SoC 101:

*a.k.a., "**Everything you wanted to know about a computer but were afraid to ask**"*

Lecture 7: Operating Systems

Prof. Adam Teman

EnICS Labs, Bar-Ilan University

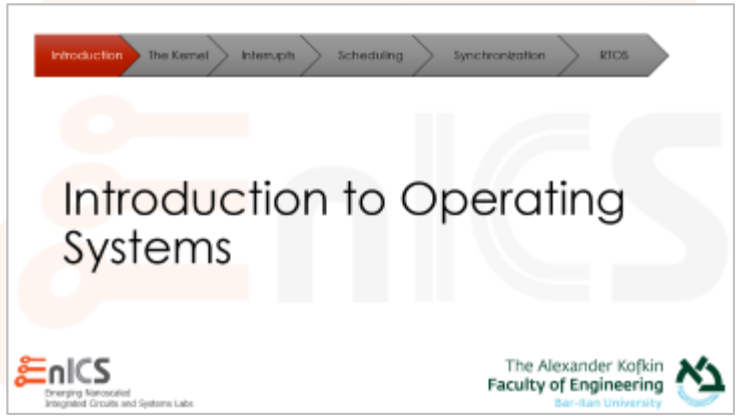
2 October 2023

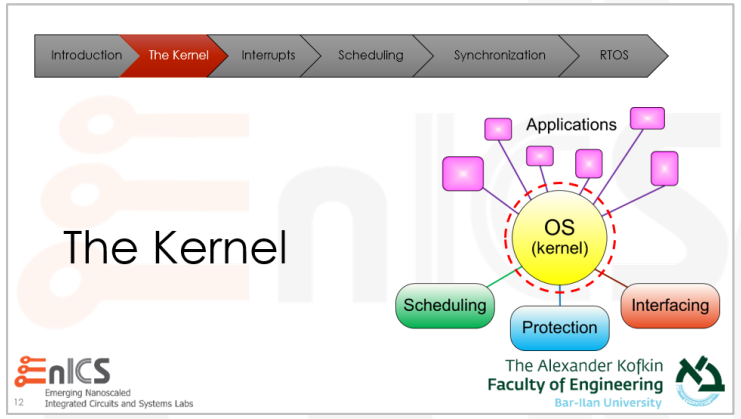
Emerging Nanoscaled Integrated Circuits and Systems Labs

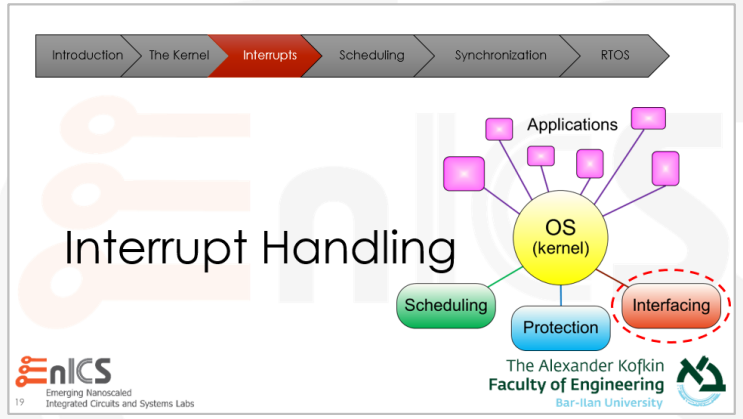
The Alexander Kofkin Faculty of Engineering

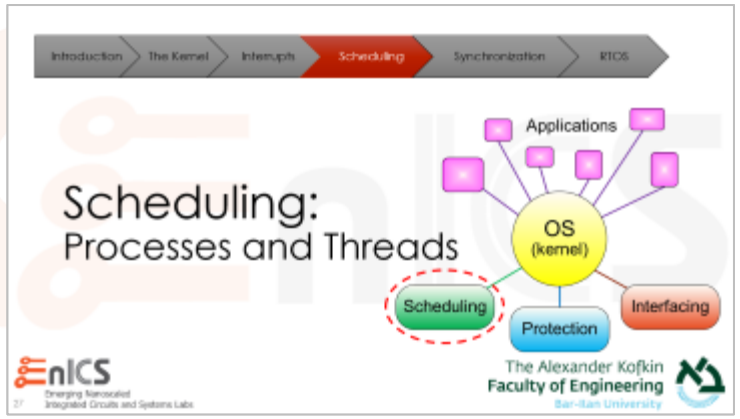
Bar-Ilan University

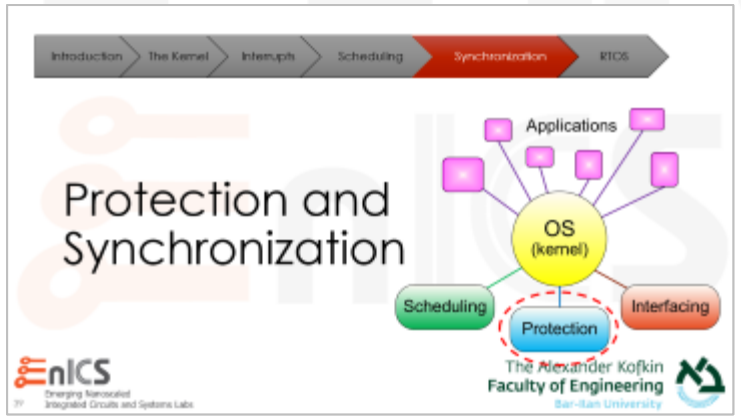
Lecture Overview

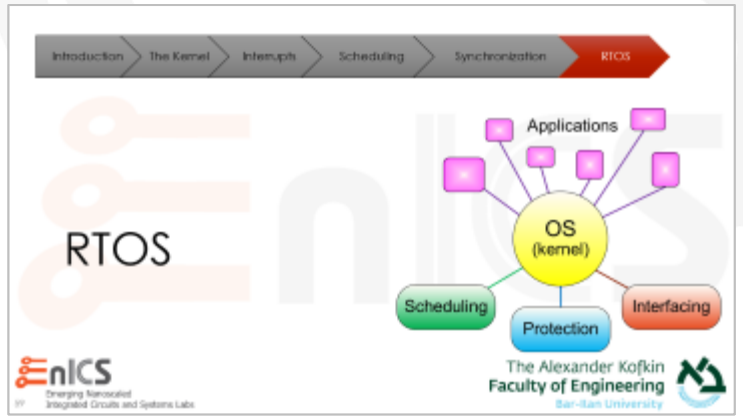












Introduction to Operating Systems

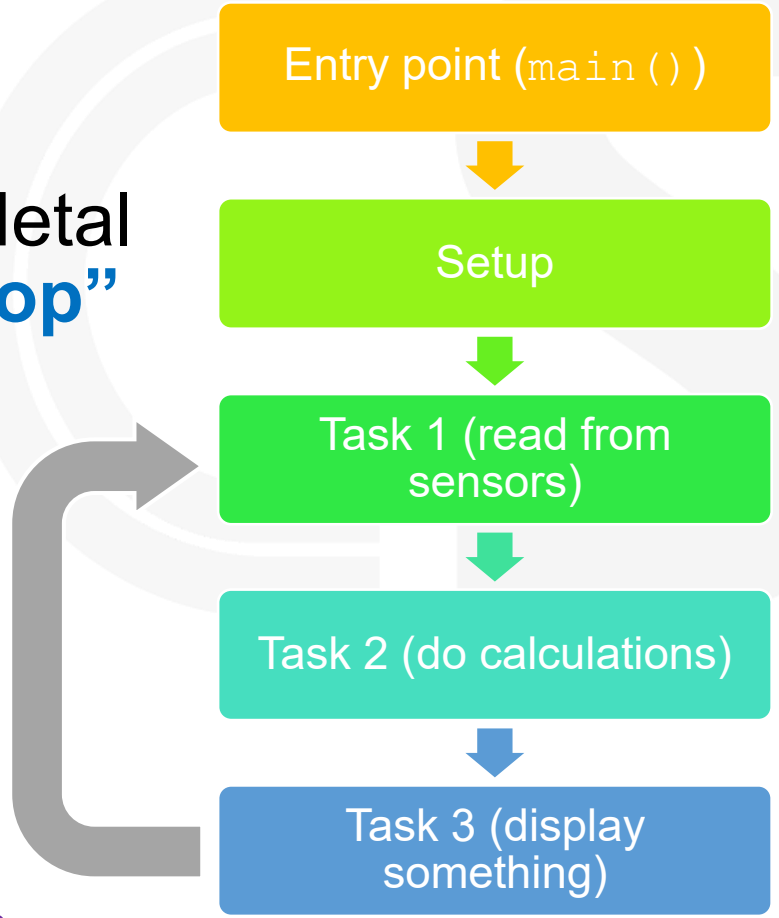
Emerging Nanoscaled
Integrated Circuits and Systems Labs

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Where we started…

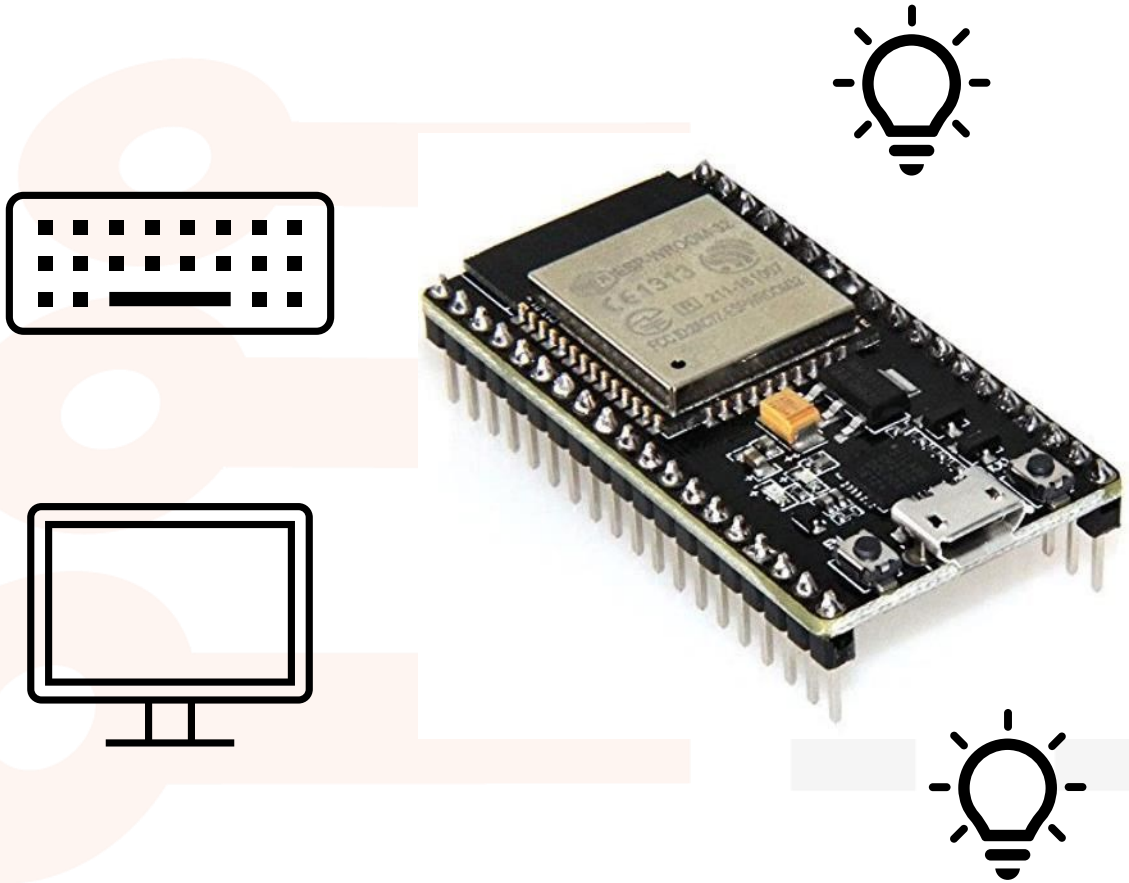
- What do we need in order to push a button and blink a LED?

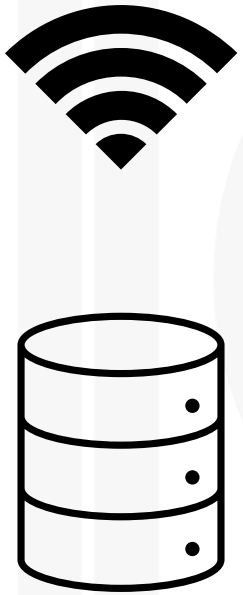
The Bare Metal
"Super Loop"

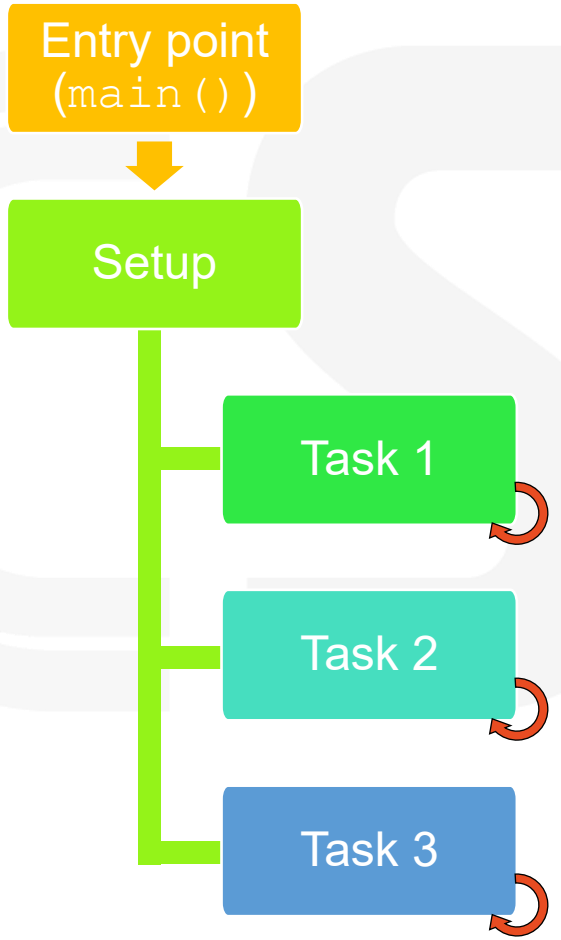


- Easy to implement
- Little overhead
- Easy to debug
- Suitable for a handful of tasks

But this can quickly get complicated…

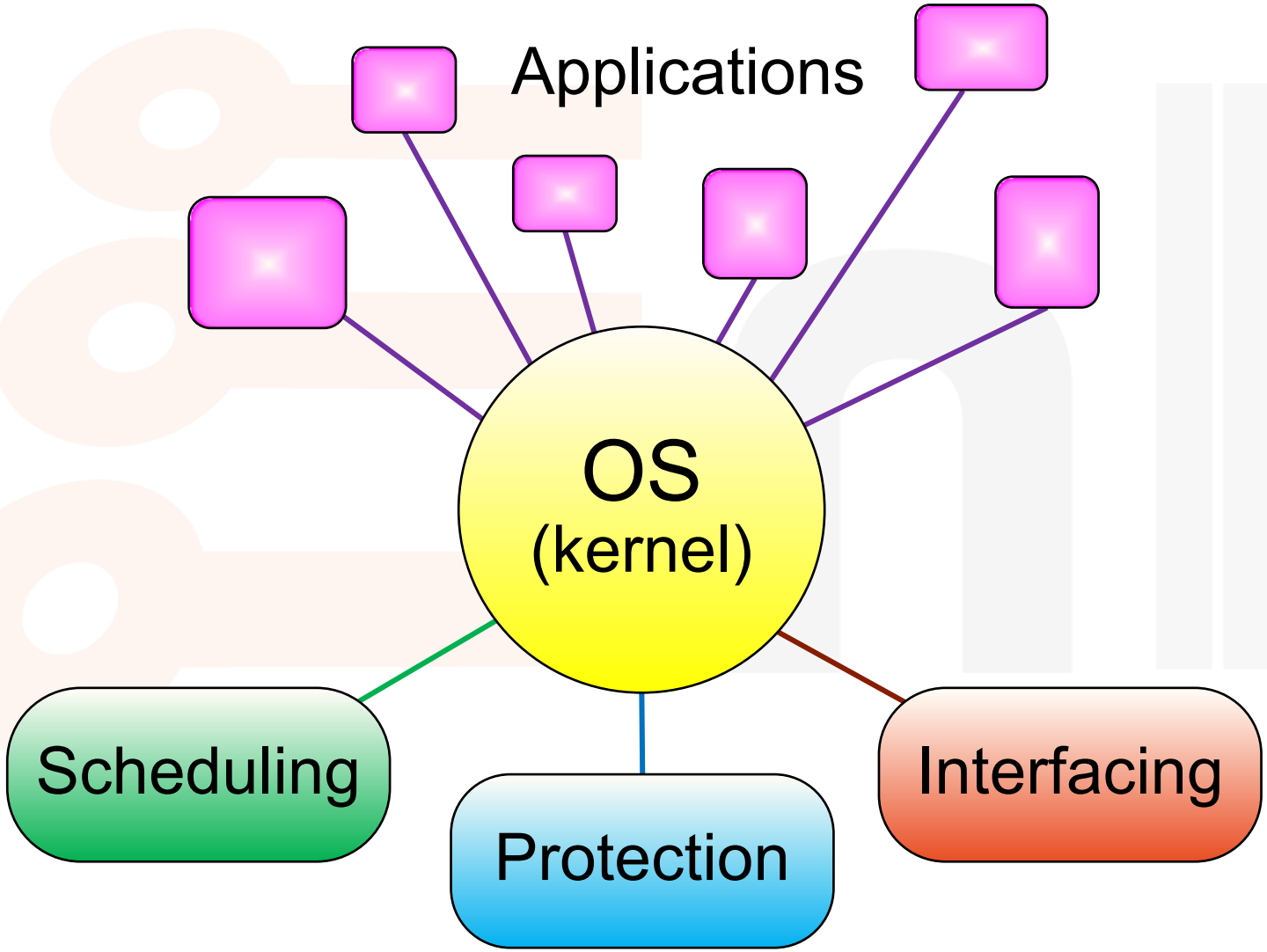






- **Maybe, we need something a bit more sophisticated….**

The Operating System



Referee

- Resource allocation among users, applications
- Isolation of different users, applications from each other
- Communication between users, applications

