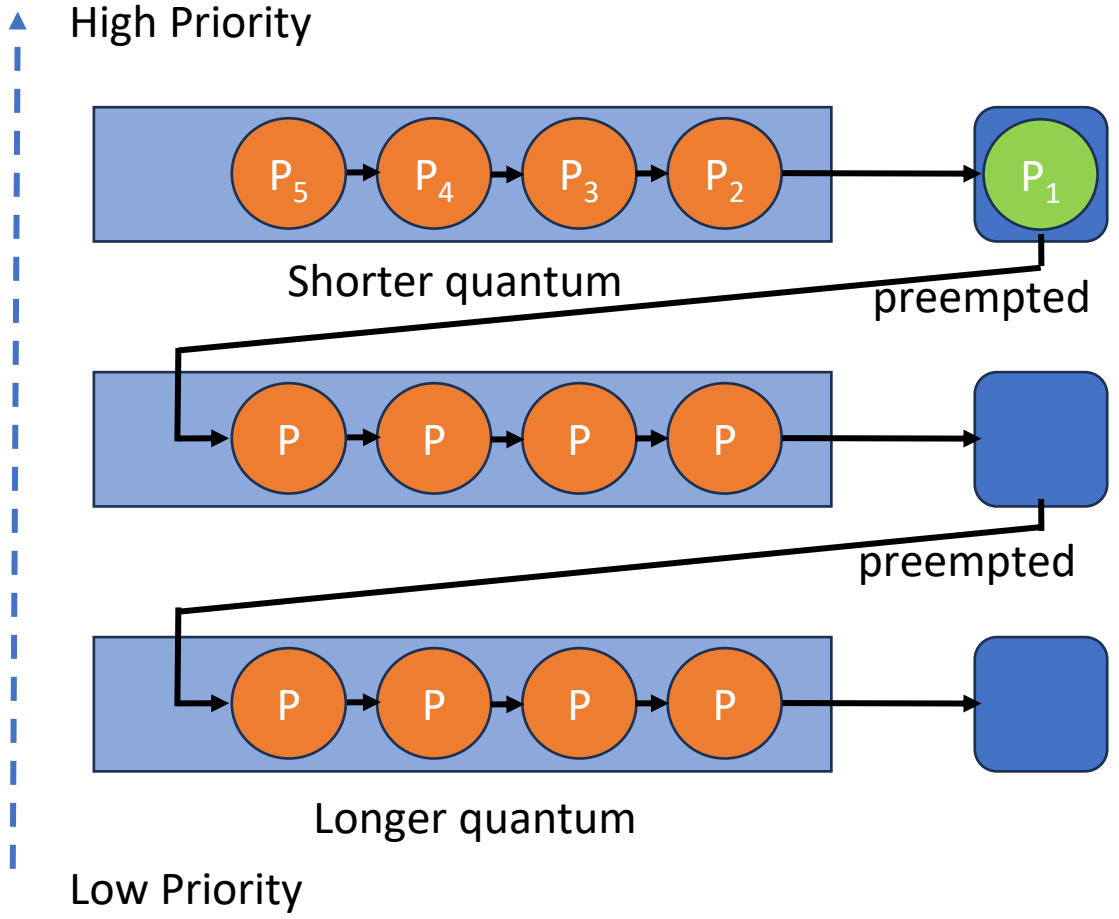
- Process dynamically moves between priority classes based on its CPU/IO activity

- Basic observations
  - CPU bound processes are likely to use its entire time slice
  - I/O bound processes may not use entire time slice



Multilevel Feedback Queues

- All processes start in the highest priority class
- If it finishes its time slice (likely CPU bound)
  - Move to the next lower priority class
- If it does not finish its time slice (likely I/O bound)
  - Keep in the same priority class
- As with any other priority based scheduling scheme, starvation needs to be dealt with



Gaming the system

- Programmer can write code that will force the system to block on some low-latency I/O operation (e.g., sleep for a few milliseconds) just before the quantum expires
- That way, the process will be rewarded for not using up its quantum even if it repeatedly uses up a large chunk of it.

The History of Linux Schedulers

- O(n) scheduler
  - Linux 2.4-2.6
- O(1) scheduler
  - Linux 2.6 - 2.6.22
- CFS scheduler
  - Linux 2.6.23 onwards

O(N) Scheduler

- At every context switch
  - Scan the list of runnable processes
  - Compute priorities
  - Select the best process to run

- O(n), where n is number of runnable processes -> not scalable!
  - Scalability issues observed when Java was introduced (JVM spawns many tasks)
  - Context switch is too much time consuming operation

- Used a global run-queue in SMP systems
  - Again not scalable

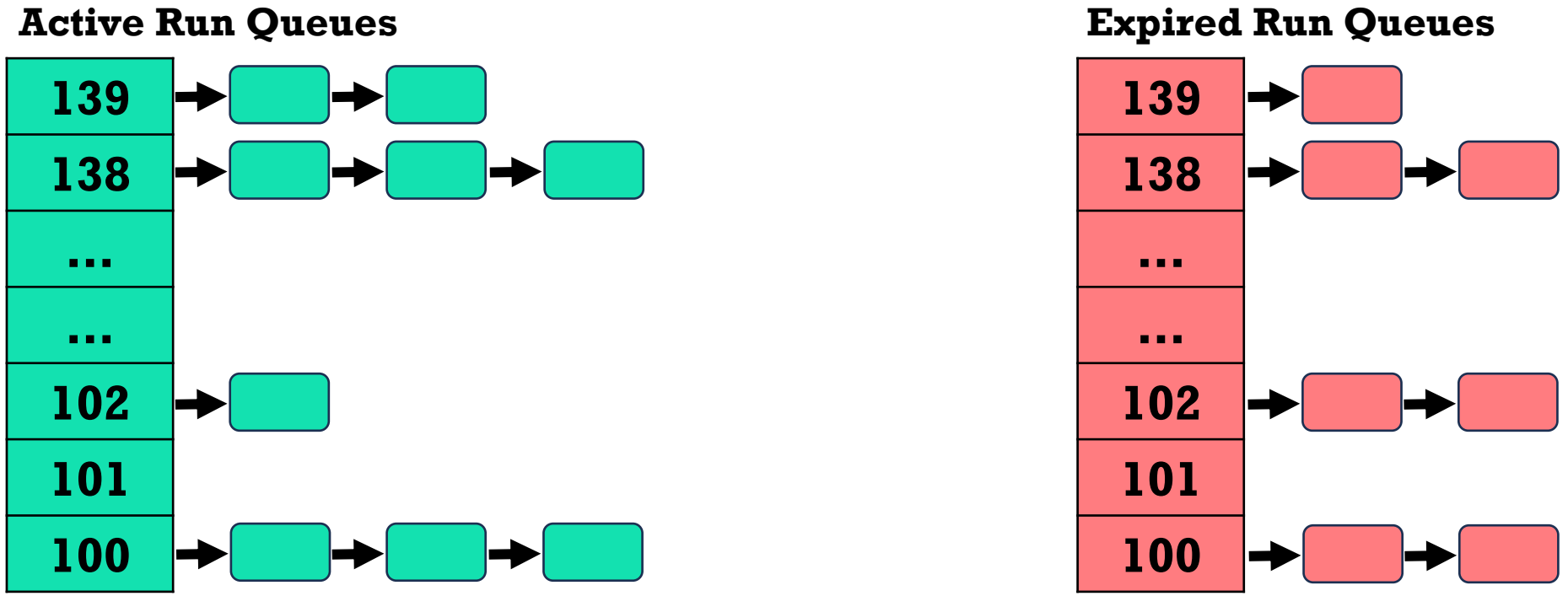
O(1) Scheduler

- Constant time is required to select the next process for execution
  - Scales to large number of processes

- Processes are divided into two groups
  - **Real time**
    - Priorities 0 - 99
  - **Normal processes**
    - Interactive (I/O bound)
    - Batch (CPU Bound)
    - Priorities from 100 to 139

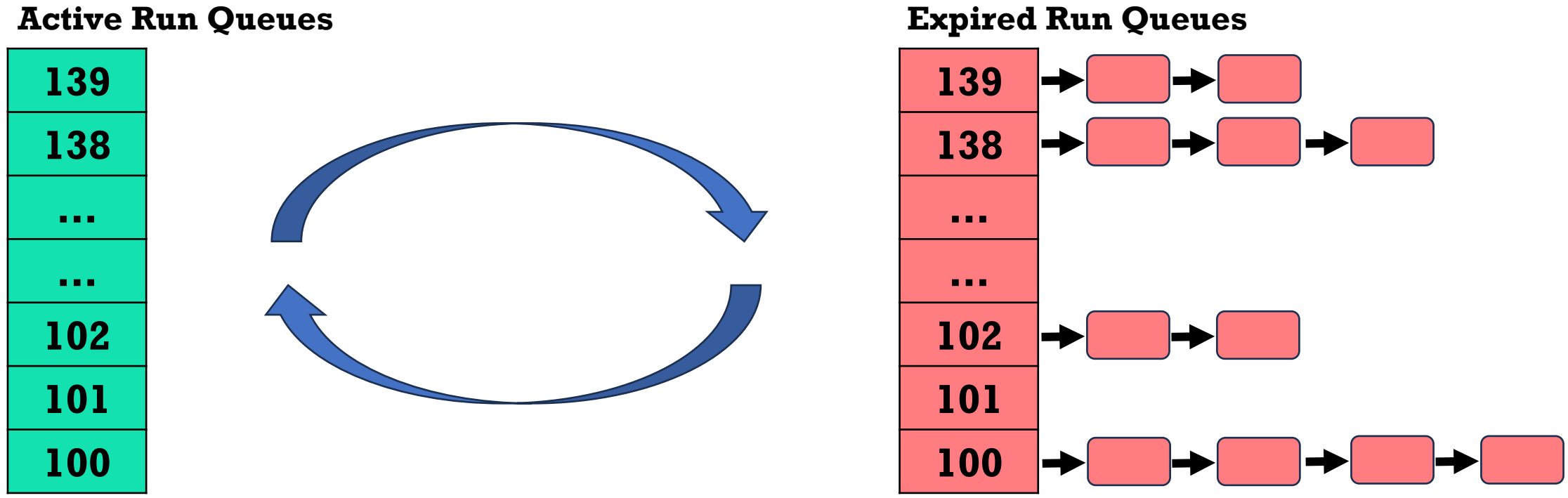
Scheduling Normal processes

- Two ready queues for each CPU
  - Each queue has 40 priority classes
  - 100 has highest priority, 139 has lowest priority



Scheduling Normal processes

- Two ready queues for each CPU
  - Each queue has 40 priority classes
  - 100 has highest priority, 139 has lowest priority



Constant Time???

- Two steps in scheduling
  1. Find the lowest numbered queue with at least one task
  2. Choose the first task from that queue
- Step 2 is obviously constant time
- Is step 1 constant time?

Constant Time???

- Two steps in scheduling
  1. Find the lowest numbered queue with at least one task
  2. Choose the first task from that queue

- Step 2 is obviously constant time
- Is step 1 constant time?

➢ **Store bitmap of run queues with non-zero entries**

➢ **Use special instruction "find-first-bit-set"**

-bsfl on Intel arch

Priorities

- 0 - 99 reserved for real time processes
- 100 is the highest priority for a normal task
- 139 is the lowest priority for a normal task
- Static priorities
  - 120 is the base priority
  - nice: command line to change the default priority of the task
  - $nice -n N ./my_prog.out
  - N is a value from +19 to -20
  - Most selfish “-20” (I want to run first)
  - Most generous “+19” (I will go last)

Dynamic Priorities

- Distinguish between batch and interactive processes
- Uses a “bonus”, which changes based on a heuristic

- DynamicPriority = MAX(100, MIN(StaticPriority - Bonus + 5, 139))

- Bonus has a value 0 - 10
  - If Bonus < 5 then more CPU bound process
  - If Bonus > 5 then more I/O bound process

Bonus Calculation

- To distinguish between batch and interactive processes
- Based on average sleep time
  - I/O bound processes will sleep more, so they should get a higher priority
  - CPU bound processes will sleep less, so should get lower priority
- Heuristic

Average Sleep Time	Bonus
0 ms - 100 ms	0
100 ms - 200 ms	1
200 ms - 300 ms	2
...	...

Time Slice Calculation

- Interactive processes have high priority
  - Likely to not complete their time slice
  - Give it the largest time slice

```
if (priority < 120)
    time_slice = (140 - priority) * 20 milliseconds
else
    time_slice = (140 - priority) * 5 milliseconds
```

O(1) Scheduler Summary

Multi-level feedback queues with 40 priority classes

Base priority is set to 120, but modifiable using nice command

Dynamic priority set by heuristics based on process' sleep time

Time slice interval for each process is set based on the dynamic priority

Limitations of O(1) Scheduler

- Too complex heuristics to distinguish between interactive and non-interactive processes
- Dependency between time slice and priority
- Priority and time slice values are not uniform

Completely Fair Scheduler (CFS)

- The Linux scheduler since v2.6.23
- Developed by Ingo Molnar
- No heuristics
- Elegant handling of I/O and CPU bound processes

Ideal Fair Scheduling

- Divide processor time equally among processes
- Ideal Fairness – If there are N processes in the system, each process should have got (100 / N) % of the CPU time