Illusionist

- Each application appears to have the entire machine to itself
- Infinite number of processors, infinite amount of memory, reliable storage, reliable network transport

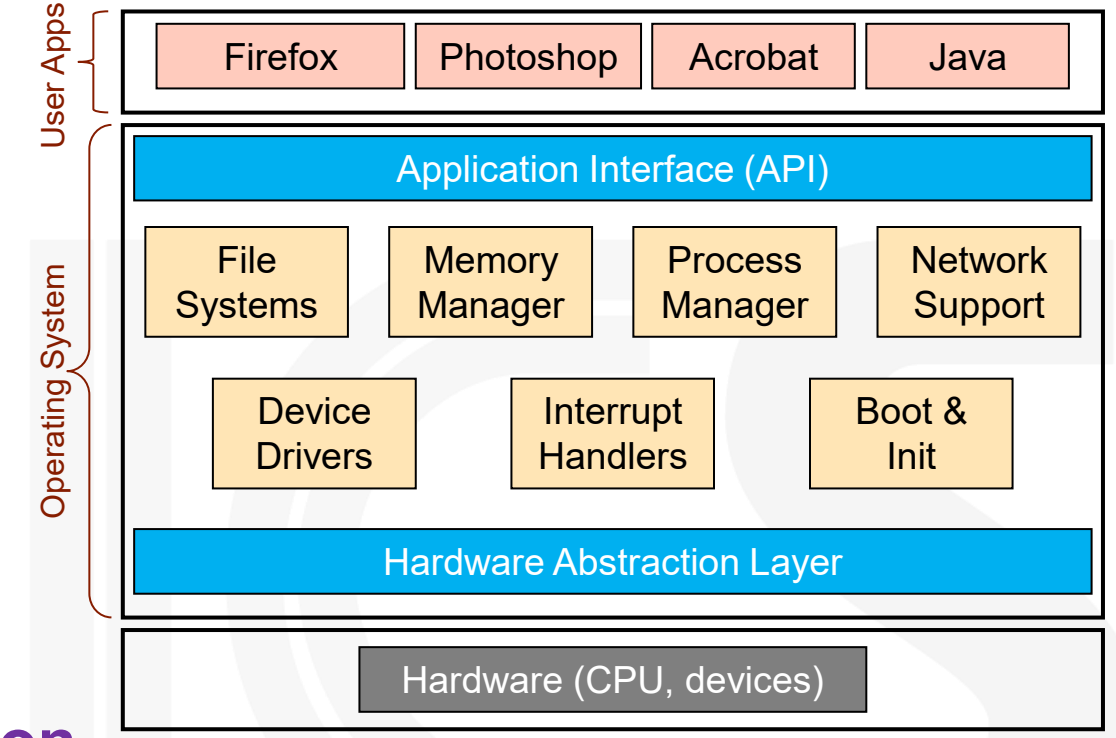
Glue

- Libraries, user interface widgets, …
- Reduces cost of developing software

The Operating System

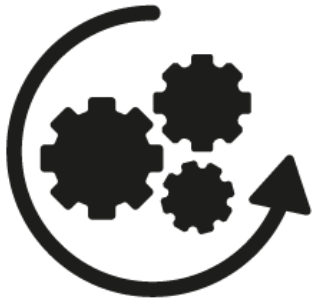
- *"The OS is everything you don't need to write in order to run your application"*

- **Primary goals of an Operating System:**
  - To **simplify** program execution.
  - To use computer **hardware** efficiently.
  - To improve overall system **reliability**.
  - To provide isolation, security and **protection**.
  - To make application **software** portable and versatile.

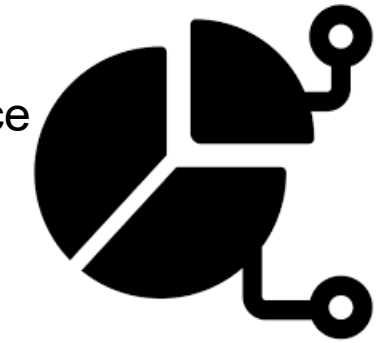


Source: Uwash, CSE 451

Process Management (Time Sharing)

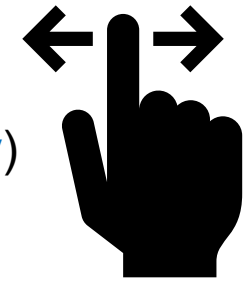


System Resource Management (Multiplexing)

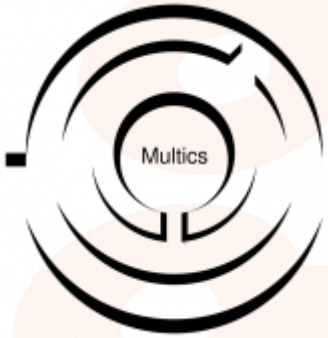


Protection of resources (Isolation)

Interface (Simplicity)



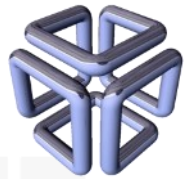
Evolution of Operating Systems



MULTICS
Bell Labs, GE, MIT
(1964-1969)

Unix
Ken Thompson,
Dennis Richie,
AT&T Bell Labs
(1969)

Unix System V
AT&T
(1974)



SGI-IRIX
Silcon Graphics

AIX
IBM

HP-UX
HP

Solaris (SunOS)
Oracle/Sun

BSD
Berkeley (1978)

NEXTSTEP

DarwinOS

Mac OS X (2001)
OS X (2012)
MacOS (2016)

iOS
(2007)

MS-DOS
Microsoft
(1981)

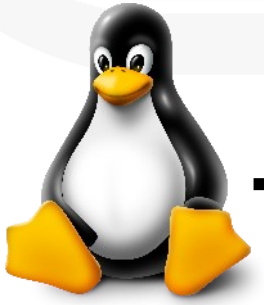
Windows 1 (1985)
Windows 95 (1995)

DEC (1977)

Windows NT (1993)

Windows XP (2001)

GNU
Richard Stallman
(1983)



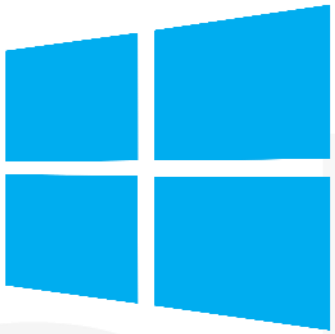
Linux
Linus Torvalds
(1991)

Android
(2008)

Main Operating Systems Today

ANDROID

42%



Windows®

28%

16%

Mac OS X

10%

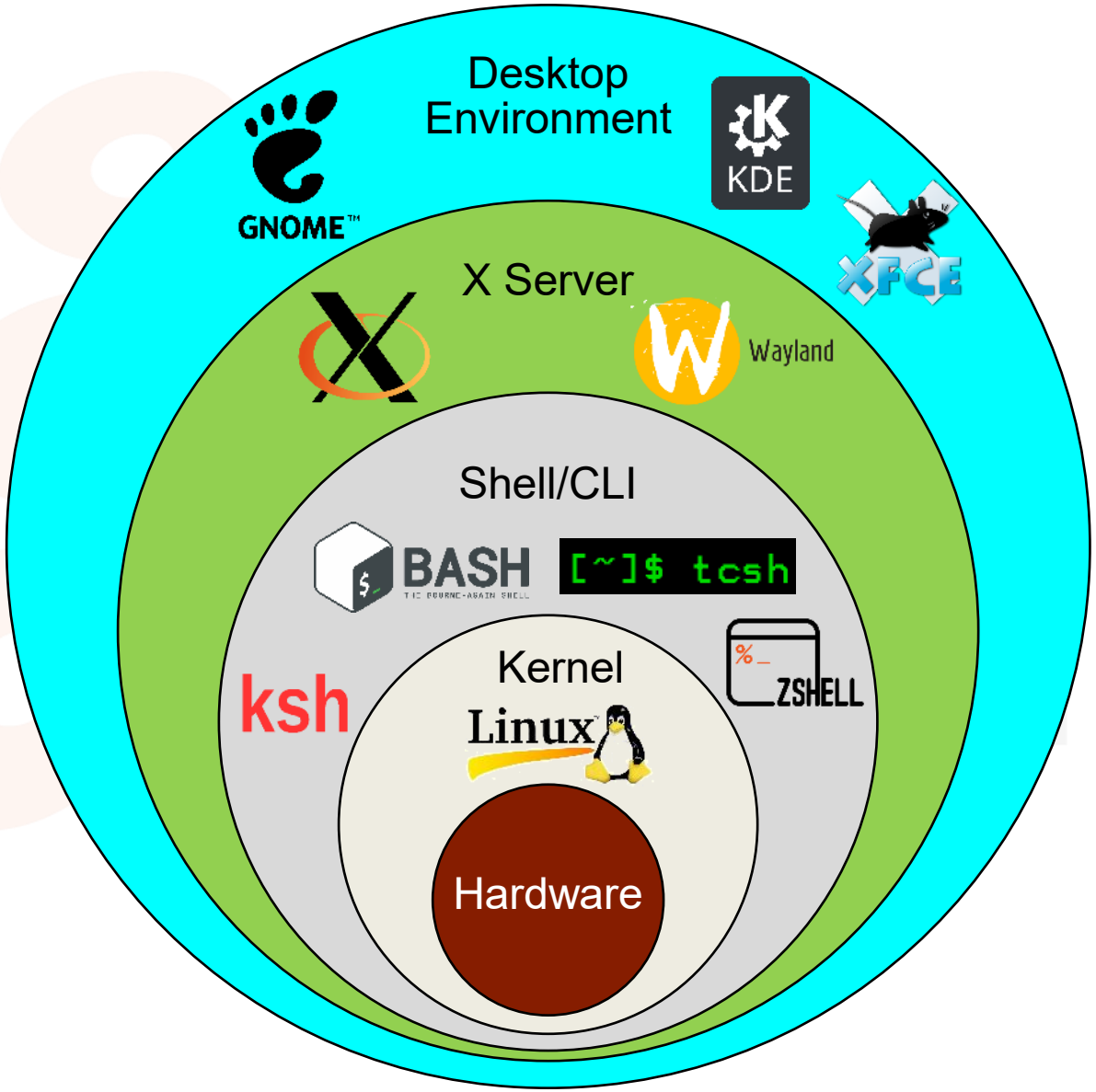
2%

Data according to Google Bard 7/2023

A Final Note: The Unix-Like Terminology

- **POSIX (Portable OS Interface) is a standard for "Unix-like" Operating Systems**
  - Basically, APIs that all compatible OSs understand
  - e.g., `cd`, `ls`, `fork`, `grep`, `echo`, `$HOME`, `$PATH`, etc.
- **Unix stopped being free in 1982**
  - So, Richard Stallman (MIT) started the GNU ("GNU's Not Unix!") project.
  - GNU has become a collection of free software (e.g., GCC, glibc, GDB, Bash)
- **Linux is a Kernel developed by Linus Torvalds in 1991**
  - Combining Linux with GNU provides a fully functional and free OS.
- **GNU/Linux are usually provided as "Distros"**
  - A Distro also includes the X Server, the Desktop, the Package Manager, etc.
  - Popular distros: Fedora (RedHat), Debian, Suse, Ubuntu, CentOS, Arch,…

A Final Note: The Unix-Like Terminology



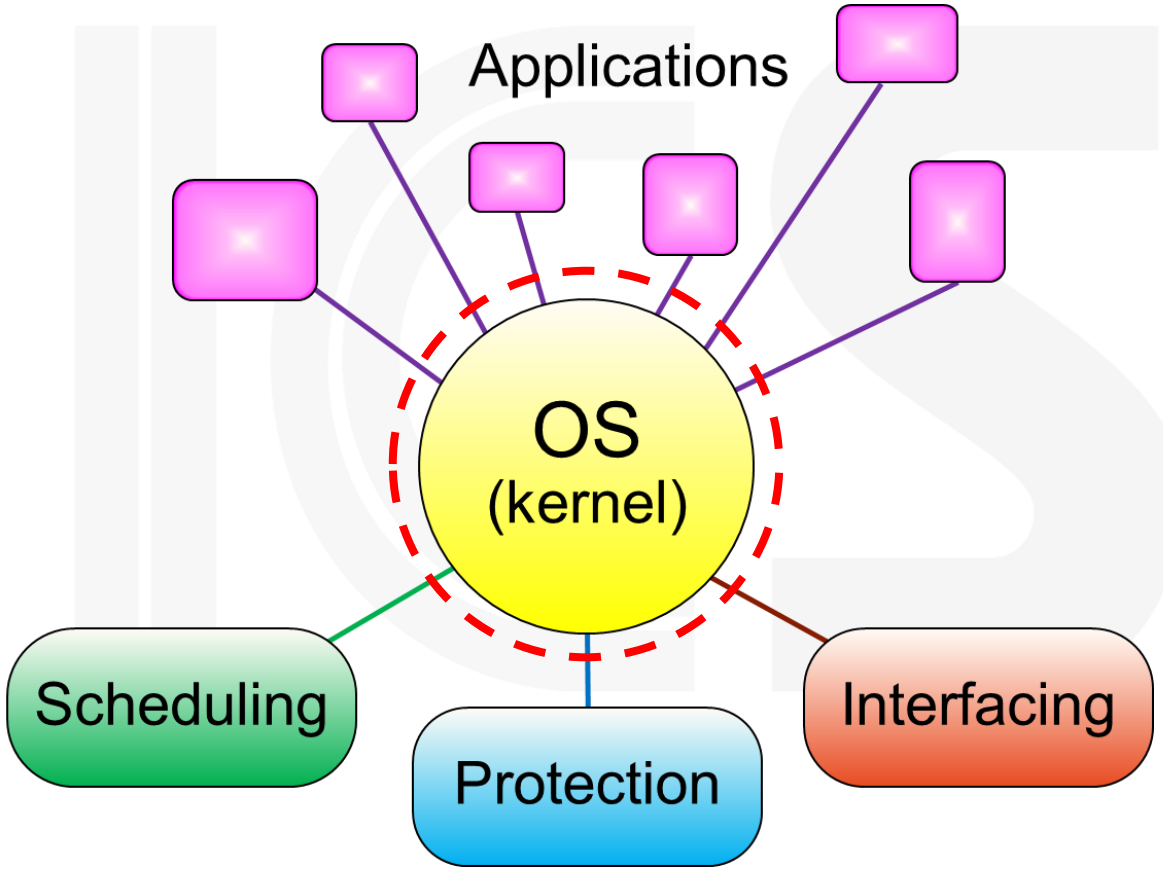
Distros

debian
fedora
openSUSE
Red Hat
Ubuntu
archlinux
CentOS
slackware linux
linuxmint

Package Managers

```
%> apt-get install
%> yum install
%> dnf install
%> pacman -S
%> zypper install
%> emerge -ask
```

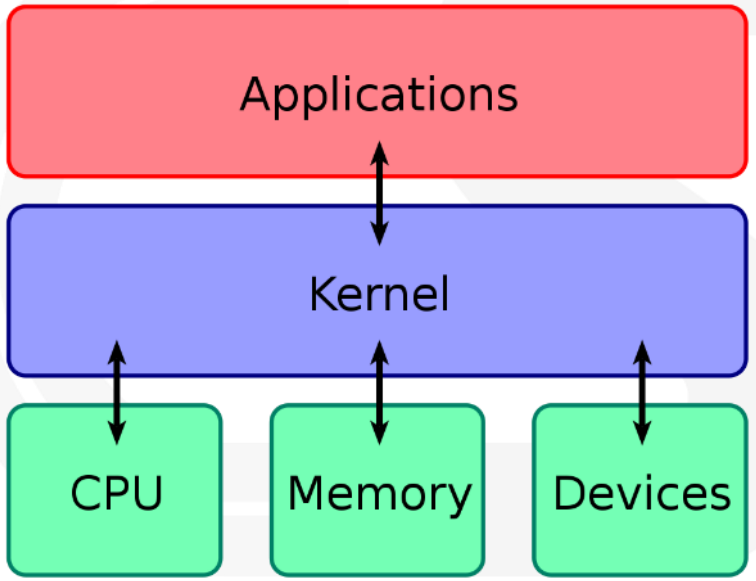
The Kernel



Emerging Nanoscaled
Integrated Circuits and Systems Labs
The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

The Kernel

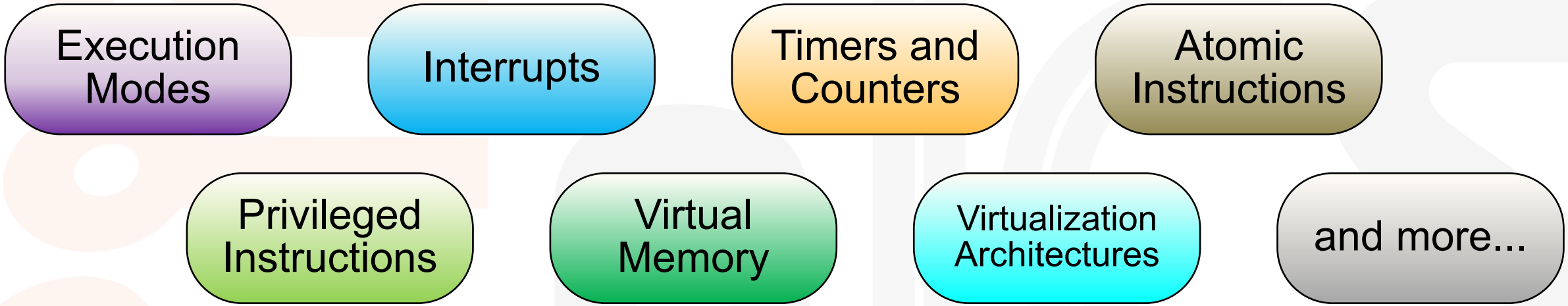
- **The Kernel is the core of the Operating System**
  - The Kernel is the first program (after the bootloader) that is loaded into memory and run.
  - The Kernel is always resident in memory.
  - The Kernel is the program that executes forever.
- **The Kernel has complete control over everything in the system.**
  - The Kernel facilitates interactions between hardware and software.
  - Everything else is an application with respect to the Kernel.
- **The Kernel is *privileged* and can carry out many operations that no other program is allowed to.**



Source: Wikipedia

Hardware Support

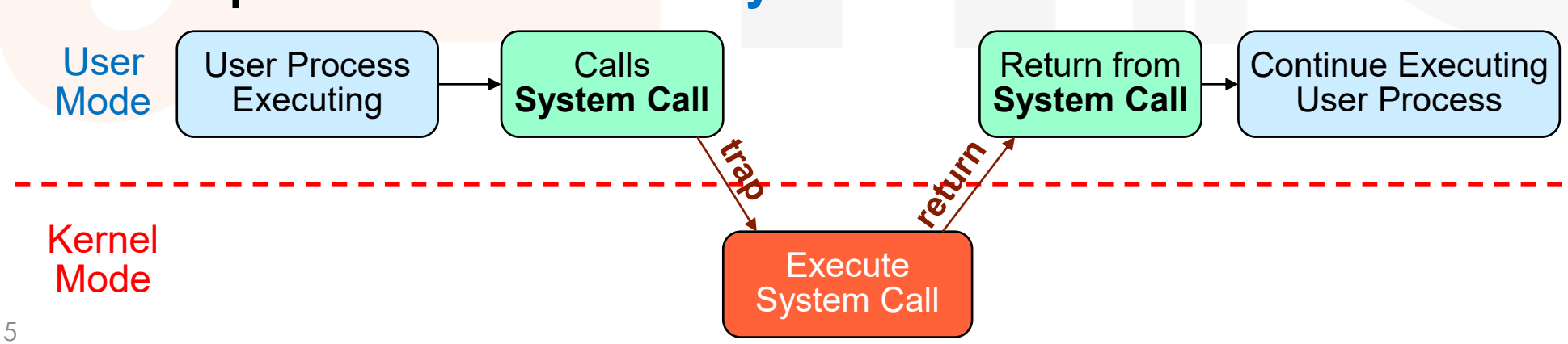
- **Running an OS requires specific hardware functionality, including:**



- **RISC-V defines a set of 4096 Control Status Registers (CSRs) that are primarily used by the OS to support its operations.**
  - CSRs are accessed by special instructions (e.g., `csrr`/`csrw` for reading/writing)
  - The majority of the CSRs and their functionalities are described in Volume II of the RISC-V Instruction Set Manual, a.k.a., the "Privileged Spec"

Execution Modes and System Calls

- **To provide privileged operation to the Kernel, the OS must support at least two distinct execution modes:**
  - **Kernel Mode**: Only for the kernel, providing very high access permissions.
  - **User Mode**: Used by all other applications, allowing restricted access.
- **A user application is not allowed to carry out privileged operations**
  - Such as reading a file, launching a process, asking for memory (malloc), etc.
  - Therefore, control must be transferred to the Kernel running in Kernel mode.
- **This operation is called a "System Call"**

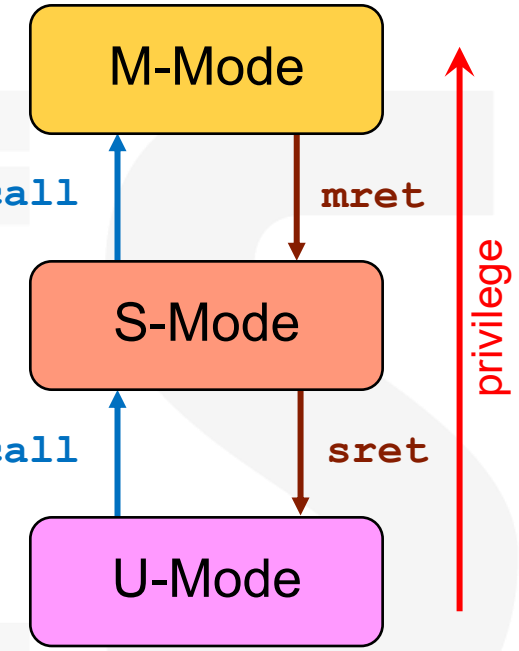


Note that Linux has ~300 system calls

`read()`, `write()`, `open()`, `close()`, `fork()`…

Privileged Modes in RISC-V

- **The RISC-V architecture supports three execution modes*:**
  - User Mode (U) → Lowest Privilege
  - Supervisor Mode (S) → Non-mandatory middle mode
    - Used to abstract away hardware from OS implementation.
  - Machine Mode (M) → Highest Privilege – mandatory in RISC-V
- **Each mode has its own set of CSRs, starting with U/S/M**
  - For example, `mstatus` is the machine mode status register while `sstatus` is the supervisor mode status register.
- **System calls are achieved with the `ecall` and `mret`/`sret` instructions**
  - `ecall` performs a system call from the U/S/M modes to a higher privilege.
  - `mret`/`sret` return from a system call back to the calling application.

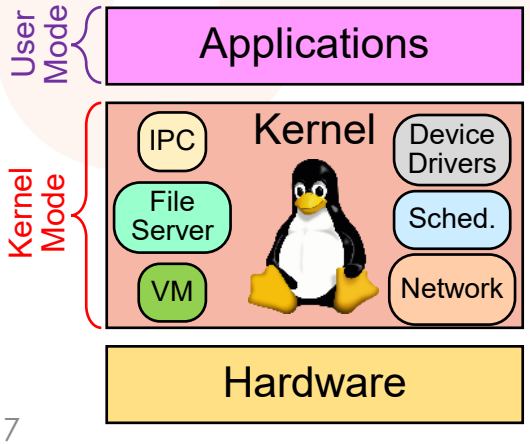


* A fourth mode "Hypervisor Mode" is not yet fully specified.
And a special "Debug Mode" is used by debuggers at a high privilege

Kernel Architectures

- **There are several options for creating Kernels, trading off performance, complexity, protection, and more. The basic architectures are:**
  - **Monolithic Kernel**: All main OS operations run as a single program in Kernel Mode, sharing the same address space.
  - **Microkernel**: The Kernel includes only the minimum, essential functionalities of the OS, while other functionalities (services) are run in User Mode.
  - **Hybrid Kernel**: A Microkernel with some additional services inside the Kernel.

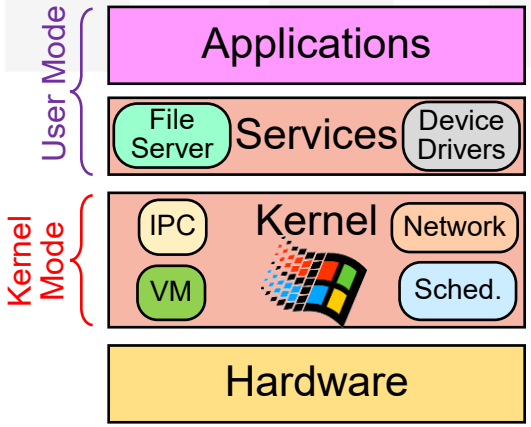
Monolithic Kernel



Advantages:
- Fast
- Easy to develop

Disadvantages:
- Large
- Hard to add functionality
- Less Secure
- Failures critical

Hybrid Kernel

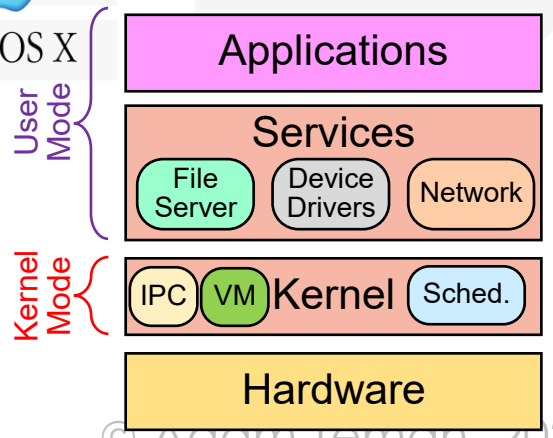


Advantages:
- Small
- Secure
- Modular
- Easy to maintain

Disadvantages:
- Slow (many system calls)

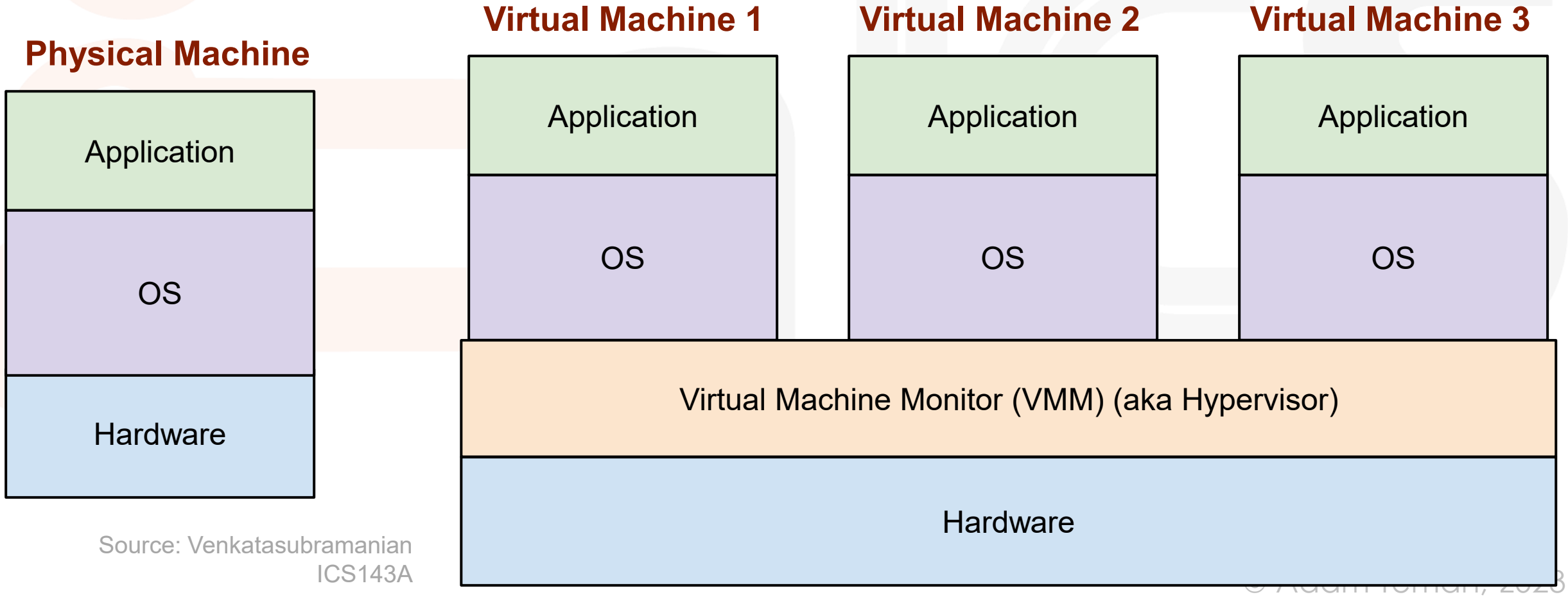
Mac OS X

Microkernel



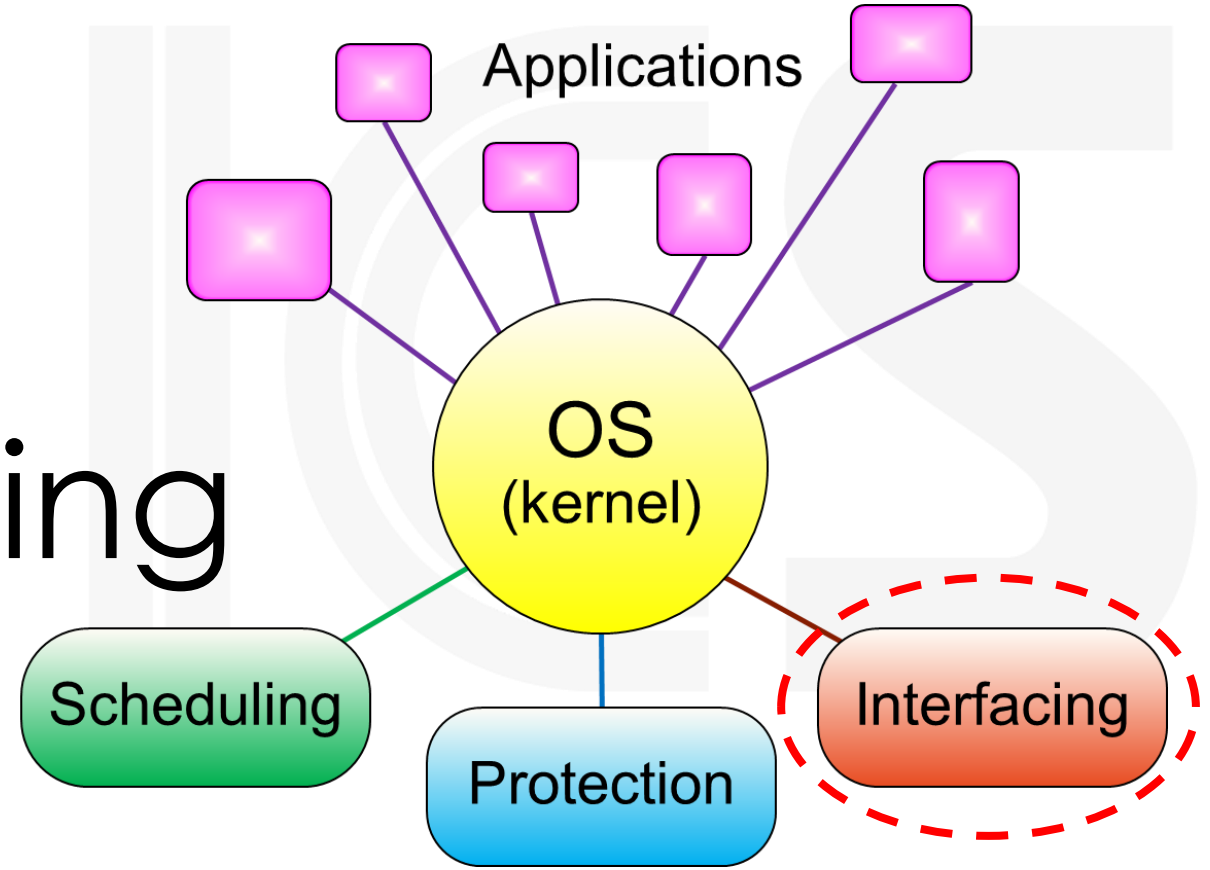
Virtual Machines

- **Virtual Machines provide another level of separation from the hardware (*Hypervisor*) to allow running several Operating Systems in parallel.**



Source: Venkatasubramanian ICS143A

Interrupt Handling

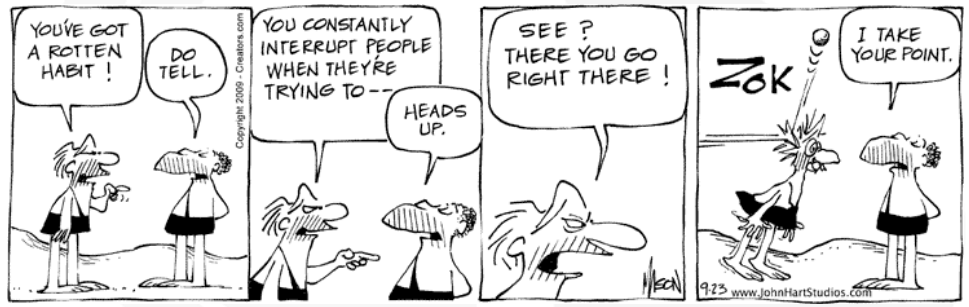


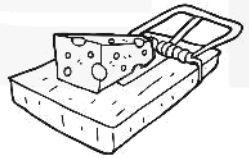
Emerging Nanoscaled
Integrated Circuits and Systems Labs
The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Interrupts and Exceptions

- **In the previous section, we were acquainted with System Calls**
  - A program initiates a System Call, when it wants to run a privileged instruction.
  - The program's operation is halted – *interrupted* – and the OS is invoked.
  - This type of interrupt is referred to as a Software Interrupt or a Trap.
- **An interrupt is a response by the processor to an event that needs attention from the software.**
- **We generally differentiate between three categories of interrupts:**
  - Software Interrupts (**Traps**):
    - An **intended** exception thrown by program code, e.g., a System Call.
  - Asynchronous **Interrupts**:
    - An interrupt coming from **outside the CPU**, e.g., a mouse click.
  - Synchronous **Exceptions**:
    - An **unexpected problem** with the instruction, e.g., divide-by-zero.

Source: John Hart Studios





$$\frac{1}{0} = ?$$

Interrupts: Teman Analogy

- **Ahh… kids are finally asleep…**
  - Time for winding down with some TV…
  - Damn! Someone logged out of Netflix…

    System Call!
  - "Honey, what's the password?"
- **Great! A new episode of "Formula 1: Drive to Sur…"**
  - Wow! Look at Max Verstappen go!

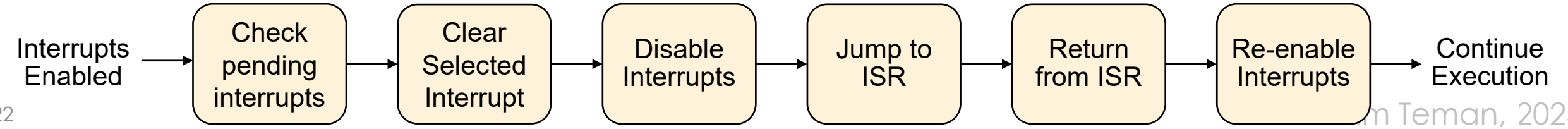
    Exception!
  - Go reset the wifi…
- **Okay, Sunday race is starting…**

  Asynchronous Interrupt!
  - "Dad, I can't sleep…"

Interrupt Handling

Interrupts should be as fast as possible, because nothing else can happen when an interrupt is being serviced.

- **Upon receiving an interrupt, it must be handled by the operating system:**
  - Handling is a done by a procedure called an Interrupt Service Routine (ISR).
  - The operating system is invoked (Kernel Mode) and the ISR is called.
  - Synchronous interrupts (traps and exceptions) do this immediately.
  - Asynchronous interrupts go through a slightly different process.
- **The operating system decides when to allow asynchronous interrupts**
  - If interrupts are enabled, the CPU will check for pending interrupts.
  - If several interrupts are pending, the OS will decide on which one to handle.
  - The selected interrupt is then cleared and interrupts are disabled.
  - Only then, the ISR is called, and upon returning, interrupts are re-enabled.