Process	Burst Time
P1	8
P2	4
P3	16
P4	4

P1	1	2	3	4	6	8			
P2	1	2	3	4					
P3	1	2	3	4	6	8	12	16	
P4	1	2	3	4					

Virtual Runtime

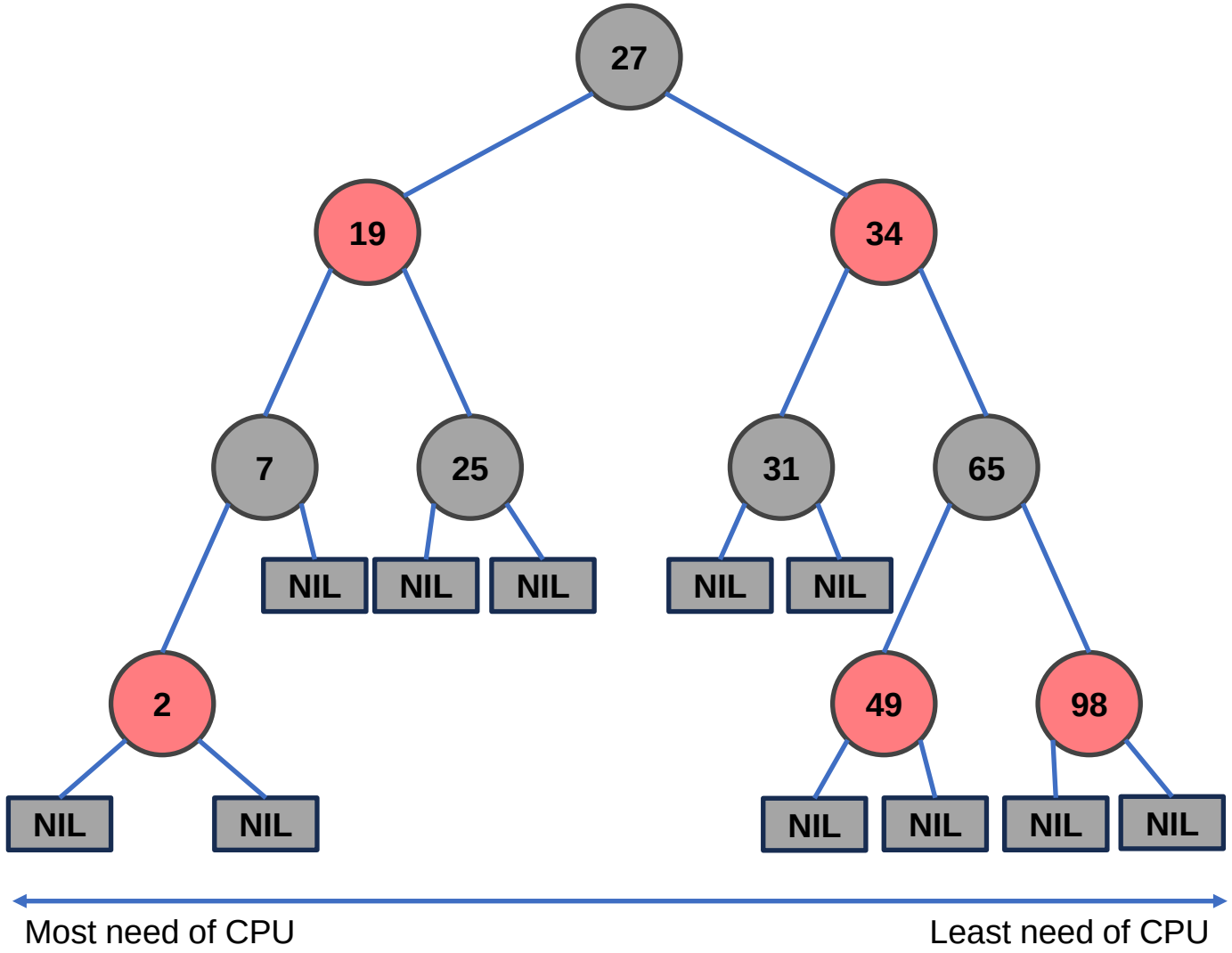
- Each runnable process has a virtual runtime (**vruntime**)
- At every scheduling step, if the process has run for **t ms**, then:
  - **vruntime += t**
- Vruntime for a process monolithically increases

The CFS approach

- When timer interrupt occurs:
  - Choose the task with the lowest vruntime (min_vruntime)
  - Compute its dynamic time slice
  - Program the high resolution timer with this time slice
- The process begins to execute by the CPU
- When interrupt occurs again:
  - Context switch if there is another task with a smaller runtime

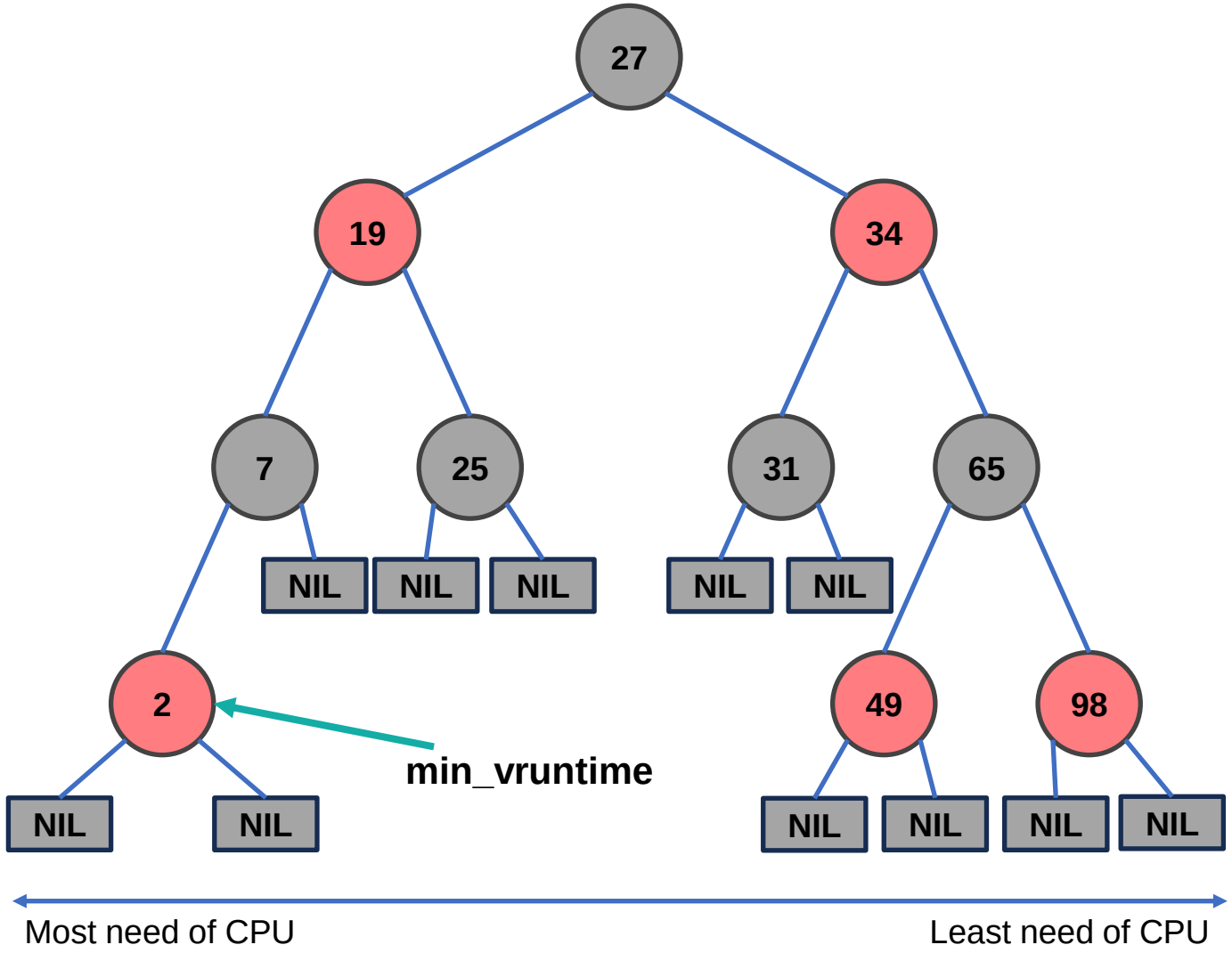
Picking the Next Task to Run

- CFS uses Red-Black tree
- Each node represents a runnable task
- Nodes are ordered according to their vruntime
- Nodes on the left have lower vruntime compared to the nodes on the right of the tree
- The left most node is the task with the least vruntime



Picking the Next Task to Run

- At context switch:
  - Pick the left most node of the tree
  - It is cached in min_vruntime. Therefore accessed in O(1)
  - If the previous process is runnable, it is inserted into the tree based on its new vruntime. Done in O(log(n))
  - Tasks move from left to right of the tree after its execution completes -> starvation is avoided



Why Red-Black tree?

- Self balancing
  - No path in the tree will be twice as long as any other path
- All operations are O(log(n))
  - Inserting / Deleting operations are quick and efficient

Priorities in CFS

- Priority (nice value) used to weigh the vruntime
- If the process has run for t ms, then
  - vruntime += t * (weight based on nice of the process)
- A lower priority implies time moves at a faster rate compared to that of a high priority task

I/O and CPU bound processes

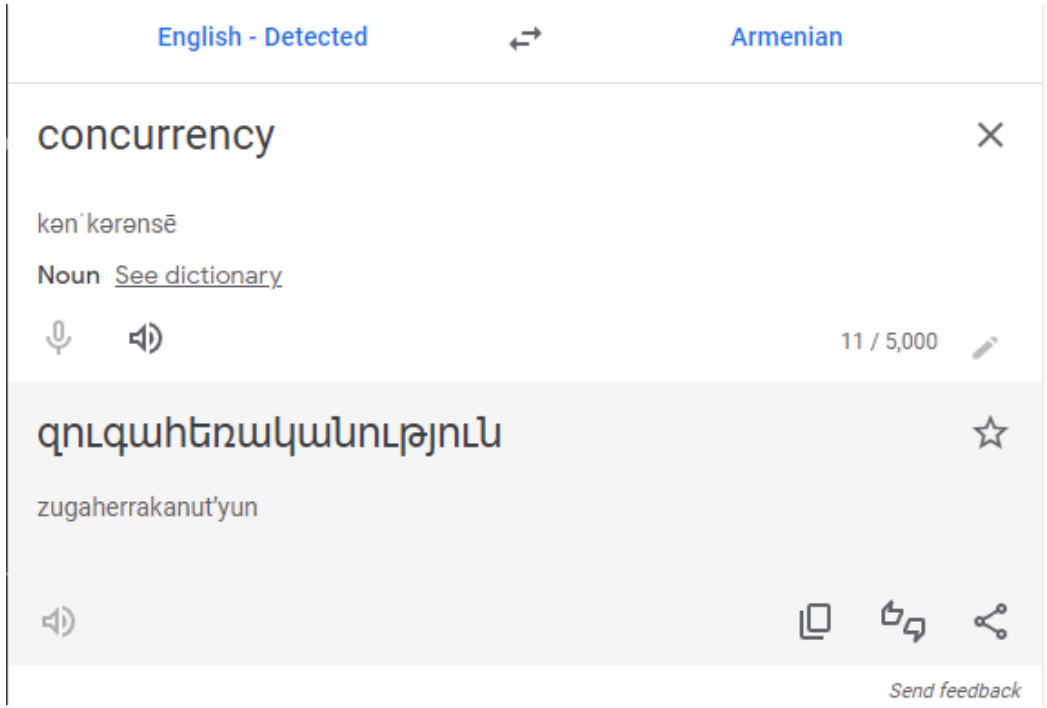
- I/O bound process should get a longer time to execute compared to CPU bound
- CFS achieves this efficiently
  - I/O bound processes have small CPU bursts and therefore will have a low vruntime. They would appear towards the left of the tree
  - I/O bound processes will typically havea larger time slices, because they have a smaller vruntime

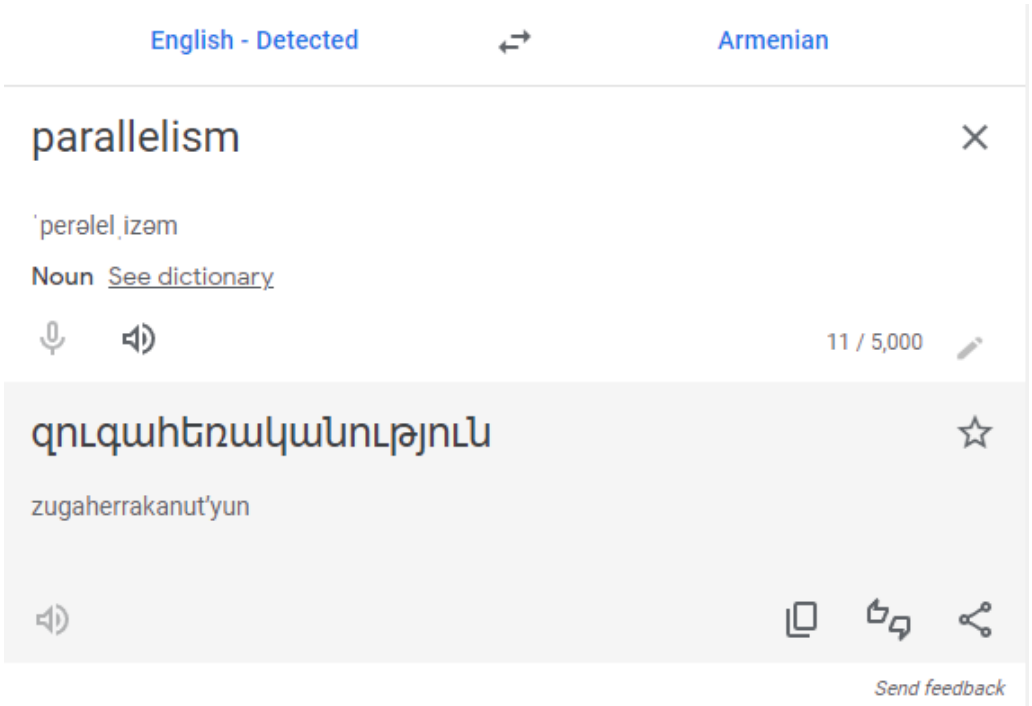
New Process

- Gets added to the RB-tree
- Starts with an initial value of min_vruntime
- This ensures that it gets to execute quickly