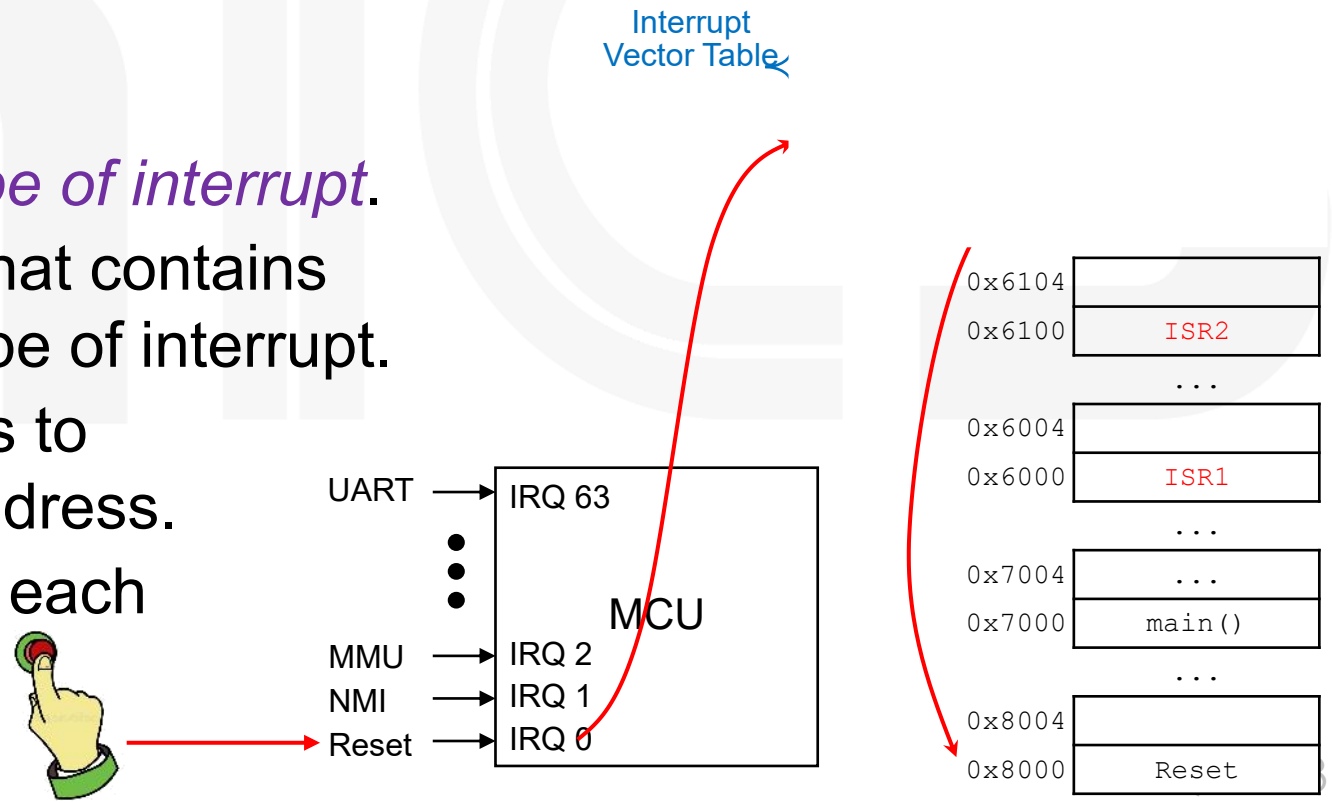


Interrupt Handling – Vectored Interrupts

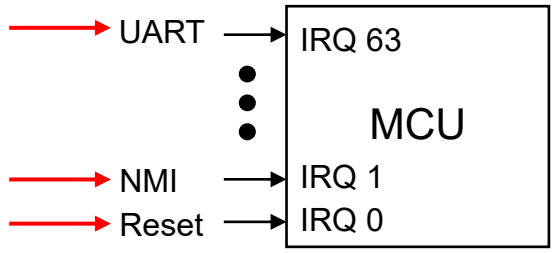
- **Which ISR is called upon receiving an interrupt?**
  - Some systems have a single ISR that handles all interrupts.
  - However, this can result in long, complex code → slow interrupt handling.
- **The solution…**

Indirection, of course!

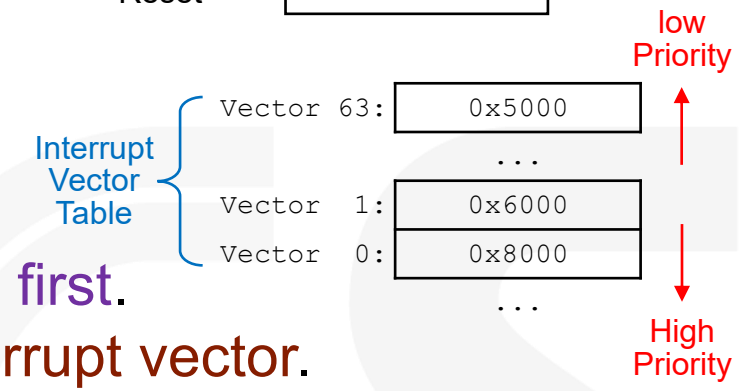
  - Provide a *different ISR* for each *type of interrupt*.
  - Use an interrupt vector → a table that contains the address of the ISR for each type of interrupt.
  - The encoding of the interrupt points to the table entry that contains this address.
  - Usually, this is an address given to each interrupt in a microcontroller.



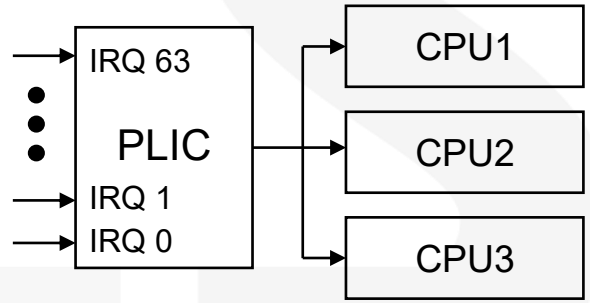
Interrupt Handling – Priorities



- **What happens when several interrupts are pending?**
  - The system has to decide which interrupt to handle.
  - This is often according to priority –
    the pending interrupt with the highest priority will be handled first.
  - In many architectures, the priority is hard-coded into the interrupt vector.



- **Is priority enough?**
  - Not really…
  - Interrupt masks may disable certain interrupts and enable others.
  - Some interrupts may be deemed non-maskable (NMI).
  - If an interrupt arrives while another interrupt is being handled, what should be done?
  - What if there are several cores? Several hardware threads (HARTs)?

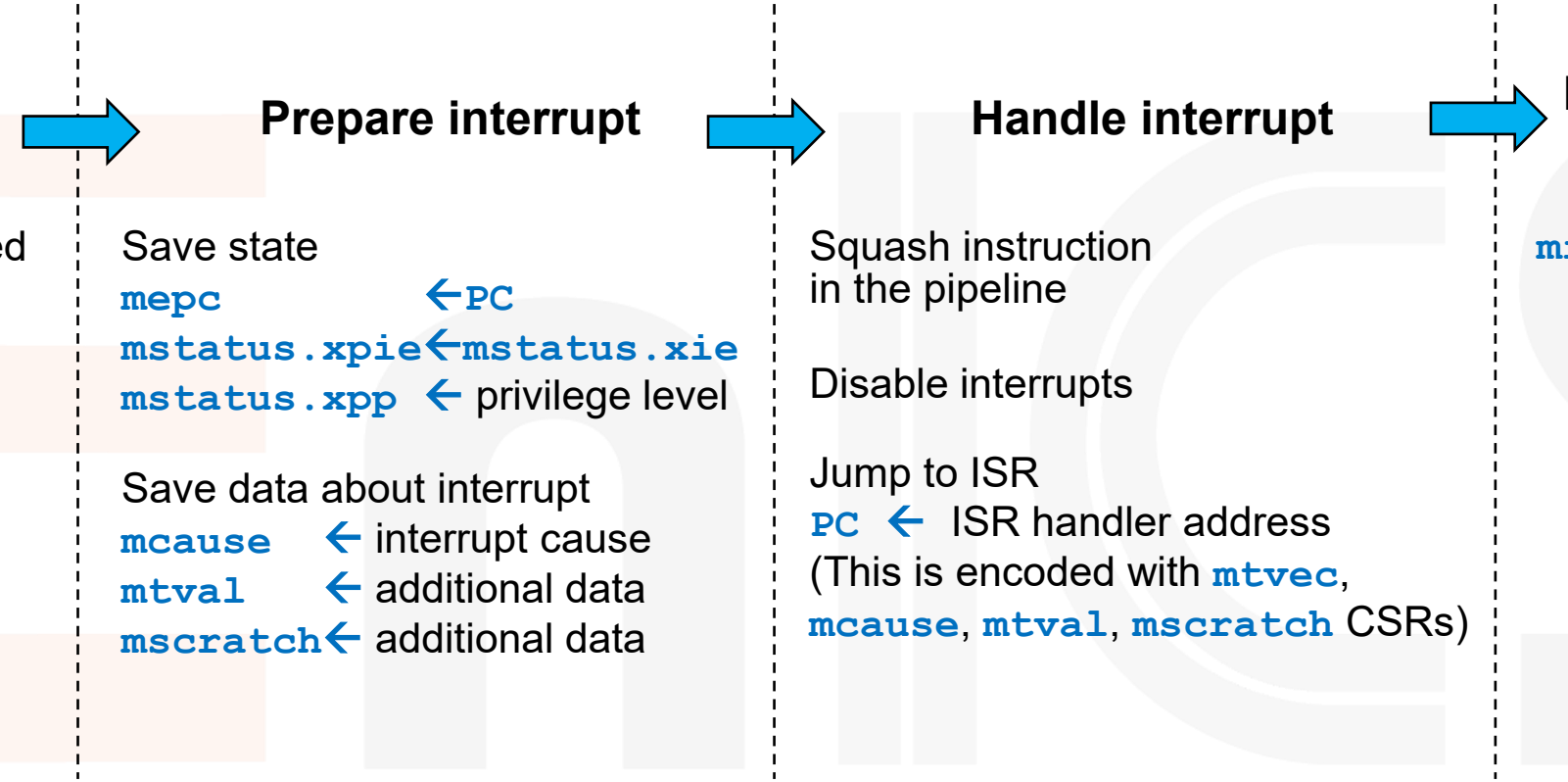


- **Therefore, an interrupt controller is usually used**
  - Usually implemented in hardware, especially for supporting nested interrupts.
  - RISC-V specifies a platform-level interrupt controller (PLIC) for multi-core/HART.

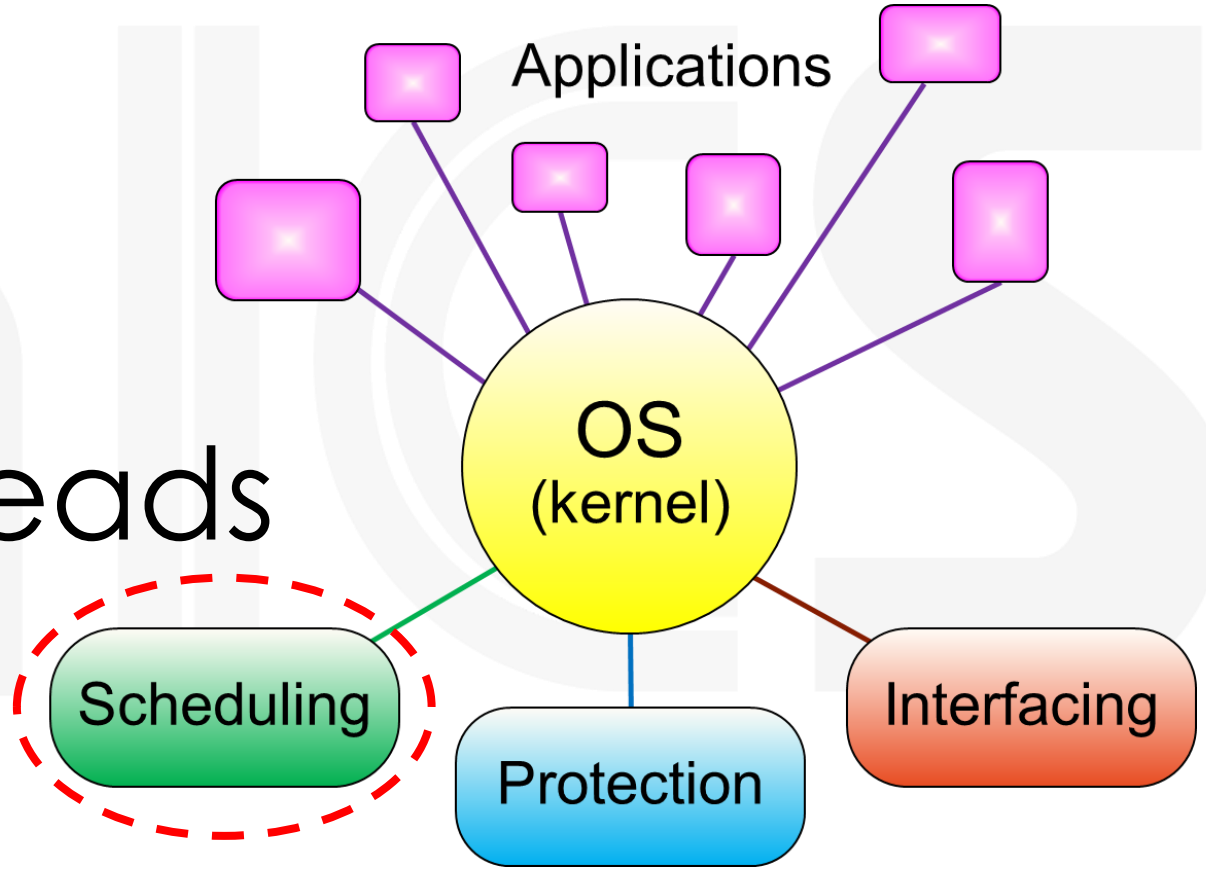
Interrupts in RISC-V

- **Interrupt handling in RISC-V centers around the `mstatus` CSR**
  - Includes bits for interrupt enable (`.xIE`) for each privilege level (`.MIE`, `.SIE`, `.UIE`)
  - Includes bits to store previous state to support nested interrupts:
    - Previous interrupt enable: `xPIE` (`.MPIE`, `.SPIE`, `.UPIE`)
    - Previous privilege level: `xPP` (`.SPP`, `.MPP`)
  - Return from interrupt instructions (`mret`, `sret`) restore state upon return.
- **Pending interrupts are listed in the `mip` CSR.**
  - Separated into external (`.meip`), software (`.msip`), and timer (`.mtip`) bits.
  - Corresponding enable bits (`.meie`, `.msie`, `.mtie`) in the **`mei`** CSR.
- **Additional CSRs store information about the interrupt**
  - **`mcause`**: cause of the interrupt, **`mtvec`**: address of the interrupt handler
  - **`mepc`**: address of interrupted instruction, **`mtval`**, **`mscratch`**: more info

Interrupts in RISC-V



Scheduling:
Processes and Threads

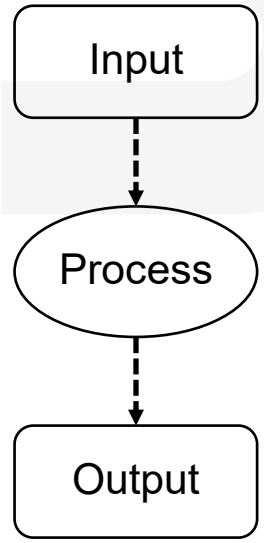


Emerging Nanoscaled
Integrated Circuits and Systems Labs
The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Concurrency

- **Many things are going on in an Operating System**
  - Applications, interrupts, background tasks, maintenance…
- **However, the CPU can only run one program at a time**
- **Solution #1: Cyclic Executive**
  - An endless loop that switches between tasks
  - This is really hard to write, maintain and optimize.
  - Can't handle interrupts!
- **Solution #2: Process Abstraction**
  - Decompose complex problems into simple ones (programs).
  - Create an instance of the program ("a process") and run it.
  - Timeshare between running processes through scheduling.
  - Each process feels like it has the entire CPU.

```
while true {
   do part of task 1;
   do part of task 2;
   do part of task 3;
}
```



Programs and Processes

Program

```
main () {
    ...
    foo();
    ...
    bar();
}
```

Process

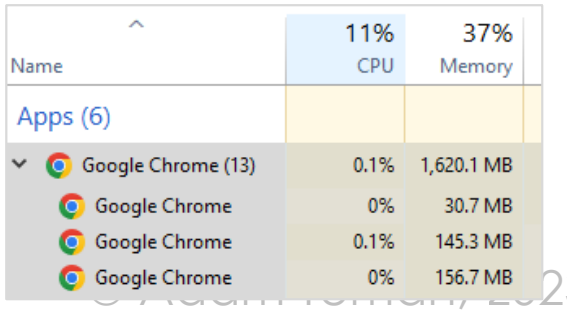
```
main () {
    ...
    foo();
    ...
    bar();
}
```

Registers

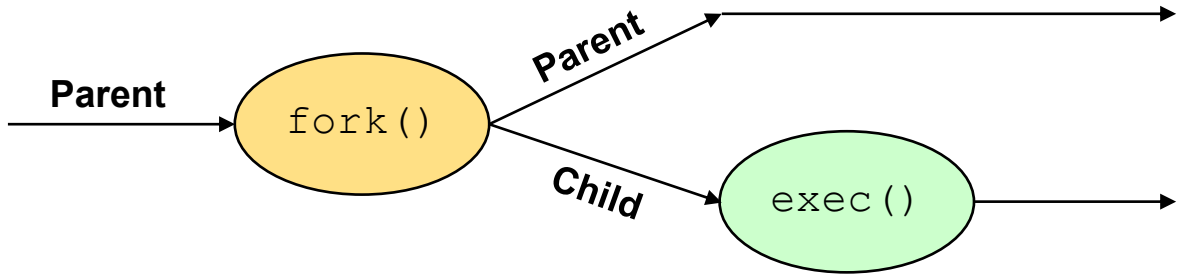
Address Space

Resources

- **A program is just the code**
  - A binary, including the text and static data.
  - Totally passive. Just bytes on a disk.
- **A process is an instance of a program in execution**
  - A copy of the program code (text and static data)
  - The processor state (PC, registers, memory)
  - Resources (file pointers, etc.)
- **A program becomes a process when an executable is loaded into memory**
  - There may be many processes running copies of the same program.
    - e.g., each tab of Google Chrome is a separate process
- **However, there is only one set of resources**
  - Process must be *scheduled* and must ensure *protection*.



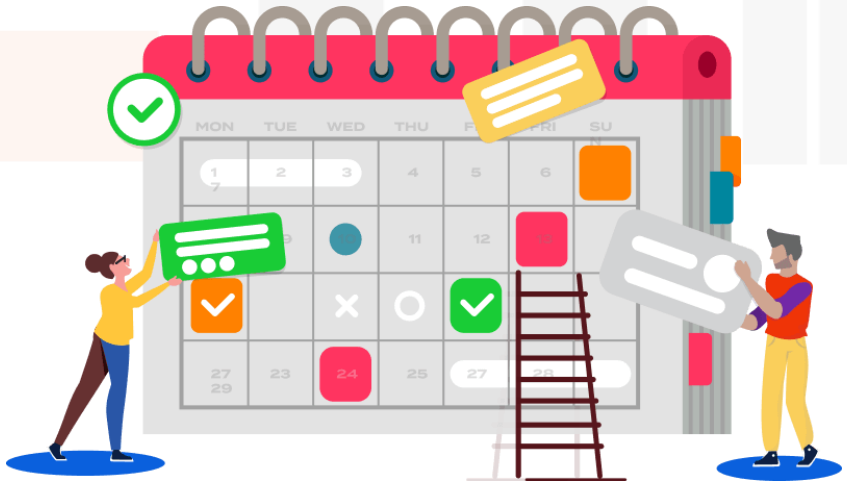
Process API



- **How is a process created?**
  - In Posix systems, the most common way to create a process is to `fork` it.
  - `fork()` is a system call that creates a copy of process.
  - The new child process is an almost exact copy.
- **Why do we want a copy of the parent process?**
  - Actually… we don't.
  - The `fork()` call is followed by an `exec()` call, which runs a new program.
- **Why create a copy of the parent process?**
  - Very good question… without a really good answer.
  - The short answer… because this API just works ("Lampson's Law")
  - The better answer… because it let's us run things on a child process before executing it. This is what enables "pipes" (|) and "redirects" (>) in Unix.

Scheduling

- **A CPU can only run one process at a time**
  - How does the OS choose which process will be executing?
  - When should the OS change the currently running process?
  - How can the OS efficiently utilize the resources?
- **The decision of which task to run and when is called Scheduling**
  - A scheduling policy tries to optimize performance and fairness.



source: tmetric.com

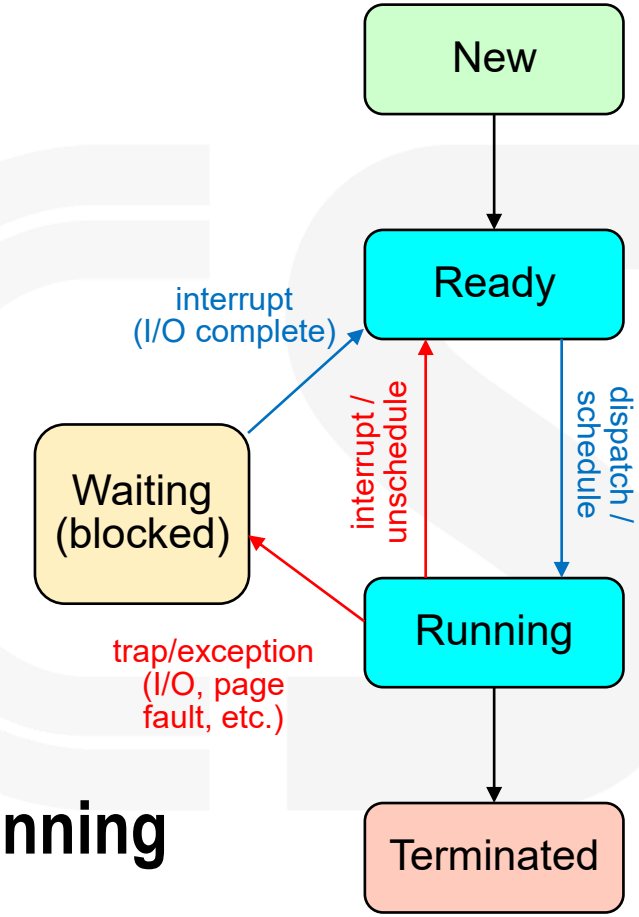
Scheduling Policies

- **The scheduling policy dictates which process will be run next.**
- **This may depend on what we want to achieve…**
  - Maximum throughput, minimize response time, fairness, etc.
- **Therefore, various scheduling policies exist:**
  - First-come First-served (FCFS)
  - Shortest Job First (SJF)
  - Shortest Remaining Processing Time First (SRPT)
  - Round Robin (RR)
  - Priority

- **The most common approach today is Multi-level Feedback Queues (MLFQ)**
  - Provide several queues with different priorities
  - Apply RR within the queues. Move between priorities based on aging.

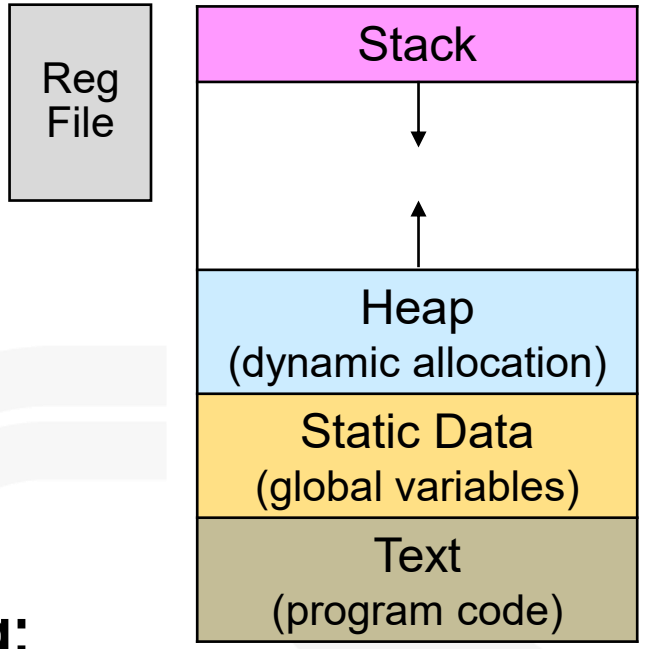
Queue-based preemptive scheduling

- **Some policies feature queues of processes waiting to run**
  - Queues for each device, timers, interrupts, messages…
- **Each process has an execution state within the queue:**
  - Ready: Waiting to be assigned to a CPU
    - Could run, but another process has the CPU
  - Running: Executing on a CPU
    - It's the process that currently controls the CPU
  - Waiting (a.k.a., "blocked"): waiting for an event, e.g., I/O completion, or a message from another process
- **The OS periodically decides to change which process is running**
  - This is known as "preemptive scheduling":
  - *"Preempt": to acquire or appropriate before someone else.* (*dictionary.com*)
- **Preemptive scheduling requires a mechanism called a "Context Switch"**



Context Switching and the PCB

- **Context Switching is the mechanism to change which process is currently running on the CPU**
  - Invoke the kernel, store the state of the running process, load the new process onto the CPU.
- **Everything necessary for loading/storing a process is stored in a data structure called a process control block (PCB), including:**
  - The code for running the program ("text")
  - The CPU registers (e.g., PC, SP, etc.)
  - The address space (static data, stack, heap, page tables)
  - I/O Status (open files, I/O devices allocated to the process)
- **And additional information used by the OS.**
  - Process identification (PID)
  - Process execution state (ready, running, waiting, …)
  - Scheduling information (priorities, queue pointers, …)
  - Accounting data (e.g., CPU used, running time, …)



PCB

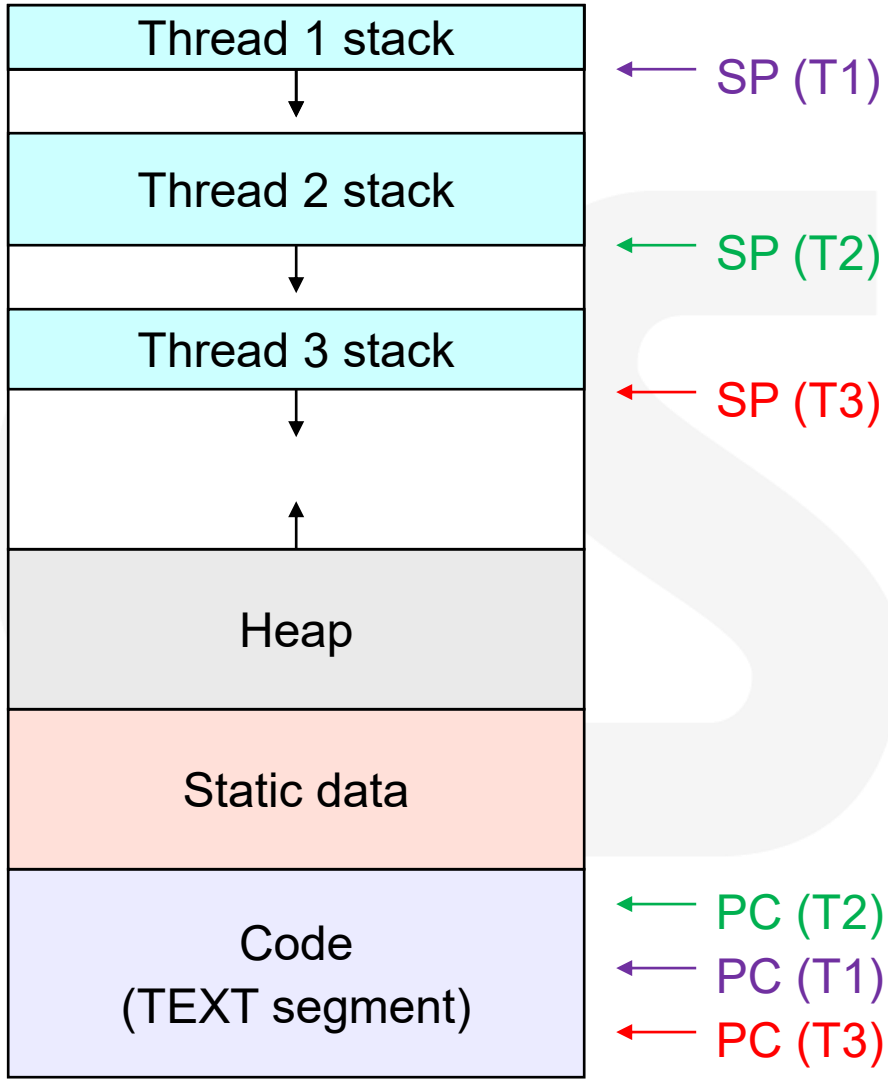
pid
execution state
program counter
stack pointer
registers
memory limits
list of open files
…

Context Switch: Back to the Teman Analogy

- **I need to prepare this lecture…**
  - Open up PowerPoint… Start working on the slides
- **WhatsApp message**
  - It's the wife… She's at the store… need anything?
  - Save work… go to check in the kitchen…
  - Back to work… where was I?
- **Phone rings…**
  - It's Alex… need to send that email – now!
  - Save work… open mail client… send mail
  - Okay, re-open PowerPoint… continue editing
- **Alarm clock rings…**
  - Time to pick up kids…
  - Save work… put computer to sleep… find car keys…

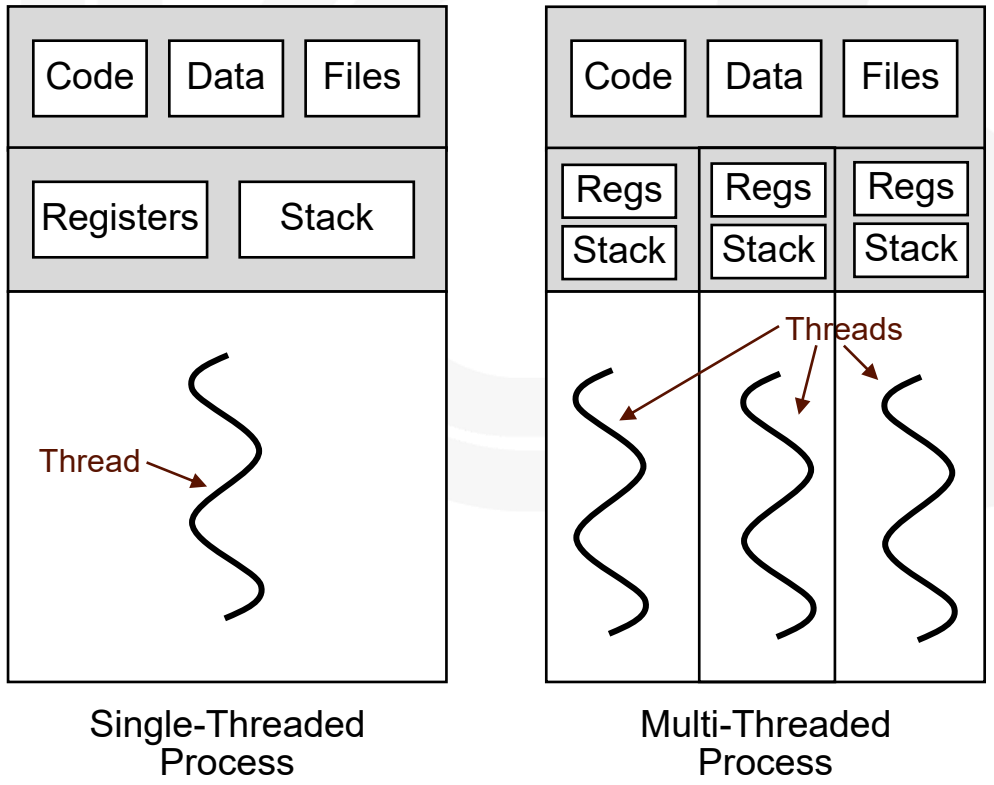
The Thread

- **Processes do not share resources well**
  - Context switching is expensive
    - e.g., saving and restoring a page table
  - Sharing data between processes is complicated
- **Idea: Separate *concurrency* from *protection***
  - Create a *lightweight process* → a "**Thread**"
  - A process can have several threads
- **A process has a single address space**
  - Code and Data sections
  - OS resources (open files, signals)
- **Each thread has its own PC, CPU registers, stack**
  - There is no protection between threads



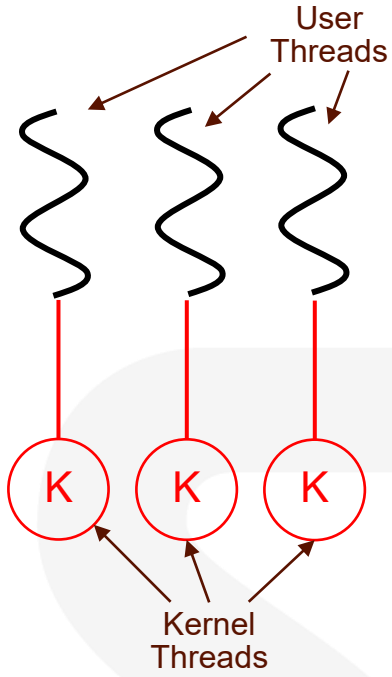
The Thread (cont.)

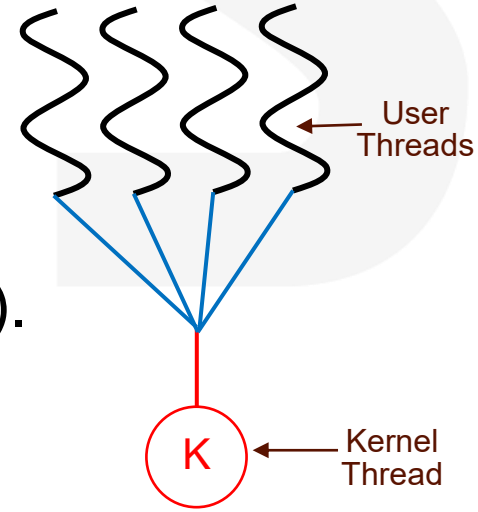
- **Threads are concurrent executions sharing an address space (and some OS resources)**
- **Threads are *cheaper* than processes:**
  - Creating threads is cheap
  - Context switching threads is cheap
  - Sharing data between threads is cheap
- **In a multiple threaded task, while one thread is blocked and waiting, a second thread in the same process can run.**
  - Higher throughput
  - Improved performance