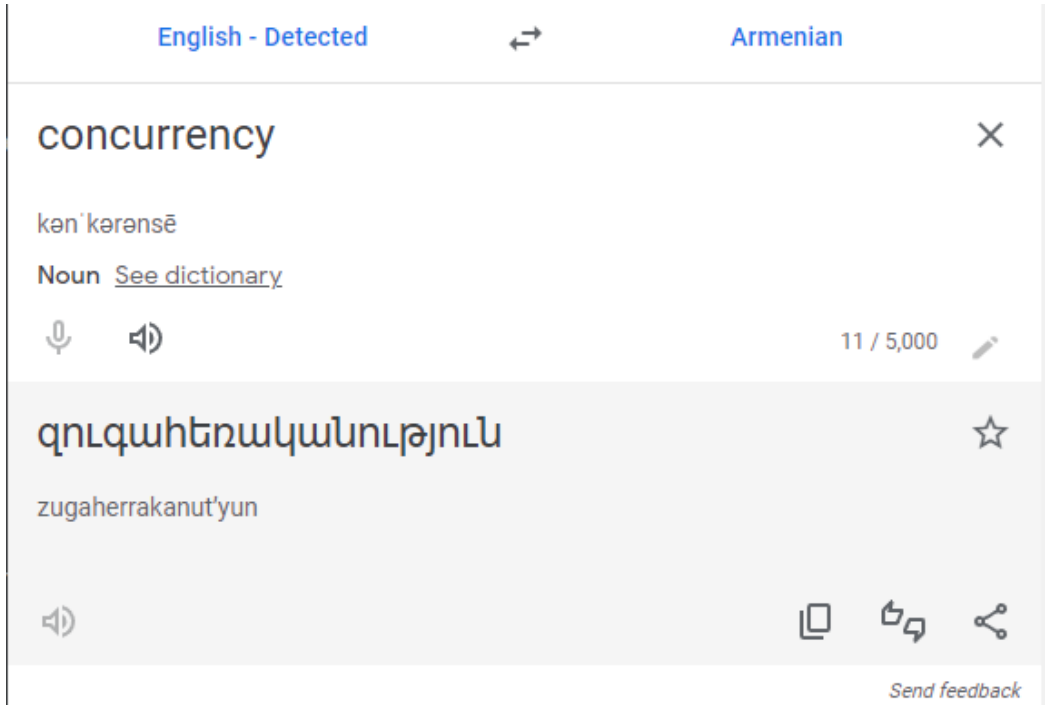
Multithreading and Concurrency

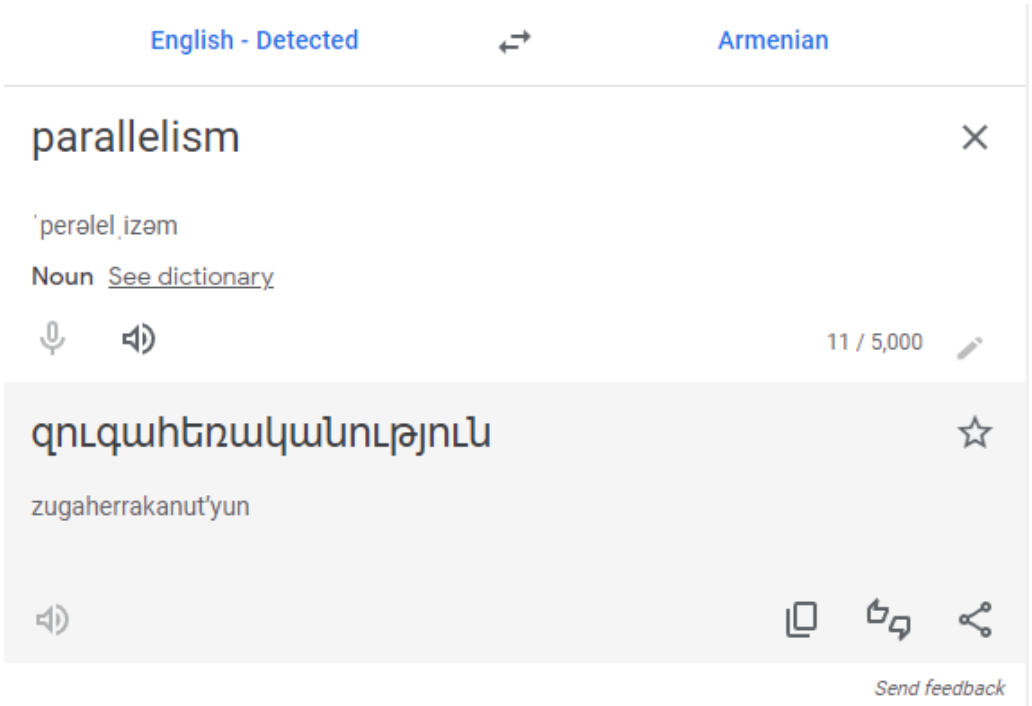
Concurrency...



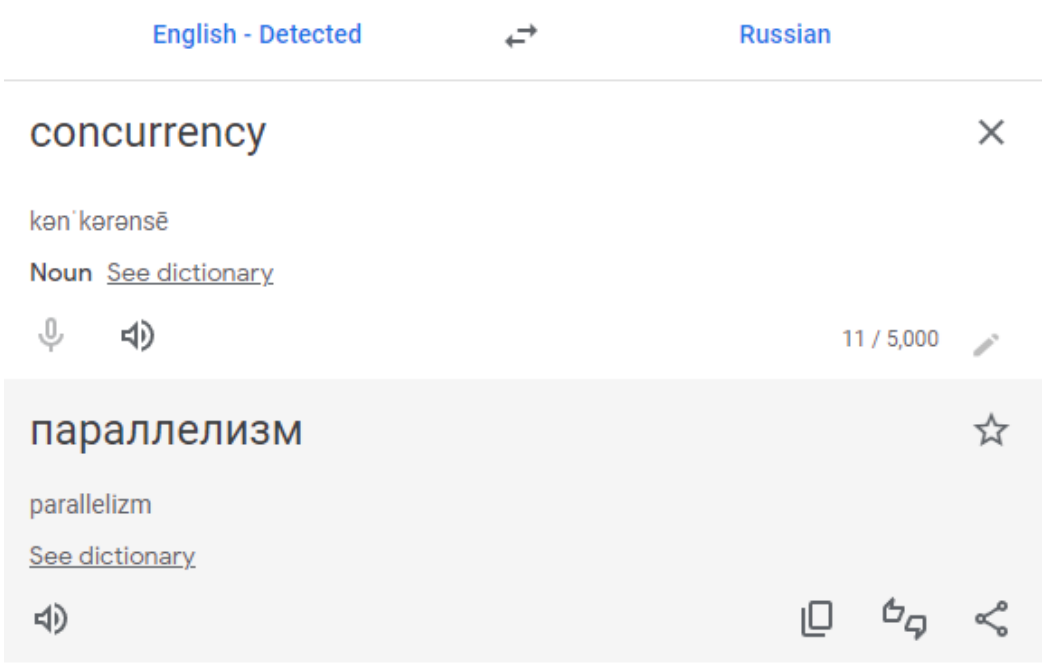


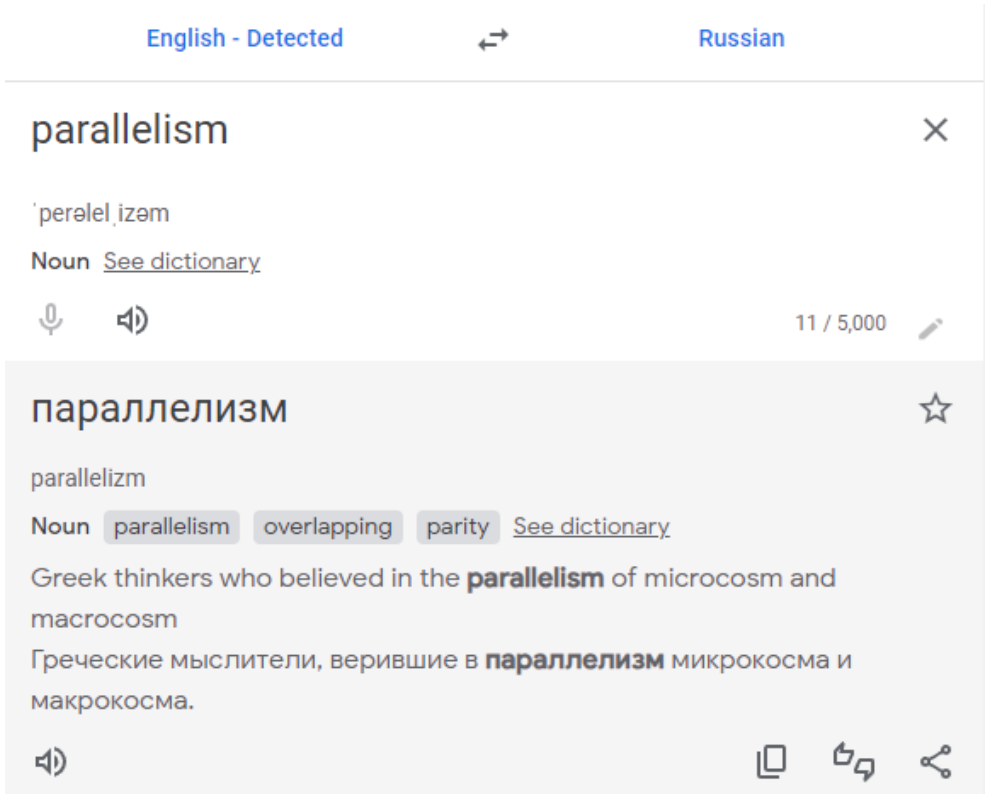
Concurrency...





Concurrency or Parallelism



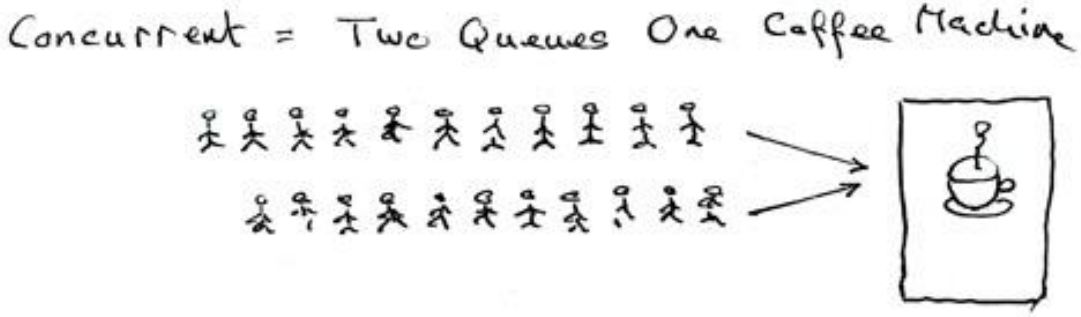


Why doesn't my program run N times faster when I run it on N-core computer

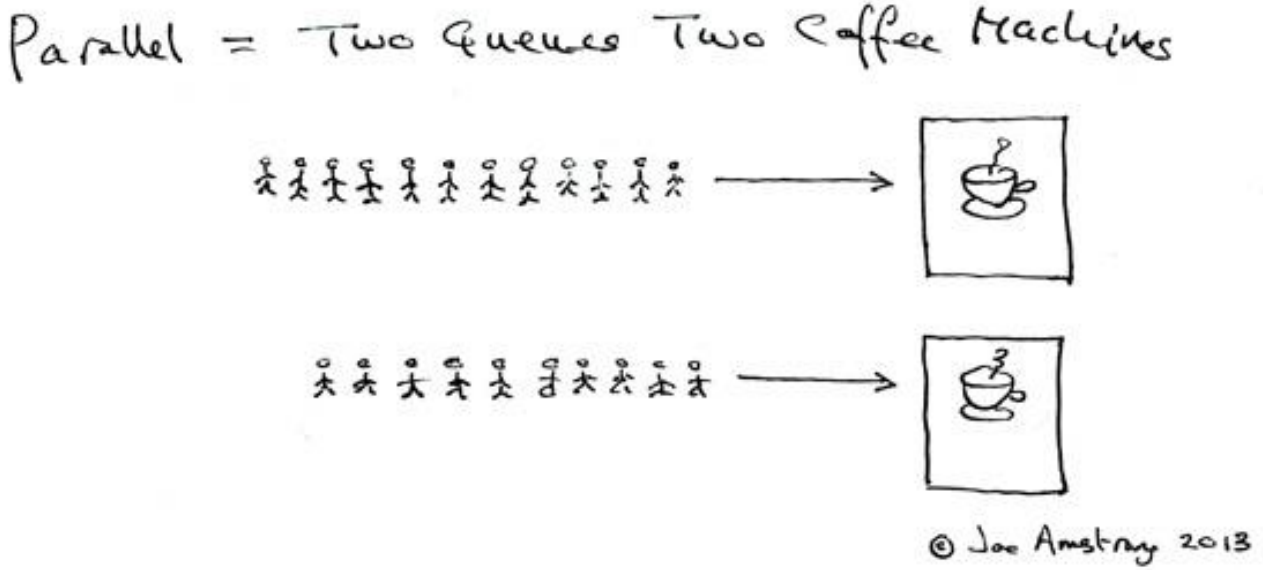
Because it's difficult to write parallel programs

Concurrency vs Parallelism

Concurrent
Two queues and one coffee machine.

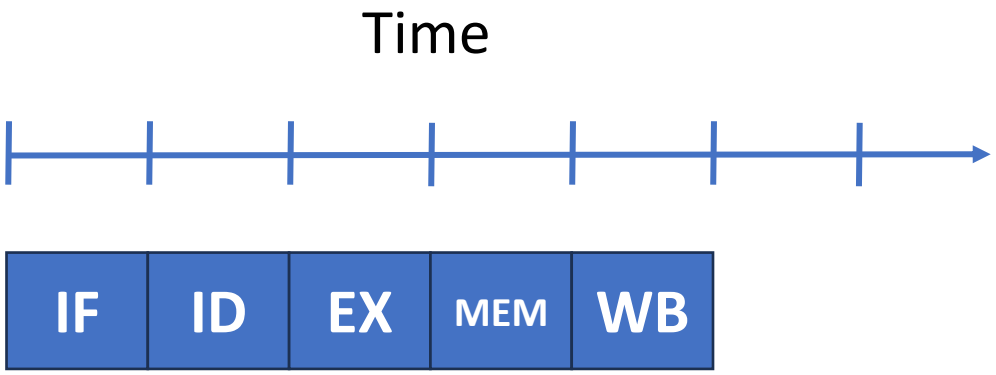


Parallel
Two queues and two coffee machines.



Back to concurrency

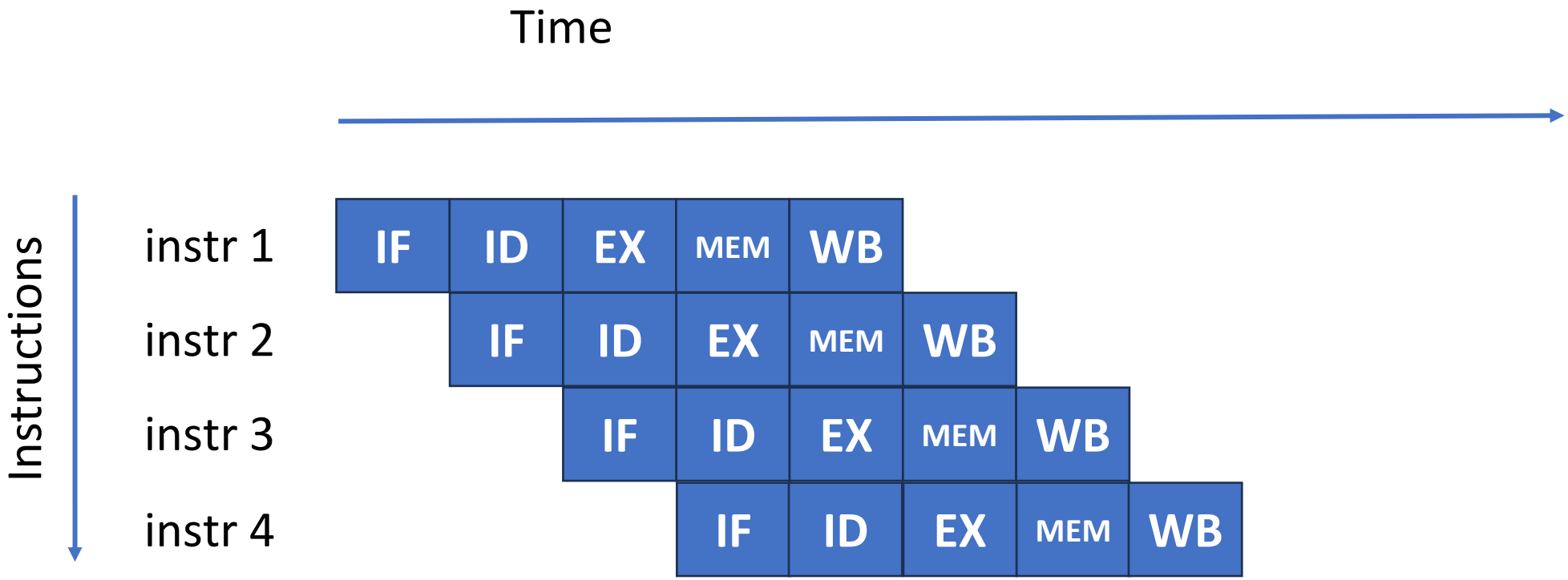
Computation stages



Classical 5-stage pipeline

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Execute (EX)
4. Memory access (MEM)
5. Write back (WB)