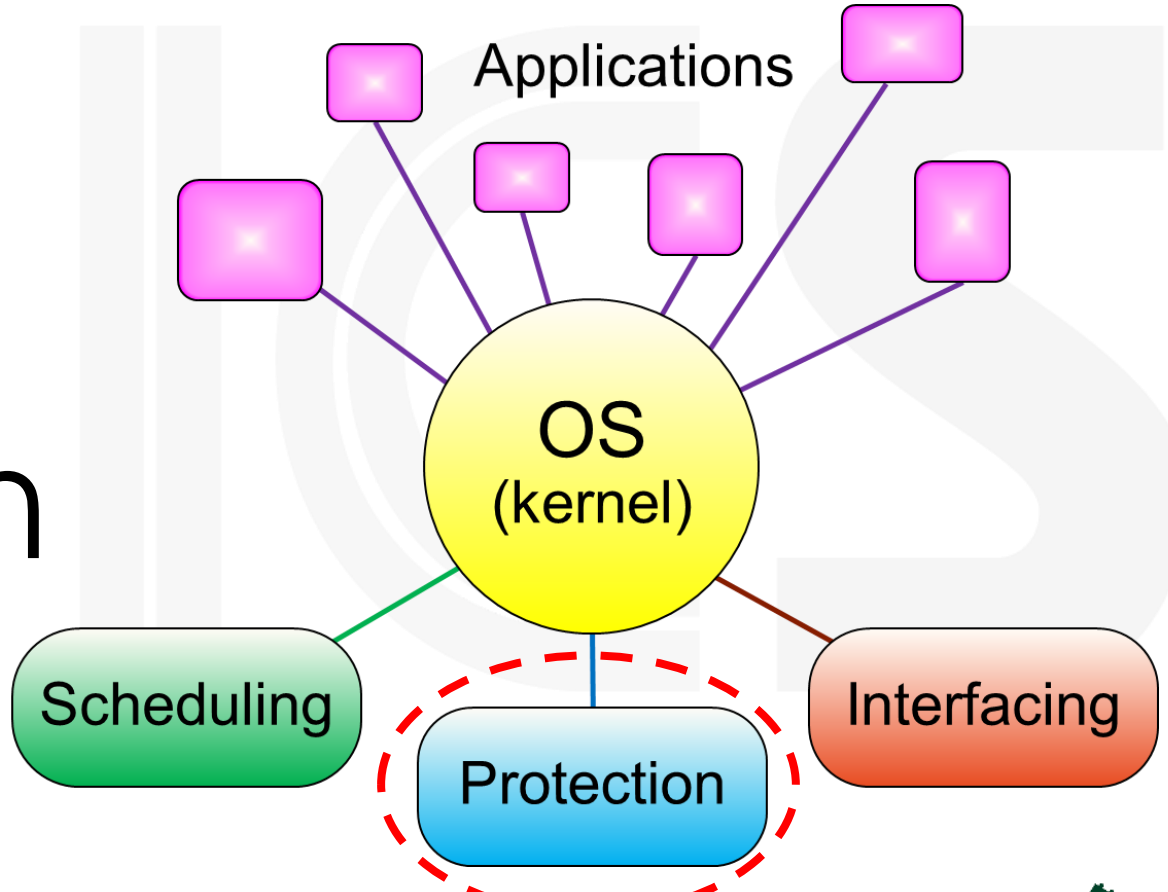
Kernel vs. User-Level Threads

- **How are threads managed?**
  - How are they created? Who allocates memory for them?
  - How are they context switched?
- **In a Kernel Thread, the OS manages the threads**
  - Each thread has a kernel thread associated with it.
  - All thread operations are implemented in the kernel.
  - Thread operations are cheaper than process ones, but still costly.
- **Alternatively, a process can manage its threads in user space**
  - User-level threads are much cheaper to manage (10-100x faster!).
  - Can be limited due to I/O operations going through the OS.
- **Hardware threads (HARTs) provide very efficient context switching**
  - Also called simultaneous multi-threading (SMT) or hyperthreading (Intel)





Protection and Synchronization



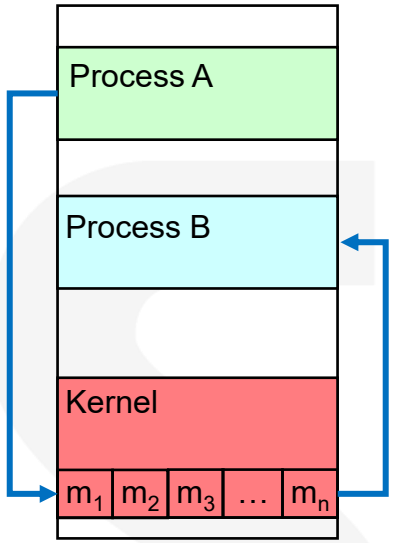
Emerging Nanoscaled Integrated Circuits and Systems Labs
The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

Protection

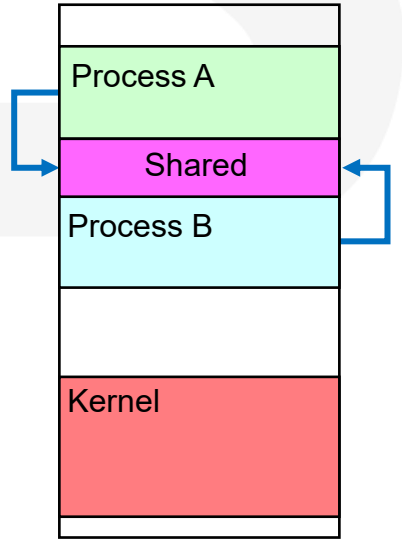
- **Protection is required for:**
  - To prevent errant or malicious misuse of the system by users or programs
  - To ensure that shared resources are used according to system policies
  - To limit the damage caused by errant programs
- **The basic means of protection is to provide each process with its own separate address space**
  - Only the kernel (supervisor) has access to the full physical address space
  - The process can only address the space defined in its PCB
  - Virtual memory and page tables help abstract away the physical memory from the program
- **But how can we share data between processes and maintain protection?**
  - In other words, how do we enable inter-process communication (IPC)?

Inter-Process Communication Models

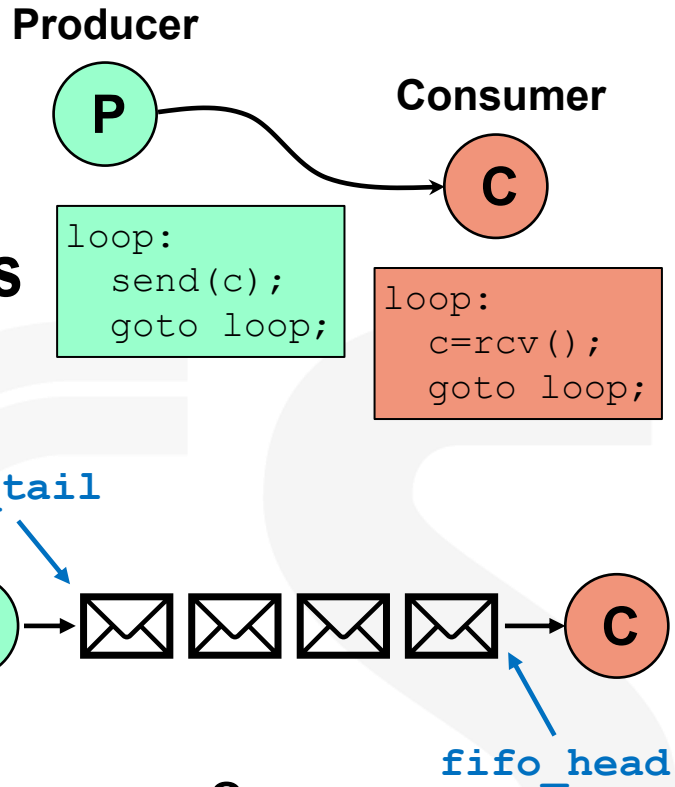
- **Option #1: The Message Passing model**
  - **Let the Kernel handle it!**
  - `send()` – store data on a queue managed by the kernel.
  - `receive()` – read data from the queue through the kernel.
  - But this suffers from a lot of overhead due to frequent system calls.



- **Option #2: The Shared Memory model**
  - **Let several processes access the same memory space.**
  - No kernel intervention is required.
  - But how can we ensure protection on the shared memory?



Can we use a queue?



- **In a Producer-Consumer model, we can restrict write access**
  - Only the producer can write to the queue.
  - For example, use a First-in First-out (FIFO) buffer.
  - This enables passing messages between tasks without accidentally corrupting each other's messages.
- **This works for two processes**
  - But what if there are several producers and/or several consumers?
  - They need to share the FIFO pointers.
  - What will happen if both try to update the pointer at the same time?

Producer 1

```
...
//Producer 1 pushes to FIFO
*fifo_tail = producer1_data;
fifo_tail++;
...
```

What happens if we context switch here?

Producer 2

```
...
//Producer 2 pushes to FIFO
*fifo_tail = producer2_data;
fifo_tail++;
...
```

The Critical Section

- **Simultaneous access to a shared variable can result in a race condition.**
  - For example, the wife and I both withdraw $50 from the ATM.
  - Preemptive scheduling could lead to the following course of action:

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  spit out cash;
}
```

Current balance: $50

```
balance = get_balance(account);
balance -= amount;
```

$50

Context Switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
spit out cash;
```

$50

Context Switch

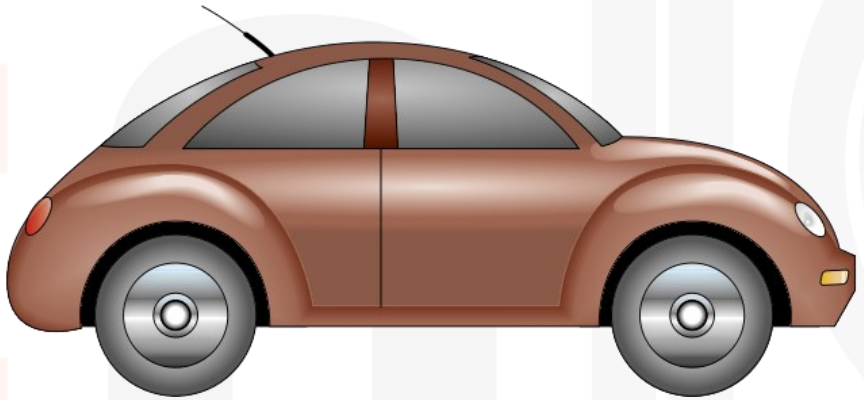
```
put_balance(account, balance);
spit out cash;
```

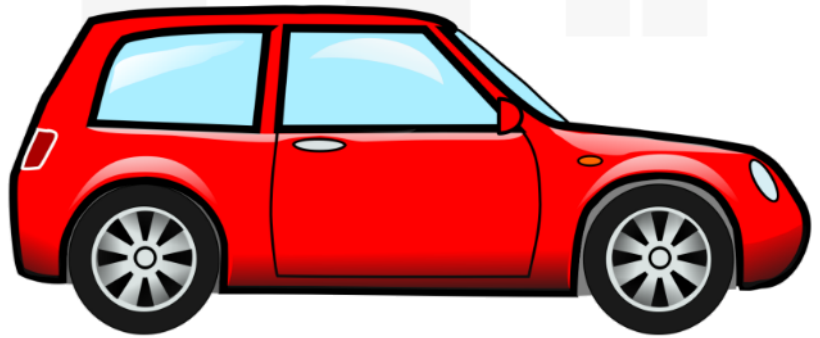
- **The code block where a shared variable is accessed is called "The Critical Section"**

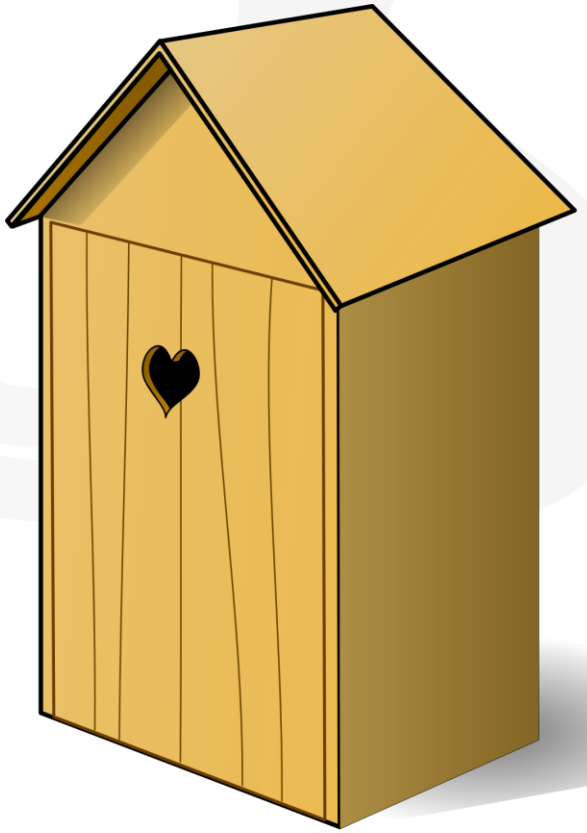
Gas Station Analogy: Critical Section

- **A gas station on the highway has a public bathroom.**
  - Two cars make a pit stop.
  - Who will get there first?

- **The bathroom is in the critical section!**

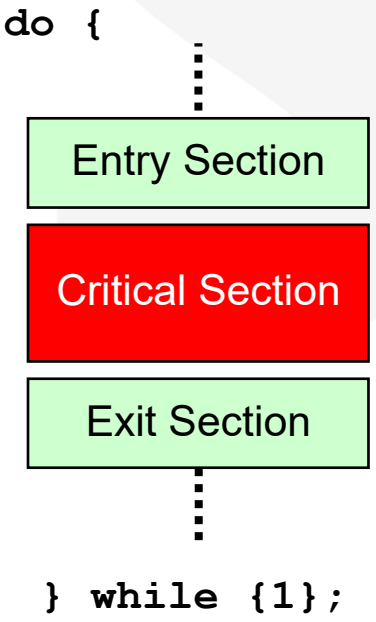






The Critical Section Problem

- **A Race Condition occurs if:**
  - Several processes/threads manipulate a shared resource concurrently, and
  - The outcome depends on the particular order in which the access takes place.
- **Synchronization around critical sections is needed to prevent race conditions.**
- **The criteria to solve the critical section problem are:**
  - **Mutual Exclusion**: At most one thread can be inside its critical section at any given time.
  - **Progress**: A thread outside its critical section, cannot prevent another thread from entering the critical section.
  - **Bounded Waiting**: A thread waiting on its critical section will eventually be allowed to enter it.



Mutual Exclusion (Mutex)

- **A Mutex is a primitive that enables synchronized access to a shared resource (i.e., variable).**
- **Define a lock:**
  - `acquire()`: obtain the right to enter the critical section
  - `release()`: give up the right to be in the critical section
- **Often accessed with a "Spinlock"**
  - Busy-wait on the lock until it is free.
- **Problem: Spinlocks have critical sections, too!**
  - `acquire()`/`release()` must be atomic
  - Atomic == executed without interruption
- **Need help from the hardware**
  - *Atomic instructions*, e.g., test-and-set
  - or *disable interrupts* to prevent context switches

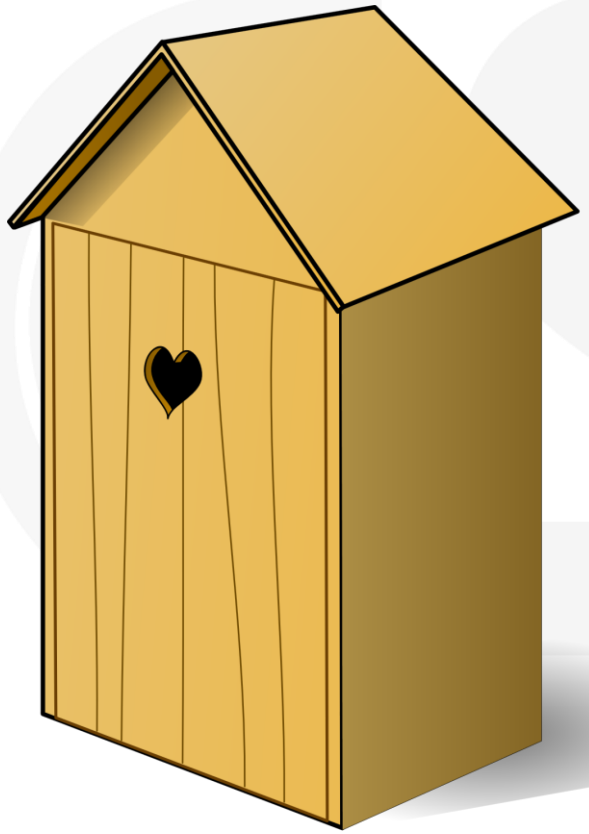
What happens if we context switch here?

```
struct lock {
  int held = 0; }

void acquire(lock) {
   while (lock->held);
   lock->held = 1; }

void release(lock) {
  lock->held = 0; }
```

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old; }

struct lock {
  int held = 0; }

void acquire(lock) {
   while(test_and_set(&lock->held));
}

void release(lock) {
  lock->held = 0; }
```

Gas Station Analogy: Mutex

- **Gas station has one bathroom and one key.**
  - First customer acquires key.
  - Second customer waits.
  - Until first customer releases the key.
  - Now second customer can acquire the k



Let's go back to our ATM

- We'll protect our critical section with a lock.

Current balance: $0

```
int withdraw(account, amount) {
  acquire(lock);
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  spit out cash;
}
```

Critical Section

```
acquire(lock)
balance = get_balance(account);
balance -= amount;
```

$50

```
acquire(lock)
```

```
put_balance(account, balance);
release(lock);
```

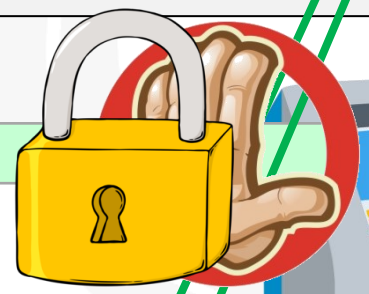
Context Switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
spit out cash;
```

$50

Context Switch

```
spit out cash;
```

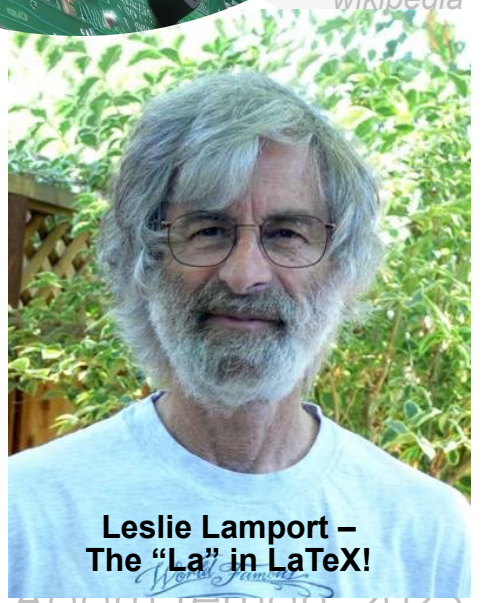


Implementing Synchronization

Image generated with Bing Image Creator

Source: wikipedia

- **Synchronization is a true "black hole" that should not be approached by the faint-hearted.**
  - Carefully written "synchronization libraries" enable programmers to implement mutual exclusion and write race-free code.
- **Mutual exclusion can be achieved through software only***
  - For example, Lamport's Bakery Algorithm (1974)
  - But these algorithms require many processor cycles to implement.
- **Therefore, locks are usually implemented using "atomic operations"**
  - Sequences that are guaranteed to complete within a single indivisible operation.
- **OS classes just "assume" the existence of such instructions…**
  - But how do we actually implement them in hardware?



Leslie Lamport – The "La" in LaTeX!

* *This will usually still require hardware barriers for multi-processors*

Atomic Instructions

Three main types of atomic instructions are found in synchronization libraries:

- **Atomic Exchange:**
  - Swap the content of a register and a memory location.
  - If the lock was free – it returns a '0', otherwise we spin.
  - But this means we continually write to memory, which is very bad…

Acquiring a lock with Atomic Exchange

```
R1=1;
while (R1==1)
    EXCH R1, lock;
```

- **Test and Set:**
  - Read the lock, and if it is free, acquire it.
  - No need to write to memory while spinning.
  - But requires load and store within single instruction.
  - Would lead to deep pipeline / performance hit.

Test and Set Command

```
if (*lock==0) {
    *lock=1;
    R1=0;
} else {
    R1=1;
}
```

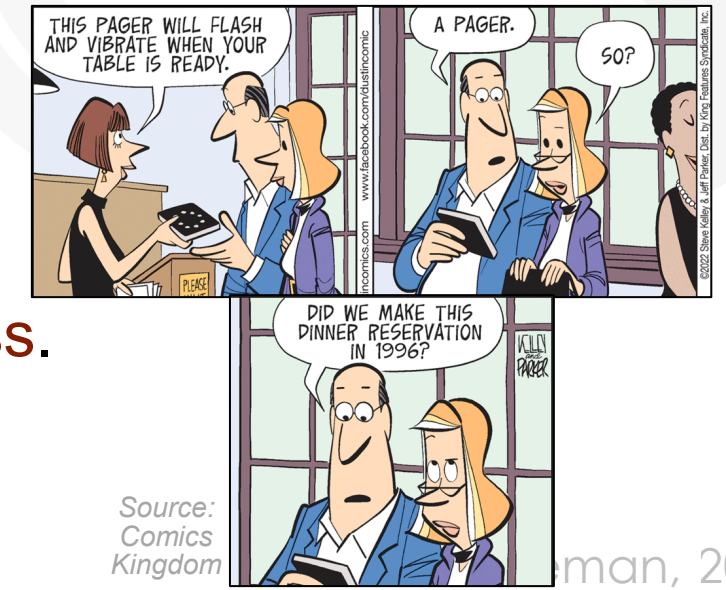
- **Load Linked / Store Conditional (LL/SC):**
  - Split the operation into two commands to simplify memory access.
  - Let's see how this works…

Load Linked / Store Conditional

- **The LL/SC approach creates an atomic operation with two instructions.**
- **Load Linked (or Load Reserved)**
  - Carry out regular load from memory to a register.
  - Create a "reservation" on the load.
- **Store Conditional**
  - Attempt to store a value to the memory location.
  - If the "reservation was broken", the store fails.
- **What "breaks a reservation"?**
  - A context switch (→breaks atomicity)
  - A store (by another processor) to the reserved address.
- **The reservation is held in a hidden register and can be accessed in parallel to the memory access.**

Implementing a lock with LL/LR

```
void lock (&lockvar) {
  trylock:
    MOV R1,1          // R1=1
    LL  R2, lockvar   // R2=*lockvar
    BNE R2, trylock   // spin if R2!=0
    SC  R1, lockvar   // try *lockvar=1
    BEQ R1, trylock   // spin if failed
```



Source: Comics Kingdom

Atomic Operations in RISC-V

- **RISC-V adopted the LL/SC approach**
  - Load-Reserved (`lr`):
    Reservation includes address of memory access
  - Store-Conditional (`sc`):
    Releases reservation and returns 0 if successful.

```
lr.w rd,(rs1)   # rd= Mem[rs1]
sc.w rd,rs2,(rs1) # Mem[rs1] = rs2
                  # rd = success ? 0:1
```

  - Further constraints (e.g., limiting number and type of instructions between the `lr`/`sc` instructions) guarantee that the sequence will eventually succeed.
  - RISC-V supports additional atomic instructions (e.g, `amoswap`, `amoadd`), but these can be implemented with `lr`/`sc` on simple processors.

Atomic Swap implemented with LL/LR

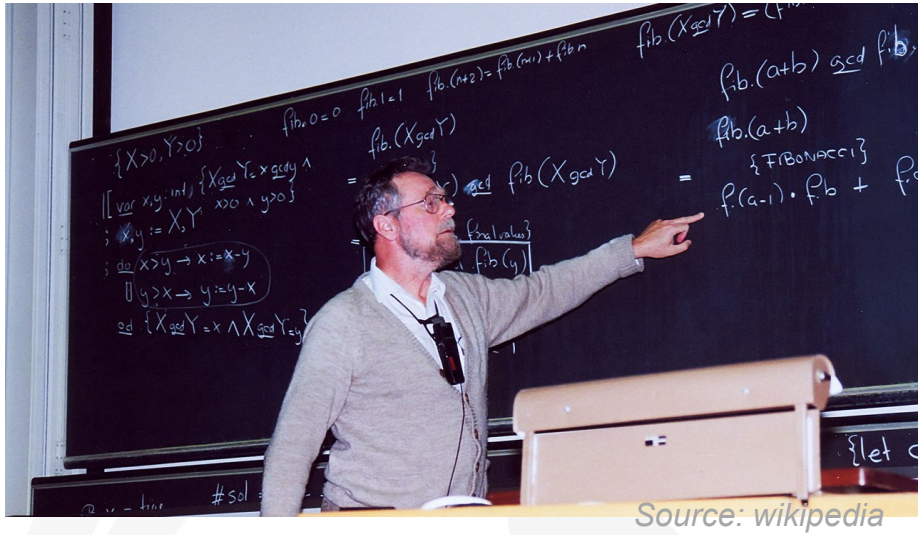
```
again:                      // swap x23 with (x20)
   lr.w x10,(x20)           // x10 = *lock
   sc.w x11,(x20),x23       // x11 = succeed/fail?
   bne  x11,x0,again        // spin if store failed
   addi x23,x10,0           // x23 = loaded value
```

Another approach: Disabling interrupts

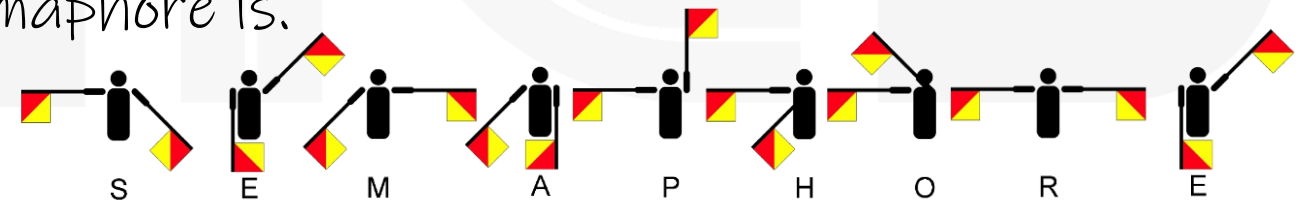
- **Disabling interrupts is a simple solution for mutual exclusion by ensuring there are no context switches while accessing the critical section.**
- **Problems with disabling interrupts:**
  - Only available to the kernel
    - Can't allow user-level to disable interrupts!
  - Insufficient on a multiprocessor
    - Each processor has its own interrupt mechanism
  - "Long" periods with interrupts disabled can wreak havoc with devices
- **Use disabling of interrupts to build higher-level synchronization constructs**


```
struct lock { }
void acquire(lock) {
   cli(); // disable interrupts
}
void release(lock) {
  sti();  // reenable interrupts
}
```

Final point… Semphores!


Source: wikipedia

- **I know, since the beginning of this lecture, you've been waiting for me to talk about semaphores…**
- *Teman's First Law of Semaphores:*
  - The **only thing** anyone ever remembers from their undergrad operating systems class is that they **learned** about semaphores.
- *Teman's Second Law of Semaphores:*
  - No one actually remembers **what** a semaphore is.



- *Teman's Third Law of Semaphores:*
  - The only thing anyone remembers about semaphores is that **Dijkstra** invented them and there were some **variables** that come from **Dutch**.

So, what is this Semaphore thing?

- **Basically, it's a shared, non-negative integer.**
- **You mean, it's a variable that can have values, such as 0, 1, 2, etc.?**
  - Yes… that's really all it is.
- **So, what's the big fuss? Why is it different from any other variable?**
  - Well, it has two operations that can be applied to it: `wait` and `signal`.
  - `wait(sem)` (P): Block until `sem>0`, then subtract 1 from `sem` and proceed
  - `signal(sem)` (V): Increment `sem`
  - Both of these operations must be atomic!
- **And how exactly does that help us?**
  - It allows one thread to signal to another.
  - Signaling is non-blocking (unlike acquiring a lock), which is very useful.

P is for "*Proberen*" ("test" or "try")
V is for "*Verhagen*" ("increase")

Signaling with Semaphores

- **Example: Semaphores for Precedence**
  - Two processes: A and B
  - **Goal**: statement A4 should complete before B3 begins

```
semaphore s=0
```

Process A	Process B
A1;	B1;
A2;	B2;
	wait(s);
A3;	B3;
A4;	B4;
signal(s);	
A5;	B5;

- **Example: Bounded Buffer with Semaphores**

SHARED MEMORY:

```
char buf[N];  // The buffer
int in=0, out=0;
semaphore chars=0;
```

PRODUCER:

```
send(char c) {
    buf[in]=c;
    in=(in+1)%N;
    signal(chars);
}
```

CONSUMER:

```
char rcv() {
    char c;
    wait(chars);
    c=buf[out];
    out=(out+1)%N;
    return c
}
```

* Insufficient for multiple producers and consumers. Requires mutex!

Source: Chris Terman, MIT 6.004

So, let's get something straight

- **Semaphores are not Mutexes!**
  - A Mutex (a lock) is used for mutual exclusion to protect a critical section.
  - A Semaphore is (remember those flags?) used to signal a process.

- **So why is everyone (*I mean EVERYONE*) so confused?**
  - Because someone, somewhere decided to confuse everyone by calling a Mutex a "binary semaphore".
  - Because you can implement a Mutex with a Semaphore.
  - And because certain systems (e.g., FreeRTOS) only provide semaphores.
- **And just to help you remember, lets see what happens if we use semaphores at our tactically situated gas station…**

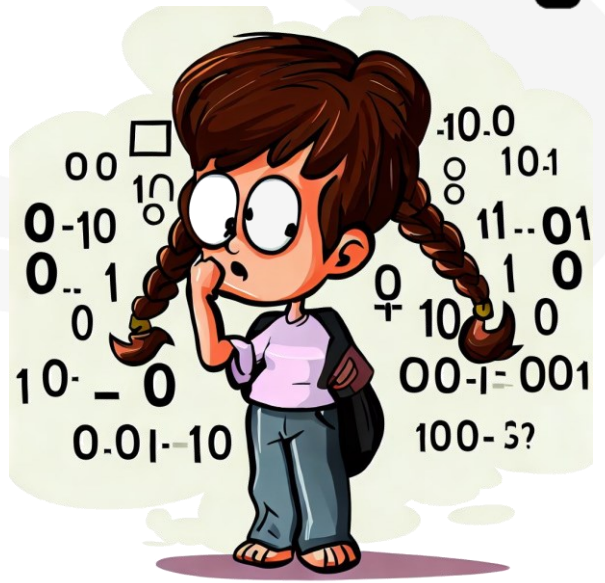
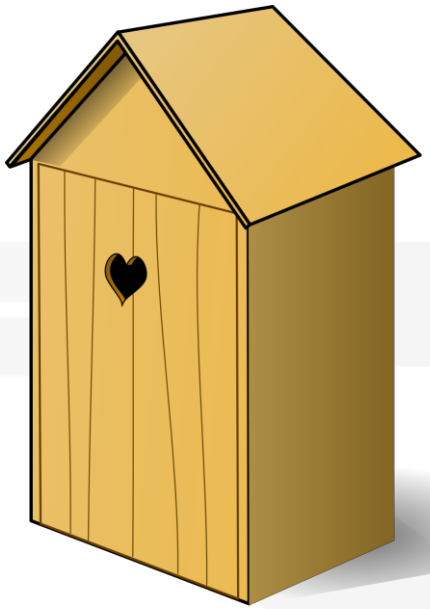
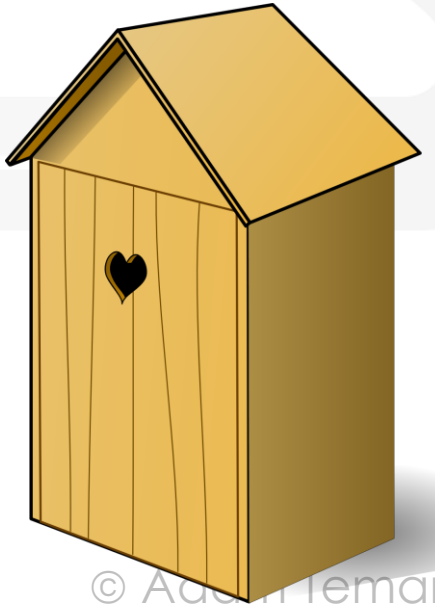


Image created with Bing Image Creator

Gas Station Analogy: Semaphore

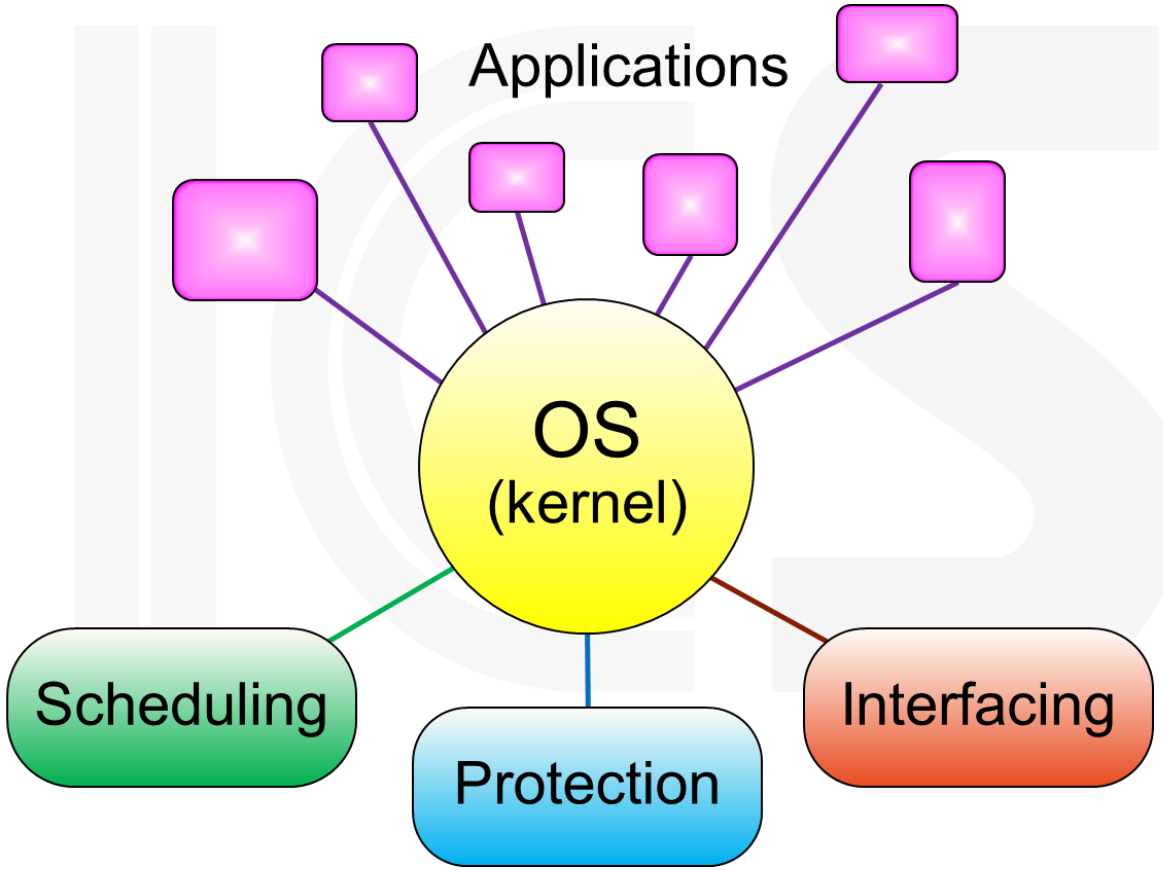
- **Gas station has several bathrooms and several keys.**
  - First customer acquires key.
  - Second customer acquires key too.
  - Third customer waits for others to finish.
  - First guy finishes and releases the key.





- **But…**
  - How does the waiting customer know <u>which bathroom</u> is occupied?
- **Remember**
  - Semaphores should not be used for mutual exclusion!.

RTOS



Emerging Nanoscaled
Integrated Circuits and Systems Labs
The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Do we need OS for Embedded Systems?