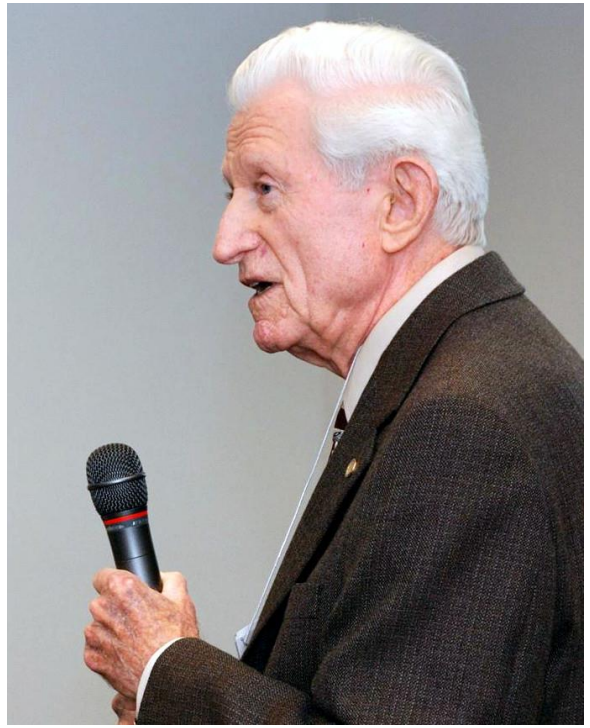
Computation pipeline



Amdahl's law: Idea

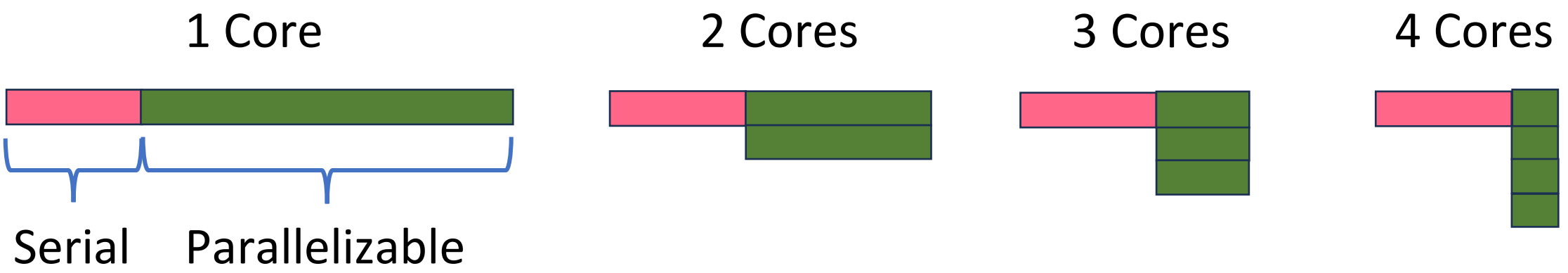
In 1967, Gene Amdahl pointed out that every algorithm consists of:

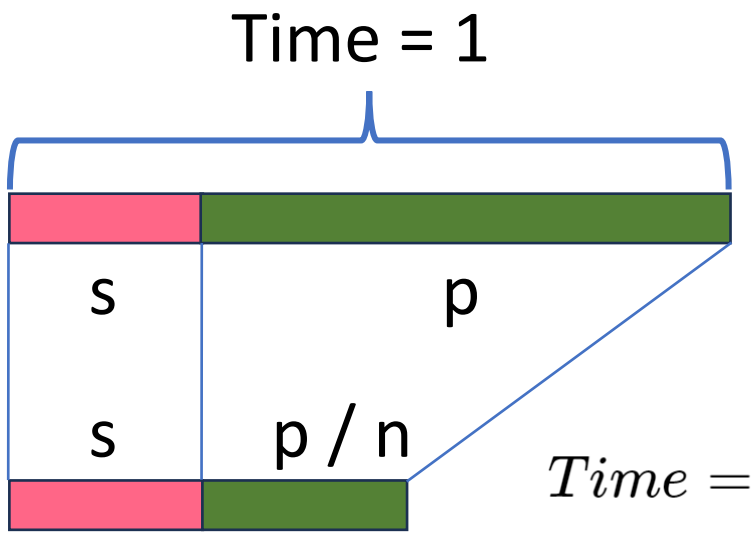
- part that can be done in parallel and
- part that cannot be, usually due to things such as data dependencies.



src: Wikipedia

Amdahl's law: Equation





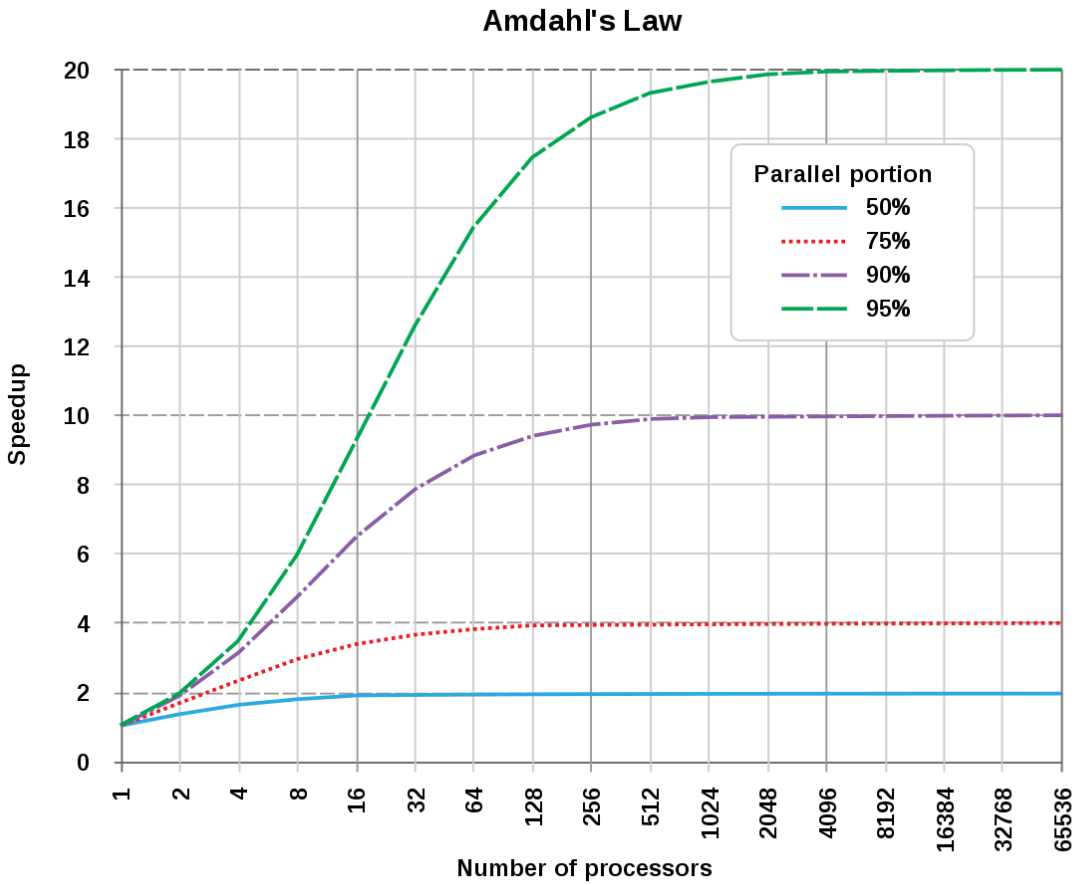
$$Time = s + \frac{p}{n}$$

$$Speedup = \frac{1}{s + \frac{p}{n}} = \frac{1}{1 - p + \frac{p}{n}}$$

Amdahl's law: Statistics

$$S = \frac{1}{0.05 + \frac{0.95}{65536}} = 20$$

Even if 95% of a program is parallelizable, you will never see a speed-up of more than 20 times.



Amdahl's law Conclusion

There’s a limited amount of parallelism in a computer program

Once it has been exploited there’s no additional benefit in adding more parallelism: a program can never run more quickly than its sequential part.

The Linux implementation of Threads

- Threads enable *concurrent programming* and, on multiple processor systems, true *parallelism*.
- They provide multiple threads of execution within the same program in a shared memory address space.
- They can also share open files and other resources.
- To the Linux kernel, there is no concept of a thread.
- Linux implements all threads as standard processes.

The Linux implementation of Threads

- In Linux, the implementation of Pthread standard is provided by *glibc*, Linux's C library.
- The Pthread API is defined in <pthread.h>.
- Every function in the API is prefixed by pthread_.
- Pthread functions may be broken into two large groupings:

  Thread management

  - Functions to create, destroy, join, and detach threads.

  Synchronization

  - Functions to manage the synchronization of threads, including mutexes, condition variables, and barriers.

Linking pthreads

- Although Pthreads is provided by *glibc*, it is in a separate library, *libpthread*, and thus requires explicit linkage.
- With *gcc*, this is automated by the *-pthread* flag, which ensures the proper library is linked into your executable:

➢ gcc -Wall -Werror **-pthread** beard.c -o beard

Creating Threads

- When your program is first run and executes the main() function, it is single threaded.
- From this initial thread, sometimes called the *default* or *master thread*, you must create one or more additional threads to become multithreaded.
- Pthreads provides a single function to define and launch a new thread, pthread_create():

```
#include <pthread.h>
int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine) (void *),
                    void *arg);
```

Creating Threads

```
int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(* start_thread ) (void *),
                    void *arg);
```

- Upon successful invocation, a new thread is created and begins executing the function provided by start_routine, passed the sole argument arg.

- The start_routine must have the following signature:

```
void * start_thread (void *arg);
```

- Thus, the thread begins life by executing a function that accepts a void pointer as its sole argument and returns a void pointer as its return value.

Thread IDs

- The *thread ID* (*TID*) is the thread analogue to the process ID (PID).
- While the PID is assigned by the Linux kernel, the TID is just assigned in the Pthread library
- A thread can obtain its TID at runtime via the pthread_self() function:

```
#include <pthread.h>
pthread_t pthread_self (void);
```

Terminating Threads

- The counterpart to thread creation is thread termination.
- Thread termination is similar to process termination, except that when a thread terminates, the rest of the threads in the process continue executing.

Threads may terminate under several circumstances

- If a thread returns from its start routine, it terminates
- If a thread invokes the pthread_exit()
- If the thread is canceled by another thread via the pthread_cancel() function

Terminating Threads

- The easiest way for a thread to terminate itself is to “fall off the end” of its start routine.
- Often you want to terminate a thread deep in a function call stack, far from your start routine.
- For those cases, Pthreads provides pthread_exit(), the thread equivalent of exit():

```
#include <pthread.h>
void pthread_exit (void *retval);
```

- retval is provided to any thread waiting on the terminating thread’s death

Joining threads

- *Joining* allows one thread to block while waiting for the termination of another:

```
#include <pthread.h>
int pthread_join (pthread_t thread, void **retval);
```

- Upon successful invocation, the invoking thread is blocked until the thread specified by thread terminates
- Once thread terminates, the invoking thread is woken up
- If retval is not NULL, provided the return value the terminated thread passed to pthread_exit() or returned from its start routine.
- All threads in Pthreads are peers; any thread may join any other.

Joining threads

```
int ret;
/* join with `thread' and we don't care about its return value */
ret = pthread_join (thread, NULL);
if (ret) {
        errno = ret;
        perror ("pthread_join");
        return -1;
}
```

Critical Section & Synchronization

Critical Section

- *n* processes competing to use some shared data.
- No assumptions may be made about speeds or the number of CPUs.
- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.

- **The Problem** – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.

General structure

do {

enter section

critical section

leave section

remainder section

} **while (TRUE);**

- Processes may share some common variables to synchronize their actions.

Example

X = 10

Thread 1

```
do {
   X++
} while (true)
```

Thread 2

```
do {
   X--
} while (true)
```

Print(X)

Possible values of X

X++	X--
Load(X)	Load(X)
Inc(X)	Dec(X)
Store(X)	Store(X)

Load(X)	Load(X)	Load(X)
Inc(X)	Load(X)	Load(X)
Store(X)	Dec(X)	Inc(X)
Load(X)	Store(X)	Store(X)
Dec(X)	Inc(X)	Dec(X)
Store(X)	Store(X)	Store(X)
X = 10	X=11	X=9

CS Solution Requirements

- There are 3 requirements that must stand for a correct solution:
  1. **Mutual Exclusion**
  2. **Progress**
  3. **Bounded Waiting**

- We can check on all three requirements in each proposed solution, even though the non-existence of each one of them is enough for an incorrect solution.

CS Solution - Mutual Exclusion

- **Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- Implications:
  - Critical sections better be focused and short.
  - Better not get into an infinite loop in there.
  - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

CS Solution - Progress

- **Progress –** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely:
  - If only one process wants to enter, it should be able to.
  - If two or more want to enter, one of them should succeed.

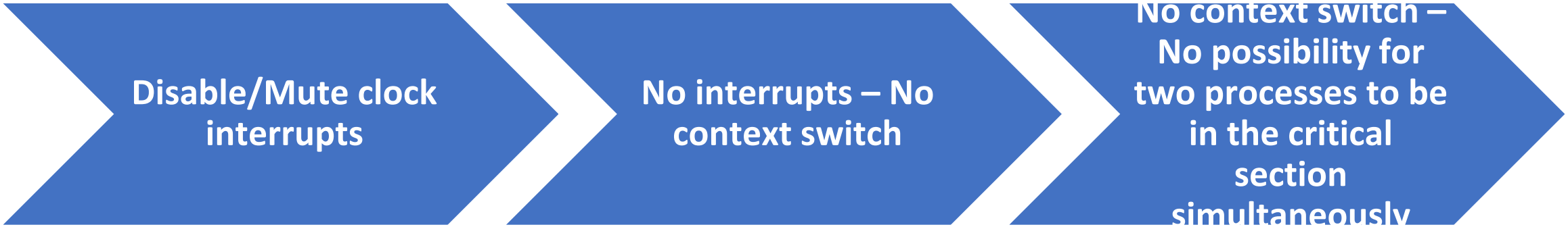
CS Solution – Bounded Waiting

- **Bounded Waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
  - Assume that each process executes at a nonzero speed.
  - No assumption concerning relative speed of the *n* processes.

CS Solution Approaches

- Hardware solutions
  - rely on some special machine instructions.
- Software solutions
  - algorithms who's correctness does not rely on any other assumptions.
- Operating System solutions
  - provide some functions and data structures to the programmer through system/library calls.
- Programming Language solutions
  - Linguistic constructs provided as part of a language.

CS Solution: Hardware solutions



Synchronization approach: Using Interrupts

Thread 1

```
while(true) {
        …. AAAAA
        …..
        disable_interrupts() //lock
        critical section;
        enable_interrupts() //unlock
}
```

Thread 2

```
while(true) {
        …. BBBBB
        …..
        disable_interrupts() //lock
        critical section;
        enable_interrupts() //unlock
}
```

- Simple
- Requires privileges
- Not suited for multicore systems

Synchronization approach #1

turn = 1

Thread 1

```
while(true) {
        …. AAAAA
        …..
        while(turn == 2); //lock
        critical section;
        turn = 2; //unlock
}
```

Thread 2

```
while(true) {
        …. BBBBB
        …..
        while(turn == 1); //lock
        critical section;
        turn = 1; //unlock
}
```

Synchronization approach #1

turn = 1

Thread 1

```
while(true) {
        …. AAAAA
        …..
        while(turn == 2); //lock
        critical section;
        turn = 2; //unlock
}
```

Thread 2

```
while(true) {
        …. BBBBB
        …..
        while(turn == 1); //lock
        critical section;
        turn = 1; //unlock
}
```

- Achieves mutual exclusion
- Busy waiting – waste of power
- If T2 arrives first it can **NOT progress**

Synchronization approach #2

T1_INSIDE = FALSE

T2_INSIDE = FALSE

Thread 1

```
while(true) {
    …. AAAAA
    …..
    while(T2_inside == true); //lock
    T1_inside = true;
    critical section;
    T1_inside = false; //unlock
}
```

Thread 2

```
while(true) {
    …. BBBBB
    ….
    while(T1_inside == true); //lock
    T2_inside = true;
    critical section;
    T2_inside = false; //unlock
}
```

Synchronization approach #2

```
while(true) {
   while(T2_inside == true); //lock
   T1_inside = true;
   critical section;
   T1_inside = false; //unlock
}
```

T1_INSIDE = FALSE

T2_INSIDE = FALSE

```
while(true) {
   while(T1_inside == true); //lock
   T2_inside = true;
   critical section;
   T2_inside = false; //unlock
}
```

CPU	T1_INSIDE	T2_INSIDE
while(T2_inside == true);	False	False
Context switch		
while(T1_inside == true);	False	False
T2_inside = true;	False	True
Context switch		
T1_inside = true;	True	True

Synchronization approach #3

T1_Wants_To_Enter = FALSE

T2_Wants_To_Enter = FALSE

Thread 1

```
while(true) {
    …. AAAAA
    …..
    T1_wants_to_enter = true;
    while(T2_wants_to_enter == true); //lock
    critical section;
    T1_wants_to_enter = false;//unlock
}
```

Thread 2

```
while(true) {
…. BBBBB
….
    T2_Wants_To_Enter = true;
    while(T1_Wants_To_Enter == true); //lock
    critical section;
    T2_Wants_To_Enter = false; //unlock
}
```

Synchronization approach #3: Deadlock

T1_Wants_To_Enter = FALSE

T2_Wants_To_Enter = FALSE

Thread 2

```
while(true) {
   T1_wants_to_enter = true;
   while(T2_wants_to_enter = true); //lock
   critical section;
   T1_wants_to_enter = false;//unlock
}
```

```
while(true) {
   T2_Wants_To_Enter = true;
   while(T1_Wants_To_Enter == true); //lock
   critical section;
   T2_Wants_To_Enter = false; //unlock
}
```

CPU	T1_Wants_To_Enter	T2_Wants_To_Enter
T1_Wants_To_Enter = true;	False	False
Context switch		
T2_Wants_To_Enter = true;	False	False

Synchronization approach #4
Peterson's Algorithm

T1_Wants_To_Enter = FALSE

T2_Wants_To_Enter = FALSE

Favored

Thread 1

```
while(true) {
    ....
    T1_Wants_To_Enter = true;
    Favored = 2;
    while(T2_Wants_To_Enter = true && Favored == 2); //lock
    critical section;
    T1_Wants_To_Enter = false;//unlock
}
```

Thread 2

```
while(true) {
    ....
    T2_Wants_To_Enter = true;
    Favored = 1;
    while(T1_Wants_To_Enter == true && Favored == 1); //lock
    critical section;
    T2_Wants_To_Enter = false; //unlock
}
```

Bakery Algorithm

- Synchronization for N processes
- Developed by Leslie Lamport

Bakery Algorithm

Critical Section for n processes:

- Before entering its critical section, a process receives a number (like in a bakery). The holder of the smallest number enters the critical section.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first (PID assumed unique).

Bakery Algorithm

- Choosing a number:
  - max (a_0,…, a_{n-1}) is a number k, such that $k \geq a_i$ for $i = 0, \ldots, n-1$
- Notation for lexicographical order (ticket #, PID #)
  - $(a,b) < (c,d)$ if $a < c$ or if $a == c$ and $b < d$

Bakery Algorithm: Initial approach

```
do {
   number[me] = max(number[0], …, number[n – 1]) + 1;
   for (j = 0; j < n; j++) {
      while ((number[j] != 0) && (number[j] < number[me])) ;
   }

   critical section

   number[me] = 0; // unlock
   remainder section
} while (TRUE);
```

Bakery Algorithm: Original Version

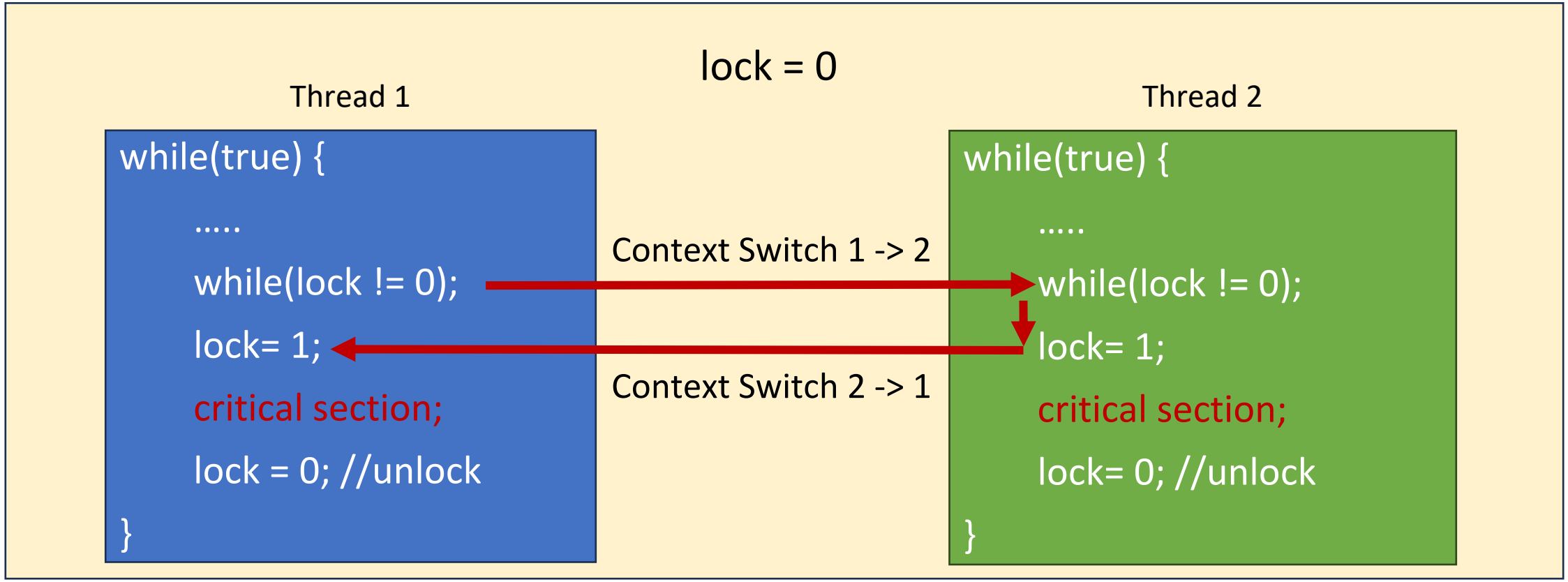
```
do {
   choosing[me] = TRUE;
   number[me] = max(number[0], …, number[n – 1]) +1;
   choosing[me] = FALSE;
   for (j = 0; j < n; j++) {
      while (choosing[j]) ;
      while ((number[j] != 0) && ((number[j],j) < (number[me],me))) ;
   }
   critical section
   number[me] = 0;
   remainder section
} while (TRUE);
```

- Shared data:

boolean choosing[n];
int number[n];

Data structures are initialized to **FALSE** and **0**, respectively.

Mutual Exclusion?



Do we have a mutual exclusion here? **No, we don't!**

Mutual Exclusion: Atomic block

```
while(true) {
        .....
        while(lock != 0);
        lock= 1;
        critical section;
        turn = 0; //unlock
}
```

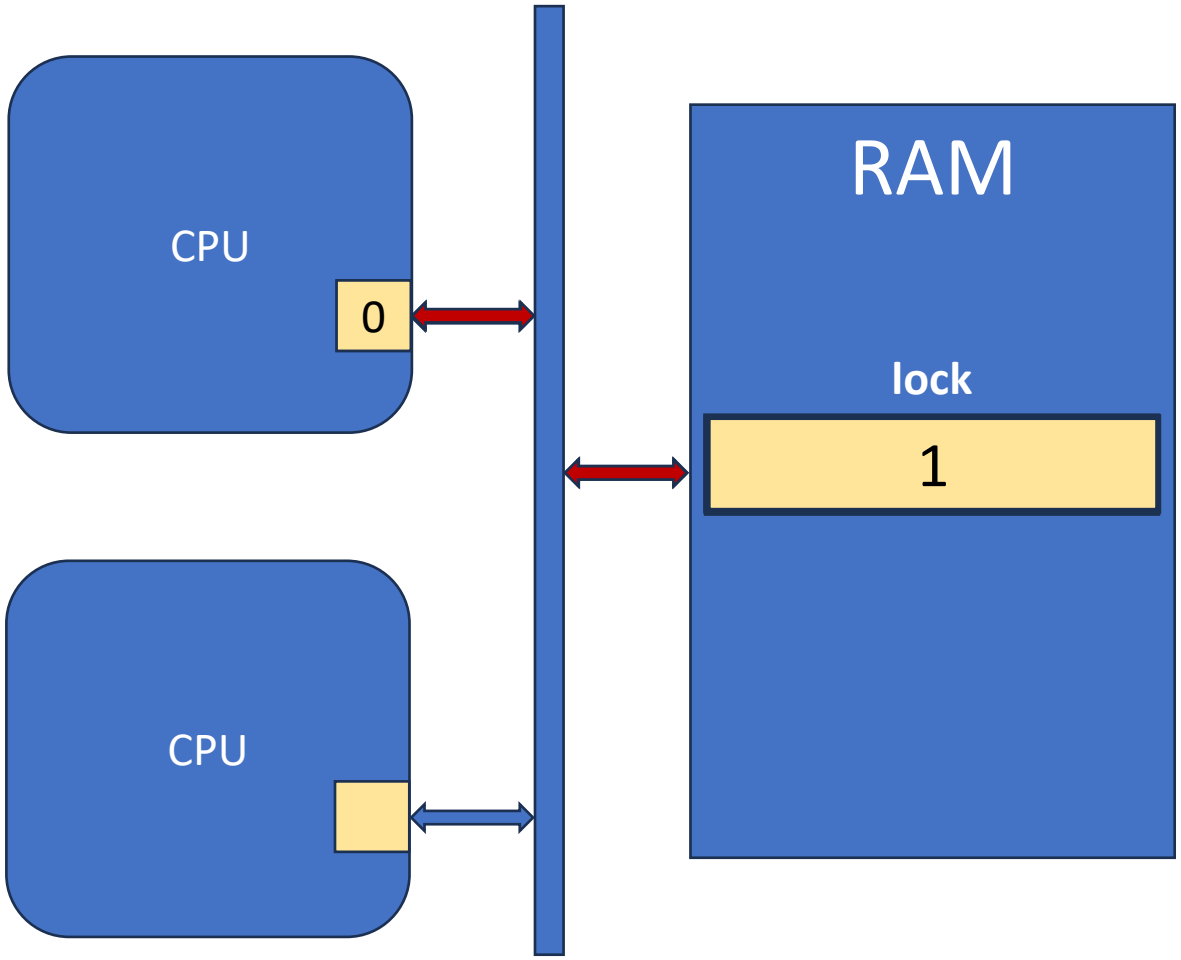
Need to execute these instructions atomically

Any ideas how to do that?

Mutual Exclusion: Hardware Support

- Test and Set instruction

```
int test_and_set(int *location) {
    bus_lock();
    int previous = *location;
    *location = 1;
    return previous;
    bus_unlock()
}
```



Locking using test_and_set

lock = 0

Thread 1

```
while(true) {
        .....
        while(test_and_set(&lock) == 1);
        critical section;
        lock = 0; //unlock
}
```

Thread 2

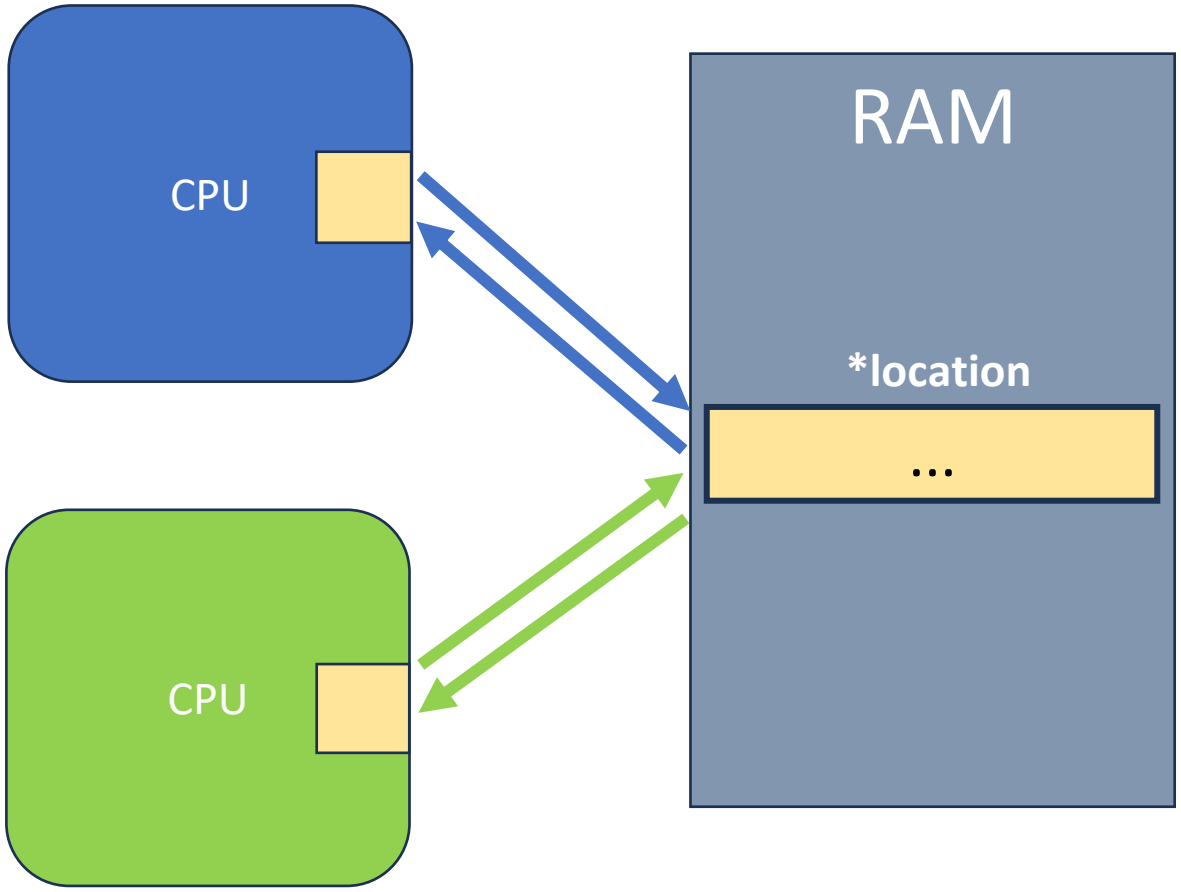
```
while(true) {
        .....
        while(test_and_set(&lock) == 1);
        critical section;
        lock= 0; //unlock
}
```

Do we have a mutual exclusion here? **Yes, we do!**

Intel Hardware Support: xchg

- xchg: write to memory and return it's old value atomically

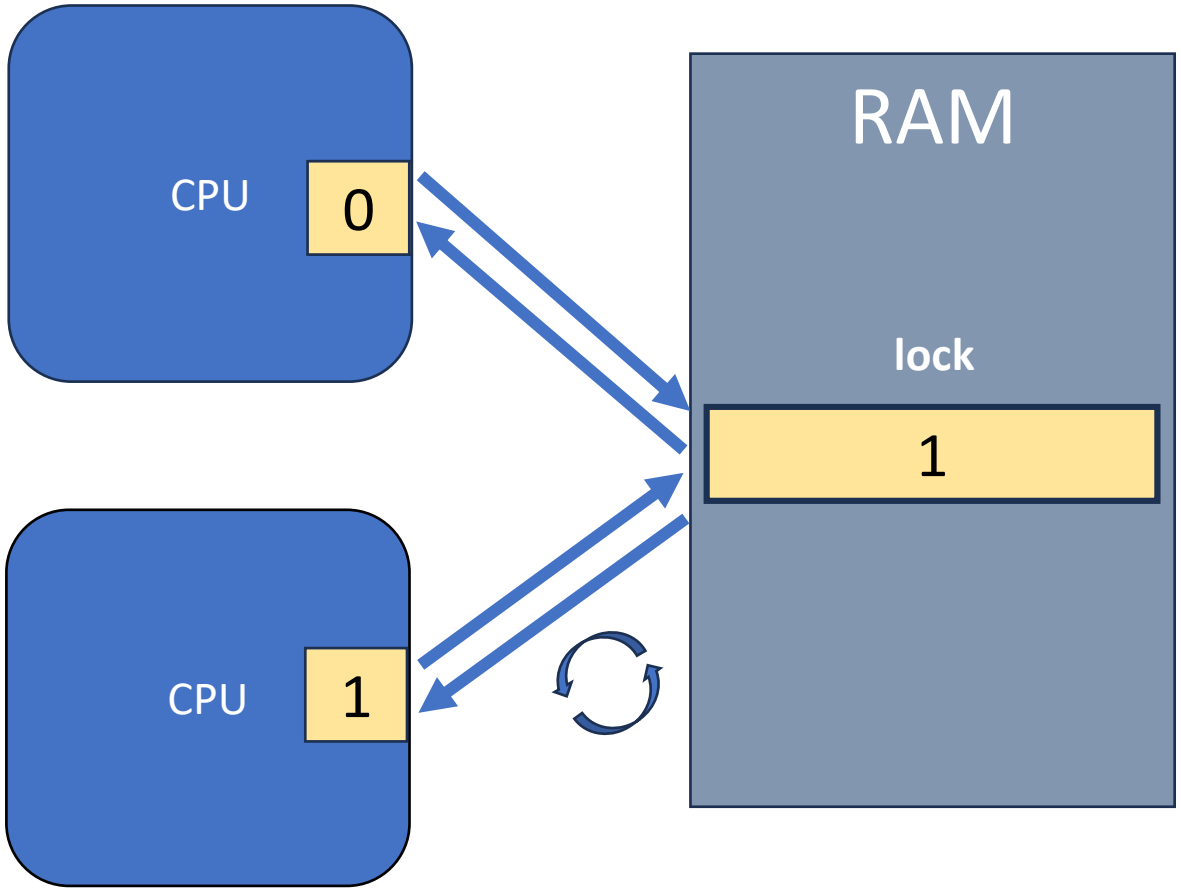
```
int xchg(int *location, int value) {
    bus_lock();
    int previous= *location;
    *location = value;
    return previous;
    bus_unlock()
}
```



Acquiring and Releasing the lock using xchg

```
void acquire(int *lock) {
    while(true) {
        if(xchg(lock, 1) == 0)
            break;
    }
}
```

```
void release(int *lock) {
    locked = 0;
}
```



Spinlock

lock = 0

Thread 1

```
while(true) {
        .....
        acquire(&lock); //lock
        critical section;
        release(&lock); //unlock
}
```

Thread 2

```
while(true) {
        .....
        acquire(&lock); //lock
        critical section;
        release(&lock); //unlock
}
```

Do we have a mutual exclusion here? **Yes, we do!**

Spinlock step by step

lock = 0

Thread 1

```
acquire(&lock); //lock
critical section;
release(&lock); //unlock
```

Thread 2

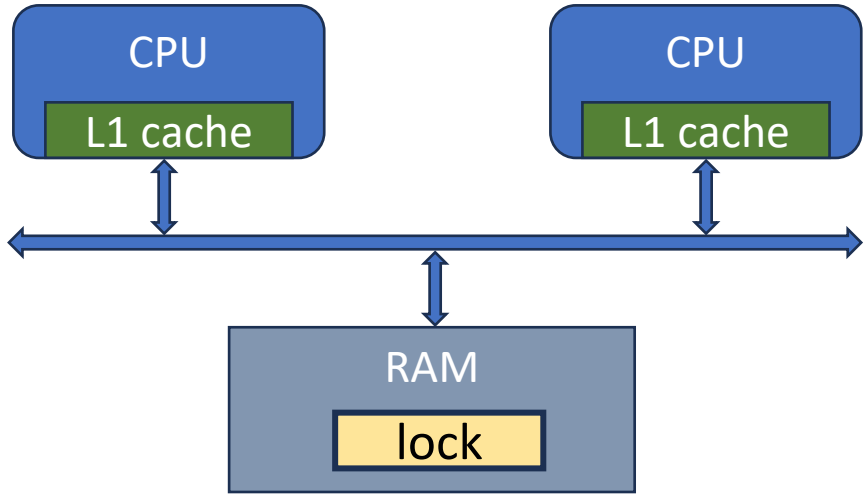
```
acquire(&lock); //lock
critical section;
release(&lock); //unlock
```

```
void acquire(int *lock) {
    while(true) {
        if(xchg(lock, 1) == 0)
            break;
    }
}
```

```
void release(int *lock) {
    locked = 0;
}
```

Spinlock Issues

- No compiler optimizations are allowed
  - lock variable should not be a register variable
- No operation reordering
  - Memory loads and stores should be serial operations
- No caching for the *lock* variable



Spinlock use cases

- **Useful** for short critical sections: Busy waiting
  - Change variable value
  - Access/modify array element

- **Not useful** when period is unpredictable or very long
  - Loading big chunks of data from the disk
  - Network data reading/writing

Mutex – Sleeping Spinlock

- If critical section is locked, then yield the CPU
  - Go to sleep state

```
void lock(int *lock) {
    while(true) {
        if(xchg(lock, 1) == 0)
            break;
        else
            sleep();
    }
}
```

- When unlocking wake up sleeping processes

```
void unlock(int *lock) {
    locked = 0;
    wakeup();
}
```

Thundering Herd Problem

Many processes wake up when the event occurs

- All waiting processes wake up
- Several context switches
- All processes go to sleep except one, which gets into the critical section
  - Large number of context switches again
  - Could lead to starvation

Thundering Herd Problem: The Solution

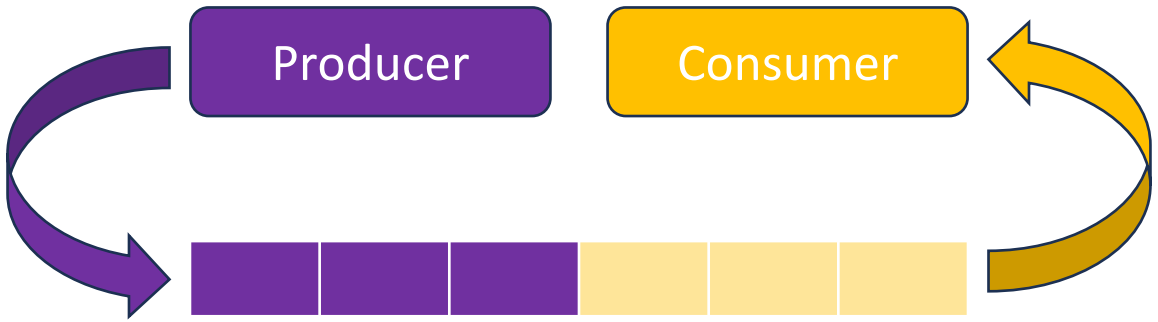
```
void lock(int *lock) {
    while(true) {
        if(xchg(lock, 1) == 0)
            break;
        Else
            Add thread/process to the queue
            sleep();
    }
}
```

```
void unlock(int *lock) {
    locked = 0;
    Remove the thread/process from the queue
    wakeup(P);
}
```

- When entering critical section push the thread/process into a queue before sleeping
- When unlocking wake up only the first process from the head of the queue

Producer - Consumer Problem

- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “ate”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data



Producer - Consumer Structure

Producer

```
while(true) {
    item = produce_item();
    insert_item(item); //into buffer
    count ++;
}
```

Consumer

```
while(true) {
    item = remove_item(); //from buffer
    count --;
    consume_item(item);
}
```

Producer - Consumer Sync (1)

Producer

```
while(true) {
    item = produce_item();
    lock(mutex);
    insert_item(item); //into buffer
    count ++;
    unlock(mutex);
}
```

Consumer

```
while(true) {
    lock(mutex);
    item = remove_item(); //from buffer
    count --;
    unlock(mutex);
    consume_item(item);
}
```

Producer - Consumer Sync (2)

Producer

```
while(true) {
    item = produce_item();
    if (count == N) sleep(m_not_full);
    lock(m_elem);
    insert_item(item); //into buffer
    count ++;
    unlock(m_elem);
    if (count == 1) wake(m_not_empty);
}
```

Consumer

```
while(true) {
    if (count == 0) sleep(m_not_empty);
    lock(m_elem);
    item = remove_item(); //from buffer
    count --;
    unlock(m_elem);
    if (count == N - 1) wake(m_not_full);
    consume_item(item);
}
```

Semaphore

- Proposed by Dijkstra in 1965
- Functions down and up are atomic
- Variants
  - Blocking
  - Non blocking

```
void down(int *S) {
    while (*S <= 0);
    *S--;
}

void up(int *S) {
    *S++;
}
```

Producer - Consumer Sync (2)

Producer

```
while(true) {
    item = produce_item();
    down(empty_slots);
    lock(mutex);
    insert_item(item); //into buffer
    count ++;
    unlock(mutex);
    up(full_slots);
}
```

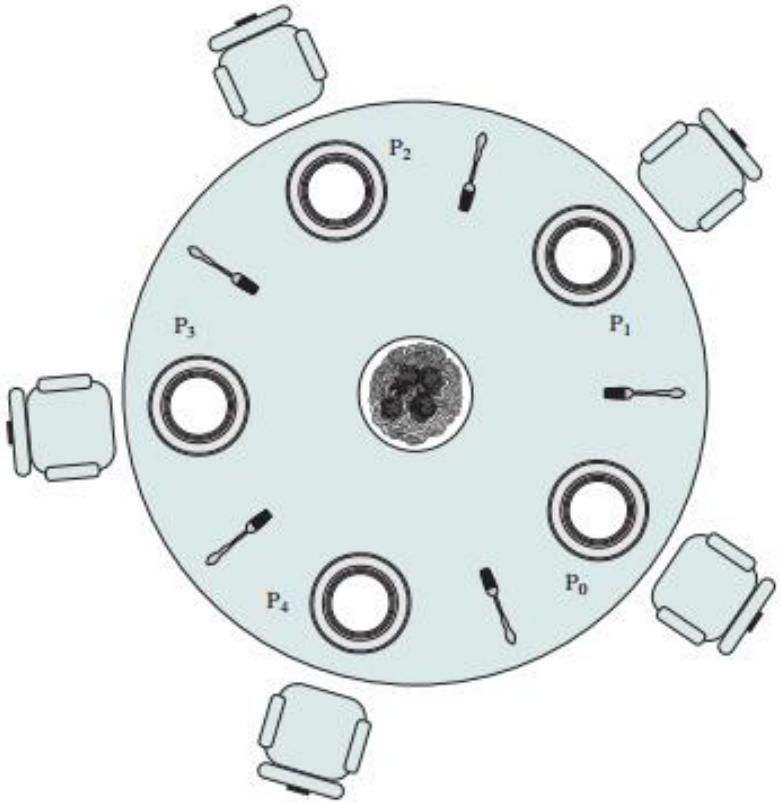
Consumer

```
while(true) {
    down(full_slots);
    lock(mutex);
    item = remove_item(); //from buffer
    count --;
    unlock(mutex)
    up(empty_slots);
    consume_item(item);
}
```

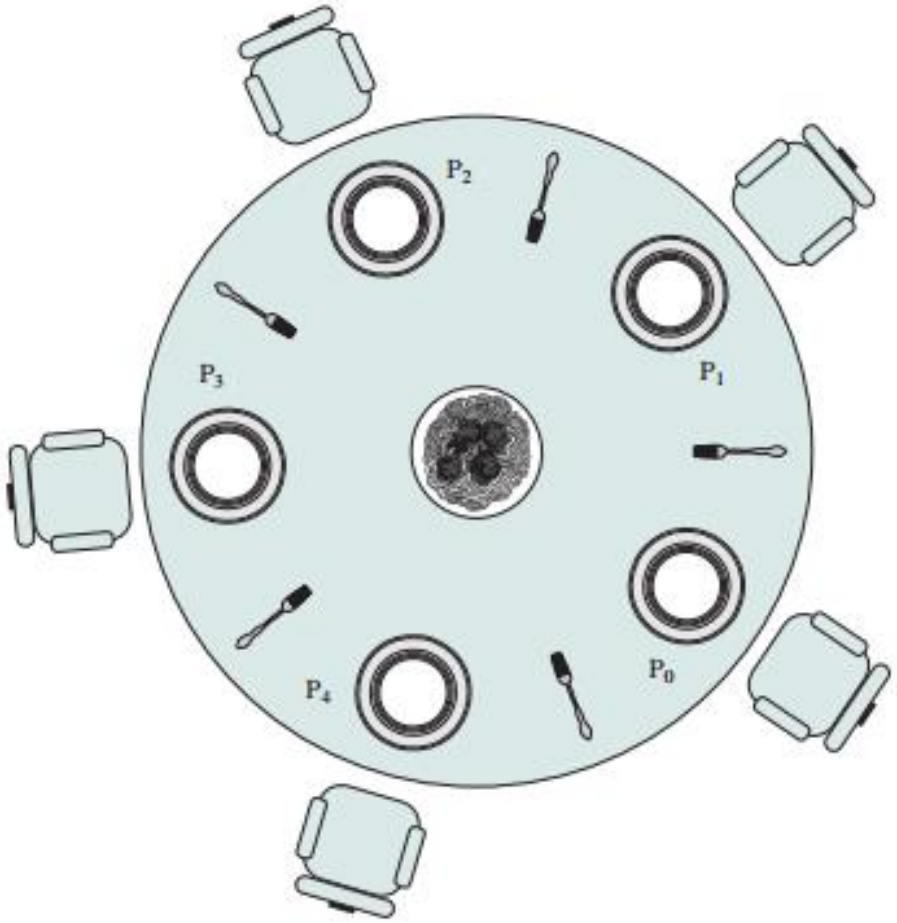
Bounded Buffer Problem

- Multiple producer-threads.
- Multiple consumer-threads.
- One bounded buffer with N entries.
- All threads modify the same buffer.
- Requirements:
  - No production when all N entries are full.
  - No consumption when no entry is full.
  - Only one thread should modify the buffer at any time.

Dining Philosophers Problem




```
#define N 5
void philosopher(int i) {
    while(true) {
        think();
        take_fork(R[i]);
        take_fork(L[i]);
        eat();
        put_fork(R[i]);
        put_fork(L[i]);
    }
}
```




```
#define N 5
void philosopher(int i) {
    while(true) {
        think();
        take_fork(R[i]);
        if (available(L[i])) {
            take_fork(L[i]);
            eat();
            put_fork(R[i]);
            put_fork(L[i]);
        } else {
            put_fork(R[i]);
            sleep(T);
        }
    }
}
```

Dining Philosophers Problem: Mutex Solution

```
#define N 5
void philosopher(int i) {
    while(true) {
        think();
        lock(mutex)
        take_fork(R[i]);
        take_fork(L[i]);
        eat();
        put_fork(R[i]);
        put_fork(L[i]);
        unlock(mutex);
    }
}
```

- Protects critical section
- Prevents deadlock
- **Disadvantage: only one philosopher can eat**

Dining Philosophers Problem: Semaphore Solution

- Uses N semaphores (s[1], s[2], …, s[N])
- All semaphores are initialized to 0
- Need a mutex
- Philosopher can be in one on the 3 states:
  - HUNGRY
  - EATING
  - THINKING
- A philosopher can only move to EATING state if neither neighbor is eating

```
void down(int *S) {
    while (*S <= 0);
    *S--;
}
```

```
void up(int *S) {
    *S++;
}
```

Dining Philosophers Problem: Semaphore Solution

```
void philosopher(int i){
    while (true) {
        think();
        take_forks();
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test_ping(i);
    unlock(mutex);
    down(s[i]);
}
```

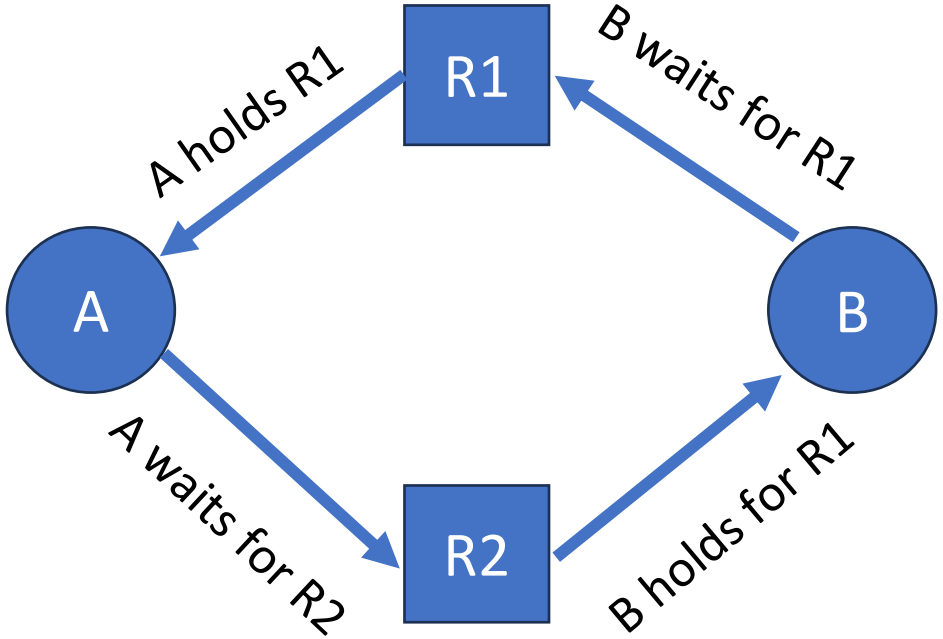
```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test_ping (LEFT);
    test_ping (RIGHT);
    unlock(mutex);
}
```

```
void test_ping(int i){
    if(state[i]==HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(s[i]);
    }
}
```

Deadlock

Deadlocks

Processes are deadlocked if each is waiting for an event that only another process can create

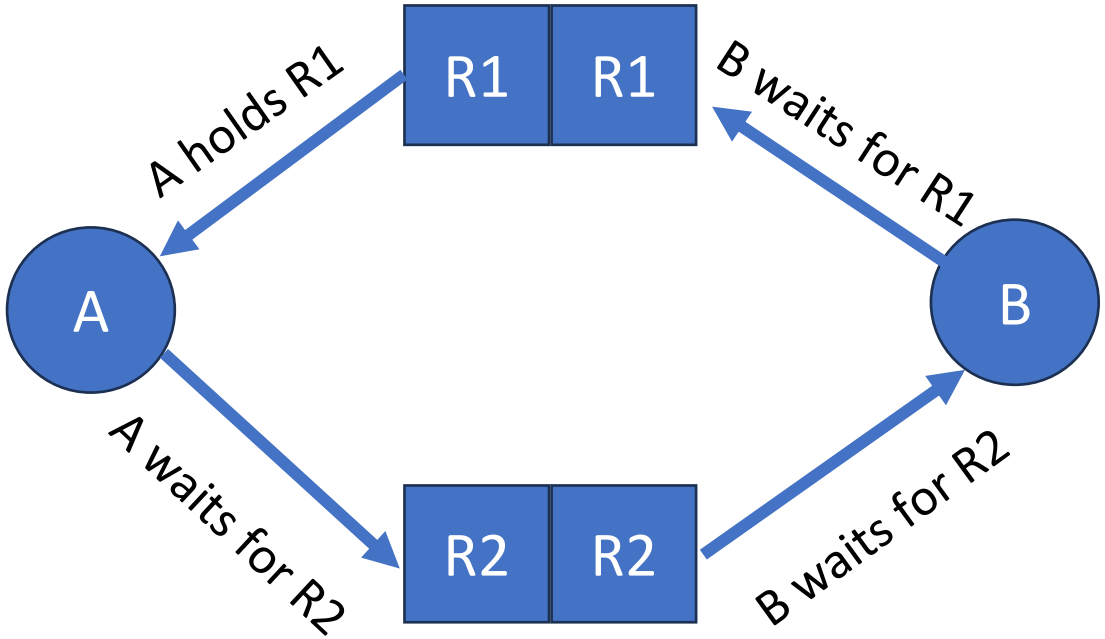


Deadlock Conditions (Coffman's conditions)

- Mutual exclusions
  - Each resource is either available or is currently assigned to one process
- Hold and wait
  - A process holding a resource requests another resource
- No preemption
  - Resources previously granted cannot be forcibly taken away
  - They must be explicitly released by the process
- Circular wait
  - There must be a circular chain of two or more processes, each of them is waiting for a resource held by the next member of the chain

Multiple Resources

Processes are deadlocked if each is waiting for an event that only another process can create



Handling Deadlocks

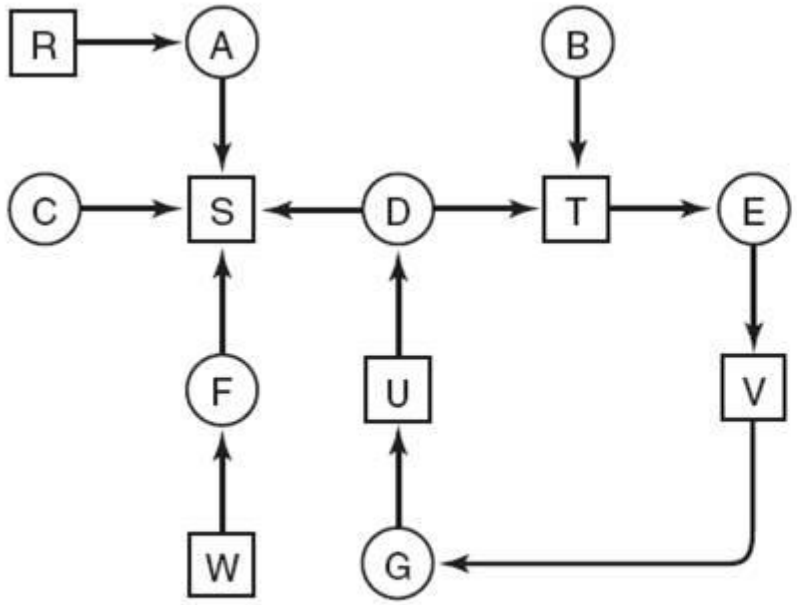
- Do not handle
- Detection and Recovery
- Avoidance
- Prevention

Deadlock Detection

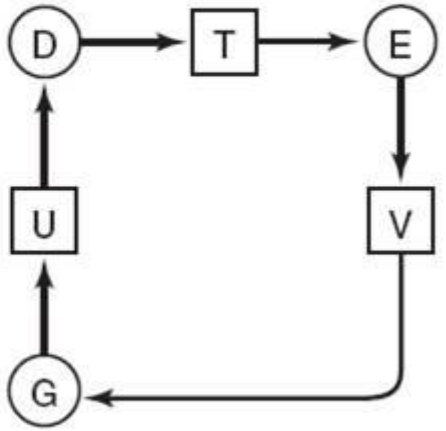
- OS keeps track of
  - Current resource allocation.
    - Which process has which resource.
    - Which process is waiting for which resource
- Use this information to detect deadlocks

Deadlock Detection

- Deadlock scenario with one resource of each type



(a)



(b)

Deadlock Recovery

What can OS do when it detects a deadlock

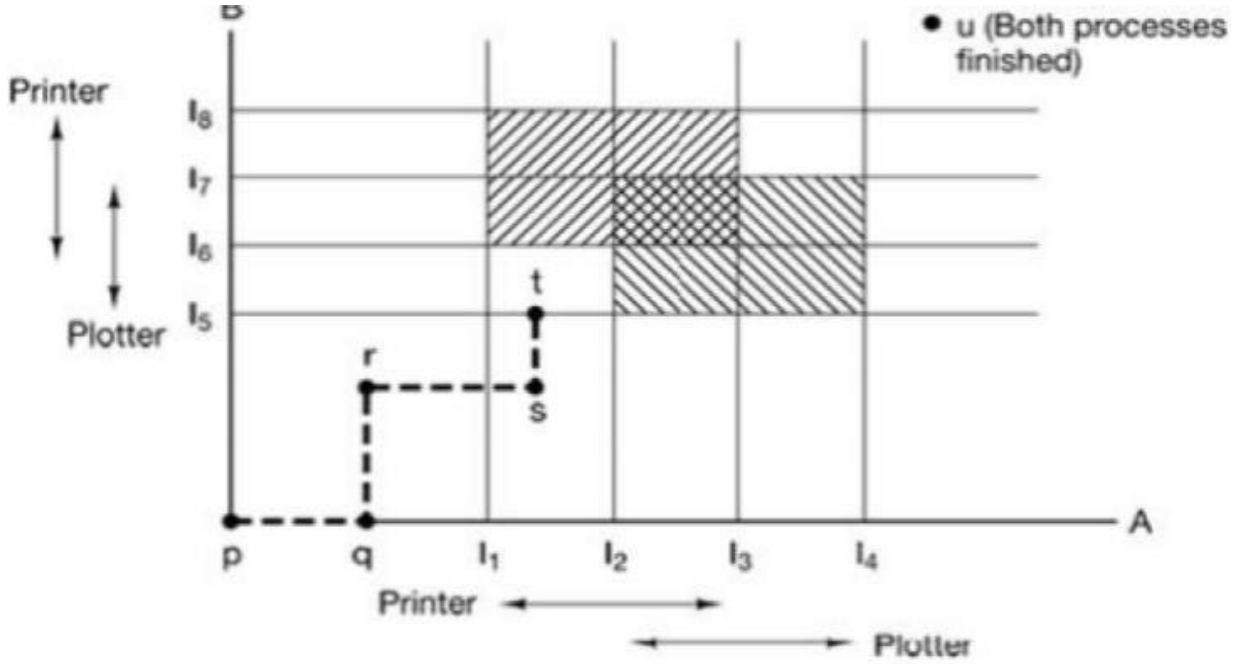
- Raise an alarm
  - Notify users/administrator
- Preemption
  - Take away a resource temporary
- Rollback
  - Checkpoint states and then rollback
- Kill process
  - Keep killing processes until deadlock is broken

Deadlock Avoidance

- Ensures system never reaches an unsafe state
- A state is safe if there is some scheduling order so that every process can run to completion even if all of them suddenly request their max number of resources

An unsafe state does not have to lead to a deadlock, it **could** leave to a deadlock

Deadlock Avoidance



Two process resource trajectories.

Banker's Algorithm

- Banker has 3 clients
  - Each client has a credit limit
  - Banker has only 10 units
  - In total 20 units are required for all clients
  - Clients declare max credit in advance
  - Banker should allocate credits so that unsafe state is not reached

Client	Has	Max
A	3	9
B	2	4
C	2	7

Safe State

Client	Has	Max
A	3	9
B	2	4
C	2	7

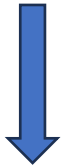
Free: 3 Units

Client	Has	Max
A	3	9
B	**4**	4
C	2	7

Free: 1 Units

Client	Has	Max
A	3	9
B	0	-
C	2	7

Free: 5 Units



Client	Has	Max
A	3	9
B	0	-
C	**7**	7

Free: 0 Units

Client	Has	Max
A	3	9
B	0	-
C	0	-

Free: 7 Units

Client	Has	Max
A	**9**	9
B	0	-
C	0	-

Free: 1 Units

Unsafe State

Client	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2 Units

Client	Has	Max
A	4	9
B	**4**	4
C	2	7

Free: 0 Units

Client	Has	Max
A	4	9
B	0	-
C	2	7

Free: 4 Units

Deadlock Prevention

- Deadlock avoidance needs to know max requests of a process
- Deadlock prevention

  Prevent at least **one of** the 4 conditions

  1. Mutual exclusion
  2. Hold and wait
  3. No preemption
  4. Circular wait

Deadlock Prevention

- **Prevent mutual exclusions**
  - Not feasible in practice
  - OS can ensure
- **Hold and wait**
  - Require all processes to request their resources before starting execution
  - May not lead to optimal usage
  - May not be possible to know requirements
- **No Preemption**
  - Preempt the resources (virtualization of resources)
- **Circular wait**
  - One way, process holding a resource cannot hold a resource and request for another one
  - Ordering requests in a hierarchical order

Deadlock Prevention

- Group resources into levels
- A process may only request resources at higher levels then any resource it currently holds
- Resources may be released in any order

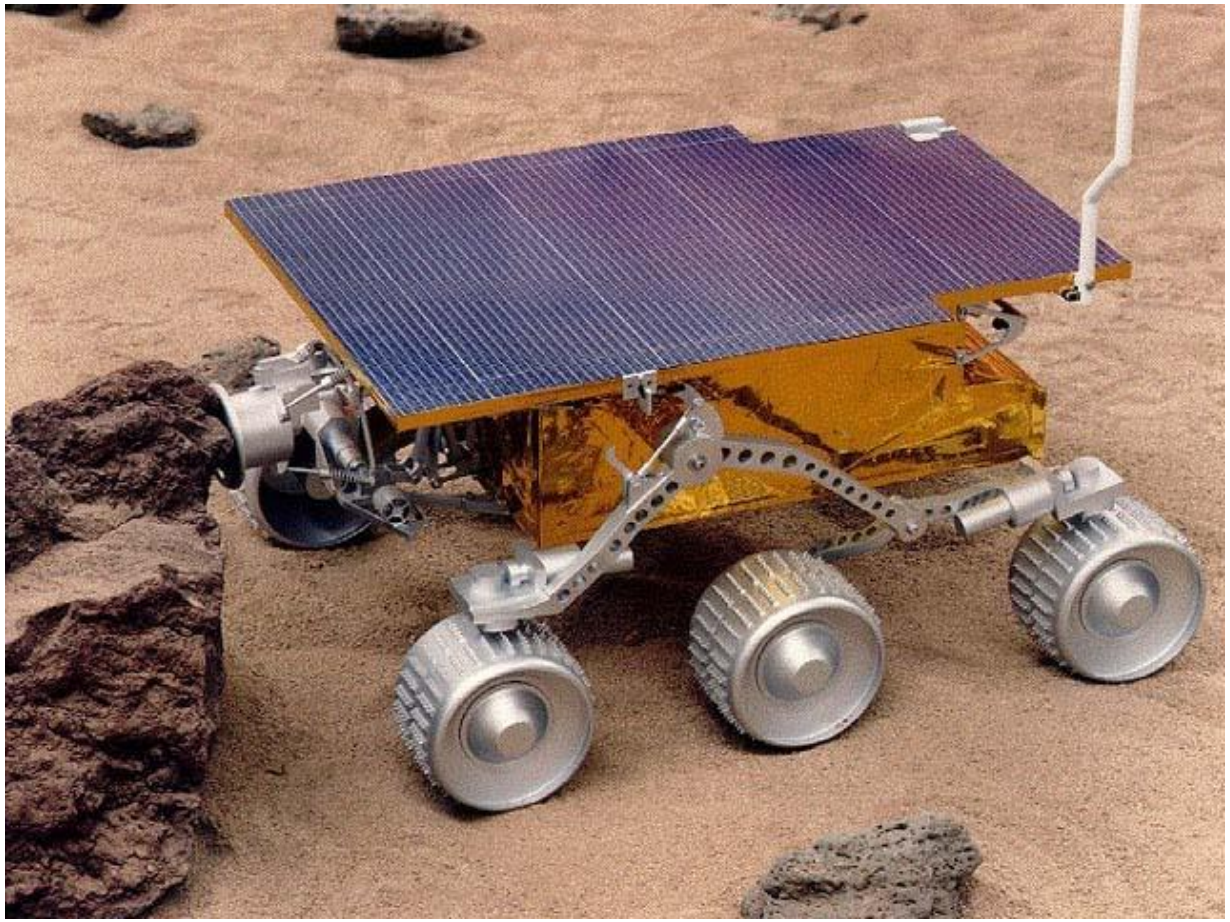
Example

Semaphores s1, s2, s3 (priorities are increasing)

- down(s1) -> down(s2) -> down(s3) **OK**
- down(s1) -> down(s3) -> down(s2) **NOT OK**

Priority Inversion

Mars Pathfinder



- Launched on December 4, 1996, landed on Mars on July 4, 1997.
- Utilized a real-time operating system (RTOS) VxWorks.
- VxWorks allows the creation and management of multiple processes, each with its own **priority** level.

Priority Based Scheduling

It's common for real-time operating systems (RTOS) like VxWorks, to implement **priority-based scheduling**.

- Each task is assigned a priority level
- Tasks with higher priority levels are scheduled to run before tasks with lower priority levels
- Preemption capability allowing a higher-priority task to interrupt the execution of a lower-priority task

Priority Based Scheduling: Advantages

- **Responsiveness:**
  - High-priority tasks are scheduled promptly, ensuring quick responses to critical events.
- **Flexibility:**
  - Allows for a flexible system design where tasks with distinct levels of urgency can be prioritized accordingly.
- **Real-Time response:**
  - Well-suited for real-time systems where meeting deadlines is crucial; critical tasks can be assigned higher priorities.
- **Resource Allocation:**
  - Enables efficient allocation of resources to higher-priority tasks, optimizing system performance.

Priority Based Scheduling: Disadvantages

- **Starvation:**
  - Lower-priority tasks may be delayed indefinitely, leading to potential starvation.
- **Priority Inversion:**
  - In scenarios where a low-priority task holds a resource needed by a high-priority task, priority inversion can occur.
- **Dynamic Changes:**
  - Managing dynamic changes in task priorities requires careful consideration to avoid unexpected behaviors.

Condition Variables

- Synchronization mechanisms need more than just mutual exclusion
- Need a way to wait for another thread to do something (e.g., wait for a character to be added to the buffer)
- Condition variables: used to wait for a particular condition to become true
- A condition variable is usually used in conjunction with a mutex lock.

Condition Variables

- Declare and initialize global variable
- Declare and initialize a condition variable
- Declare and initialize an associated mutex
- Crate threads A & B

A

- Do work
- Lock associated mutex and check condition variable
- If value does not meet some condition, perform a **blocking wait** (automatically and atomically)
- When signaled, wake up (mutex is automatically and atomically locked)
- Explicitly unlock the mutex
- Continue

B

- Do work
- Lock mutex
- Change the value of condition variable that thread A is waiting upon
- If the new value is met the condition desired by thread A, **signal** it to thread A
- Unlock the mutex
- Continue

Join

Condition Variables: Example in C

- In the main thread we have created two additional threads:
  - waiter
  - signaler
- The waiter thread should wait till the global variable is fully incremented in the signaler thread
- When the global variable is fully incremented till MAX_COUNT, the signaler thread will signal to the waiter thread through the conditional variable
- The waiter thread will continue its execution

Condition Variables: Example in C

```
int main() {
    pthread_t tid_waiter, tid_signaler;
    // Create the threads
    pthread_create(&tid_waiter, NULL, waiter, NULL);
    pthread_create(&tid_signaler, NULL, signaler, NULL);

    // Join the threads
    pthread_join(tid_waiter, NULL);
    pthread_join(tid_signaler, NULL);
    return 0;
}
```

```
#define MAX_COUNT 10
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Mutex
pthread_cond_t condition = PTHREAD_COND_INITIALIZER; // Condition variable
int count = 0; // Shared variable
```

```
int main() {
    pthread_t tid_waiter, tid_signaler;
    // Create the threads
    pthread_create(&tid_waiter, NULL, waiter, NULL);
    pthread_create(&tid_signaler, NULL, signaler, NULL);

    // Join the threads
    pthread_join(tid_waiter, NULL);
    pthread_join(tid_signaler, NULL);
    return 0;
}
```

```
void *waiter(void *arg) {
    pthread_mutex_lock(&mutex); // Lock the mutex

    while (count < MAX_COUNT) {
        // Wait for the condition to be met
        // Release the mutex and wait for a signal
        pthread_cond_wait(&condition, &mutex);
    }

    // When the condition is signaled, print count
    printf("Waiter Thread: Count = %d\n", count);

    pthread_mutex_unlock(&mutex); // Unlock the mutex
    pthread_exit(NULL);
}
```

```
void *signaler(void *arg) {
    pthread_mutex_lock(&mutex); // Lock the mutex

    // Do some work before signaling the condition
    for (int i = 0; i < MAX_COUNT; ++i) {
        count++;
        printf("Signaler thread count to %d\n", count);
    }

    // Signal the condition
    pthread_cond_signal(&condition);

    pthread_mutex_unlock(&mutex); // Unlock the
    mutex
    pthread_exit(NULL);
}
```

Condition Variables: Spurious Wake-ups

- Spurious wake-ups happen when a thread waiting on a condition variable (using functions like pthread_cond_wait()) wakes up even though no explicit signal was sent to wake it.
- The behavior is often a result of how the operating system manages threads and resources. It's considered a part of the design of condition variables in some threading libraries.
- It's also influenced by the internal mechanisms of thread scheduling and signaling, which can sometimes lead to a thread waking up without a signal due to the underlying system behavior.

Condition Variables: Handling Spurious Wake-ups

- **Looping with Condition Check**

To handle spurious wake-ups, it's common to use a loop with a condition check

```
pthread_mutex_lock(&mutex);
while (!condition_is_met) {
    pthread_cond_wait(&condition, &mutex);
}
pthread_mutex_unlock(&mutex);
```

ABA problem

- The **ABA problem** occurs during synchronization, when a location is read twice , has the same value for both reads.
- The read value being the same twice is used to conclude that nothing has happened in the interim

- However, another thread can execute between the two reads and change the value, do other work, and then change the value back, thus fooling the first thread into thinking nothing has changed even though the second thread did work that violates that assumption.

ABA problem

The ABA problem occurs when multiple threads (or processes) accessing shared data interleave. Below is a sequence of events that illustrates the ABA problem:

1. Process P1 reads value A from some shared memory location,
2. P1 is preempted, allowing process P2 to run,
3. P2 writes value B to the shared memory location
4. P2 writes value A to the shared memory location
5. P2 is preempted, allowing process P1 to run,
6. P1 reads value A from the shared memory location,
7. P1 determines that the shared memory value has not changed and continues.

Although P1 can continue executing, it is possible that the behavior will not be correct due to the "hidden" modification in shared memory.

ABA problem: Workaround

- **Tagged state reference**
  A common workaround is to add extra "tag" or "stamp" bits

Monitors

- Brinch Hansen (1973) and Hoare (1974) proposed a higher-level synchronization primitive called a monitor.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Syntax of Monitors

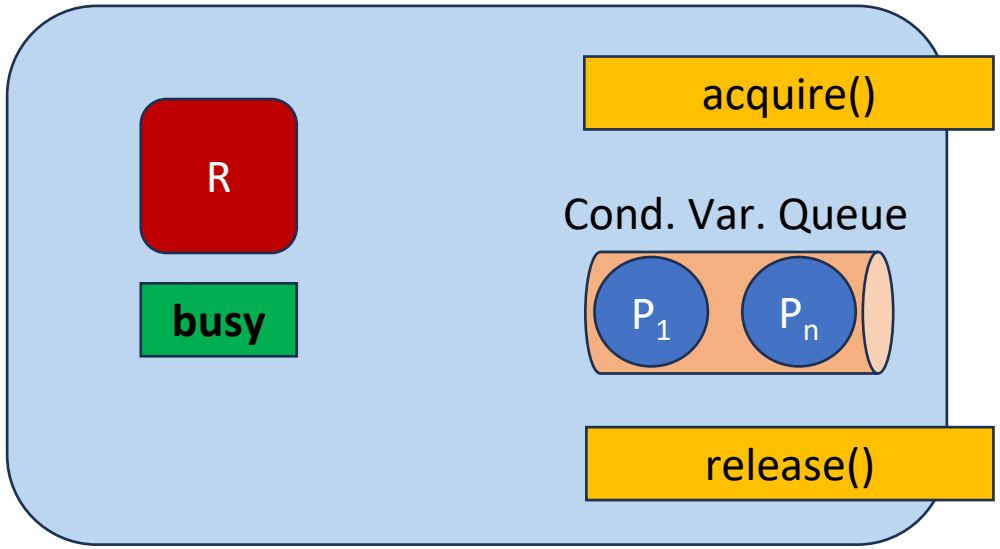
```
monitor my_monitor
{
    Shared variables
    Function F1(...) {
        ...
    }
    …
    Function Fn(...) {
        ...
    }
    Initialization() {
    }
}
```

- Monitor's Functions can access only the variables defined in the monitor
- Local variables in the monitor can be accessed only by the local functions
- The monitor ensures that only one process at a time can be active within that monitor
  - It uses Conditional variables
  - wait() and signal() mechanism

Syntax of Monitors

```
monitor ProducerConsumer {
    condition_variable full, empty;
    int count = 0;

    void insert(int item) {
        if (count == N) wait(full);
        Insert_item(item);
        count := count + 1;
        if (count == 1) signal(empty);
    }

    int remove() {
        if (count == 0) wait(empty);
        remove = remove_item();
        count := count - 1;
        if (count == N – 1) signal(full);
    }

    count := 0;
}
```

Monitors: Single Resource Allocation Problem




```
monitor SingleResource{
    condition_variable nonbusy;
    bool busy = false;

    void acquire() {
        if (busy) {
            wait(nonbusy);
        }
        busy = true;
    }

    void release() {
        busy = false;
        signal(nonbusy);
    }
}
```