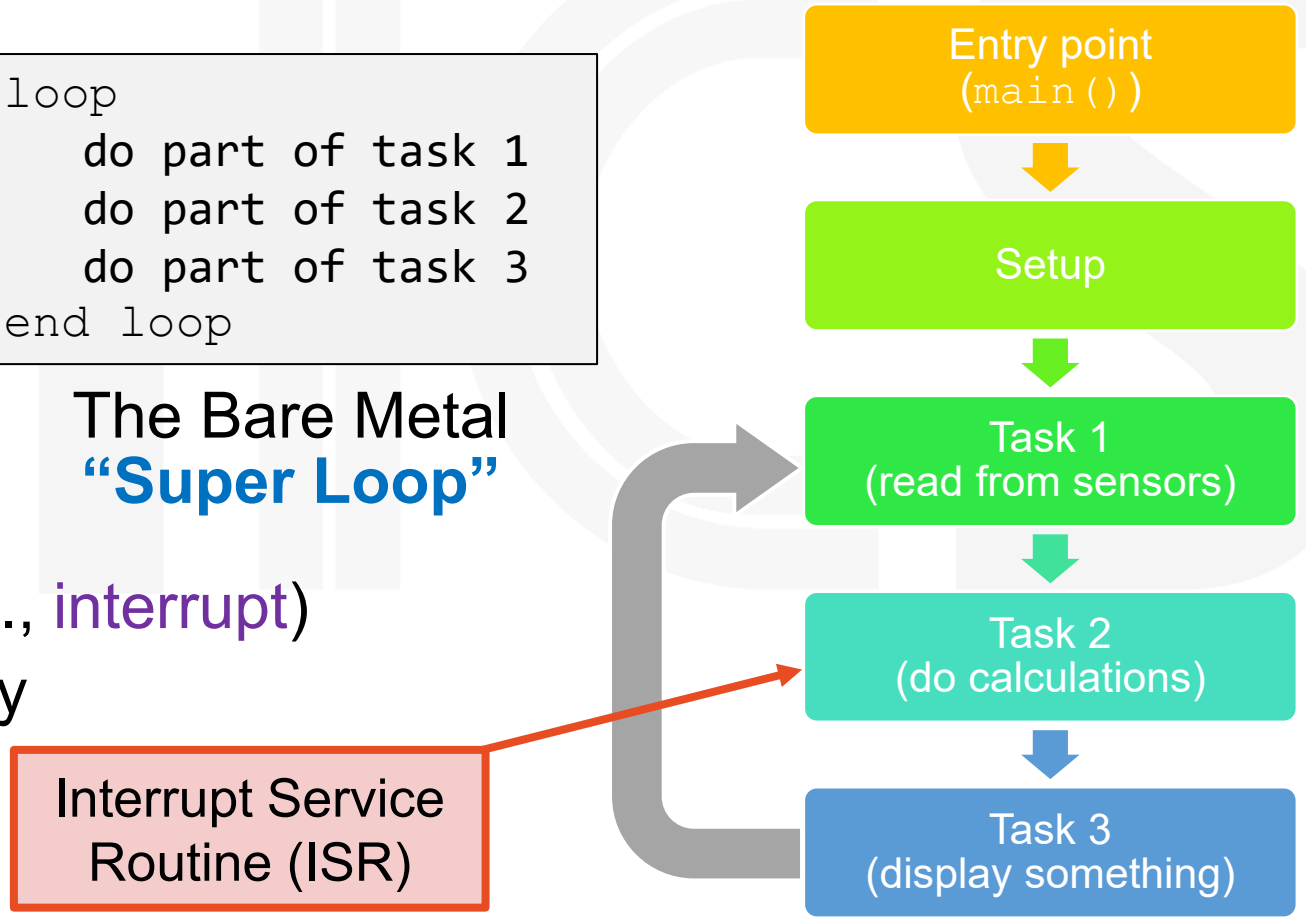
Not always!

- **For example, we want to multitask several programs**
  - The *cyclic executive* approach:

```
loop
   do part of task 1
   do part of task 2
   do part of task 3
end loop
```

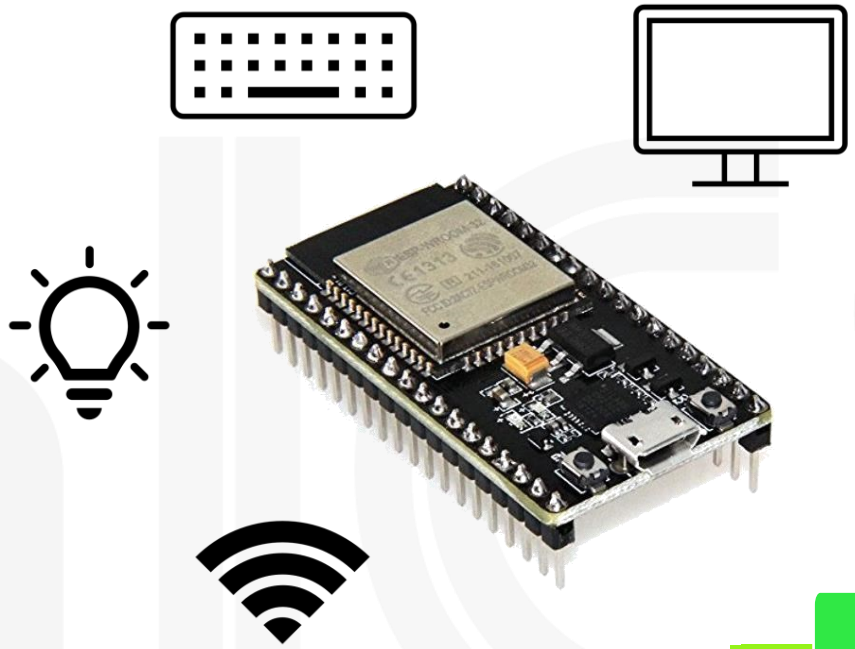
The Bare Metal
"Super Loop"

- **Pros**
  - Simple implementation
  - Low overhead
  - Very predictable
- **Cons**
  - Can't handle sporadic events (e.g., interrupt)
  - Code must be scheduled manually



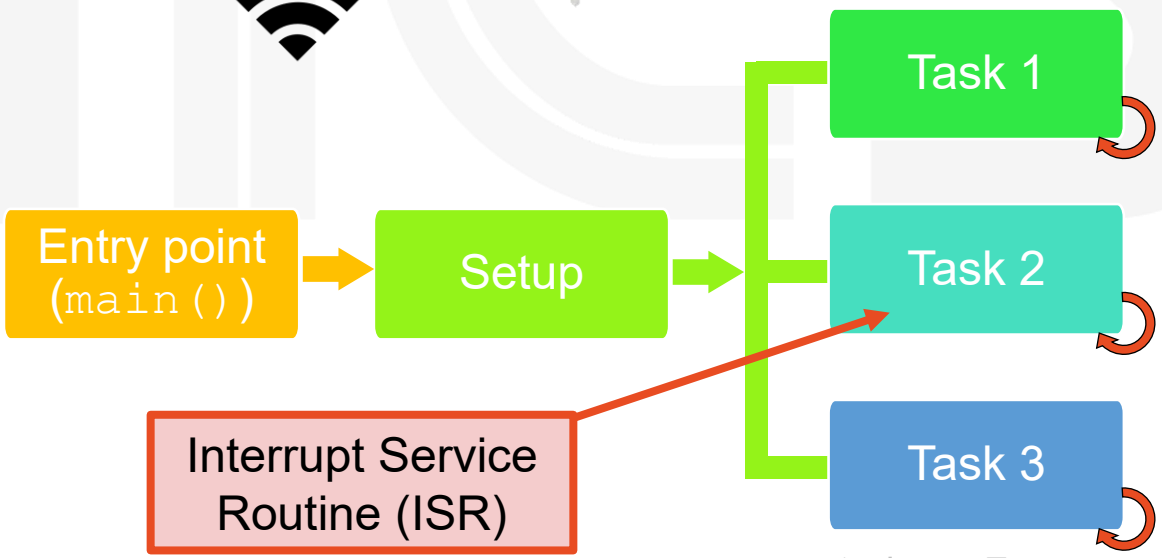
So, when do we need an OS?

- **If we have a more complex system…**

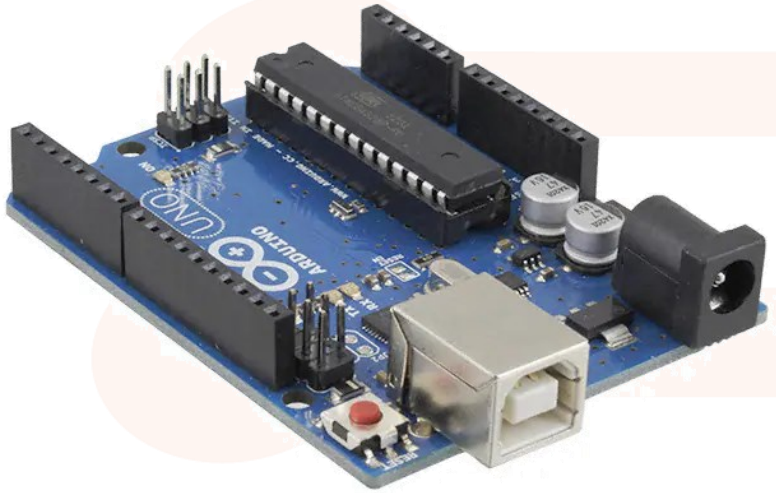


- **Embedded systems need an OS for:**
  - Convenience
  - Multitasking and threads
  - Real-time requirements

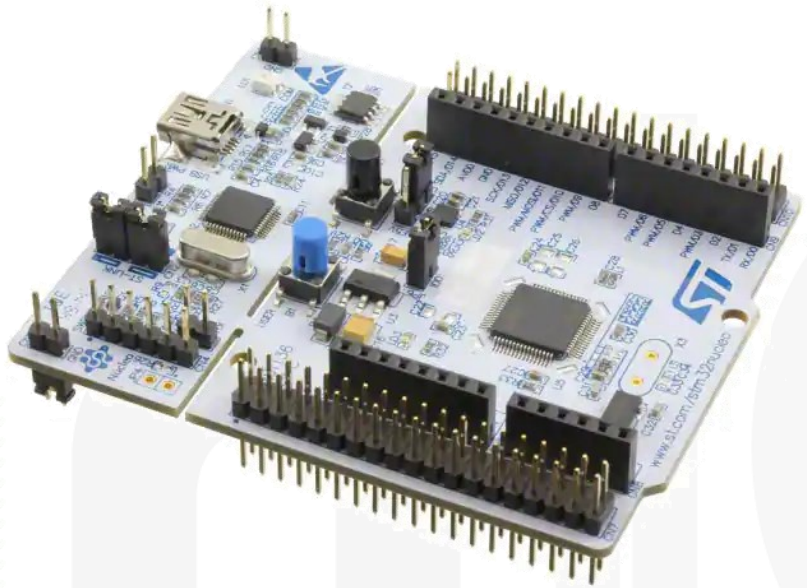
- **This is often provided by a Real Time Operating System (RTOS)**



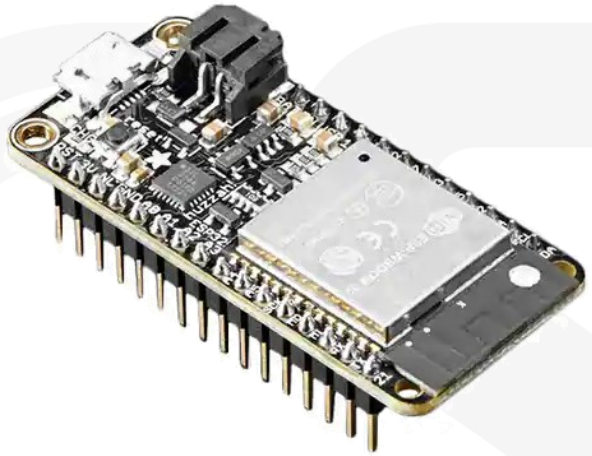
The choice is (also) hardware dependent



ATmega 328p
- 16 MHz
- 32 kB flash
- 2 kB RAM



STM32L476RG
- 80 MHz
- 1 MB flash
- 128 kB RAM



ESP-WROOM-32
- 240 MHz (dual core)
- 4 MB flash
- 520 kB RAM

Super Loop → RTOS

Source: Shawn Hymel, Digikey

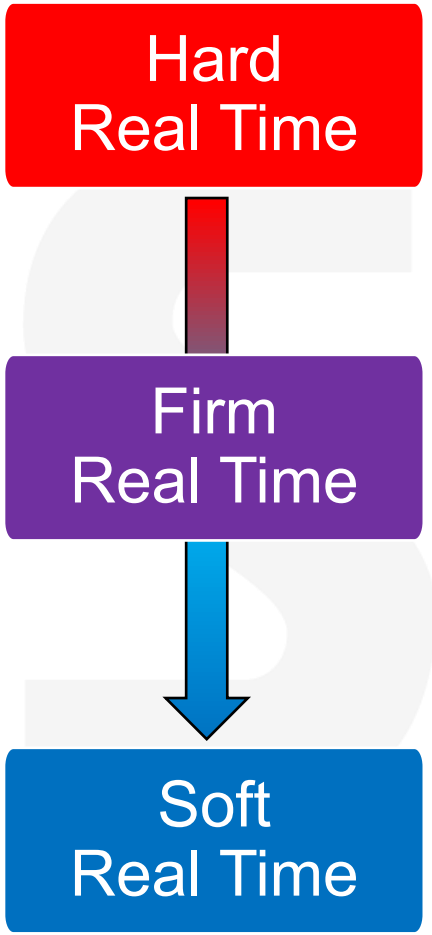
Why “Real Time”?

Image created with Bing Image Creator

- **A general-purpose operating system (GPOS) facilitates:**
  - Management of resources (CPU, memory, peripherals)
  - Multi-tasking (scheduling) and protection (synchronization)
  - User interface
- **A GPOS can (usually) schedule processes based on priority**
  - This enables differentiation between more critical and less critical tasks
- **However, timing cannot be guaranteed with standard scheduling policies**
  - A GPOS may switch from a critical task to another operation, completing the critical task only when the opportunity arises.
- **“Real Time” requirements mean that a task must be completed within a specific timeframe (usually within milliseconds).**

Implications of Real Time operation

- **In a real-time system, a process has the following requirements:**
  - **Initiation Time**:
    The time at the process goes from the waiting to the ready state
  - **Deadline**:
    The time at which the computation must be finished.
- **What happens when a process misses a deadline?**
  - *Catastrophic*, e.g., missed deadline in automotive control system
  - *Impede Quality*, e.g., missed deadline in telephone system
  - For safety-critical systems, compensatory measures will be taken, e.g., approximating data or switching to a special safety mode.
  - For non-safety-critical systems, the failure may be ignored or maybe bad data will not be propagated (i.e., silence in an audio stream)



So how do we achieve "Real Time"?

- **Easy…**

Priority-based Preemptive Scheduling!

- **Split and prioritize the application code**
  - Each task is designed separately
  - Each task is given a priority
  - Tasks generally wait for an event to trigger
  - The RTOS always runs the highest priority task that is available
- **RTOSs are preemptive**
  - Most RTOS have a time-based interrupt ("tick")
  - Every tick, the running task is interrupted (preempted), the scheduler is invoked and it decides which task should run next.

Easy to program

Prioritize the most important task

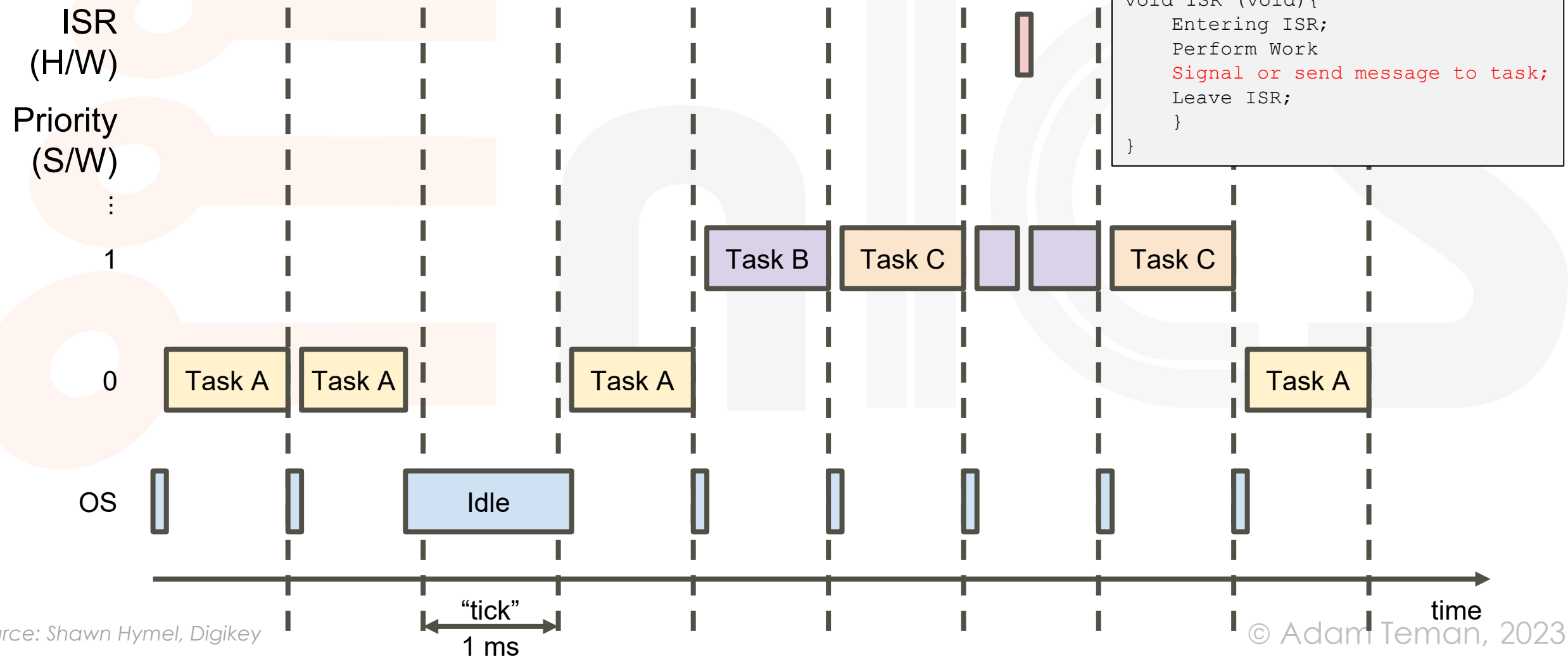
Event-triggered – avoids polling

Low-priority tasks don't affect the responsiveness of high priority tasks

Preemptive Scheduling

```
void EachTask (void){
    Task initialization;
    while (1) {
       Setup to wait for event;
       Wait for MY event to occur;
       Perform task operation;
    }
}
```

```
void ISR (void){
    Entering ISR;
    Perform Work
    Signal or send message to task;
    Leave ISR;
    }
}
```



Source: Shawn Hymel, Digikey

GPOS vs. RTOS

- **GPOS**
  - Complex for general-purpose – multiple applications on a system
  - Maximize resource usage, optimize average run time, ensure fairness
  - Non-predictable behavior
  - Pre-emptive or non-Pre-emptive
  - Coarse timer resolution (~10ms)
  - Large, protected virtual memory
  - Heavy and Large

- **RTOS**
  - Simple for embedded computing – single application on a system
  - Ensure timing, meet hard deadlines
  - Predictable
  - Pre-emptive and priority-based
  - Fine timer resolution (~1ms)
  - Small, flat memory hierarchy
  - Lightweight and small

Zephyr™

Drawbacks of Using an RTOS

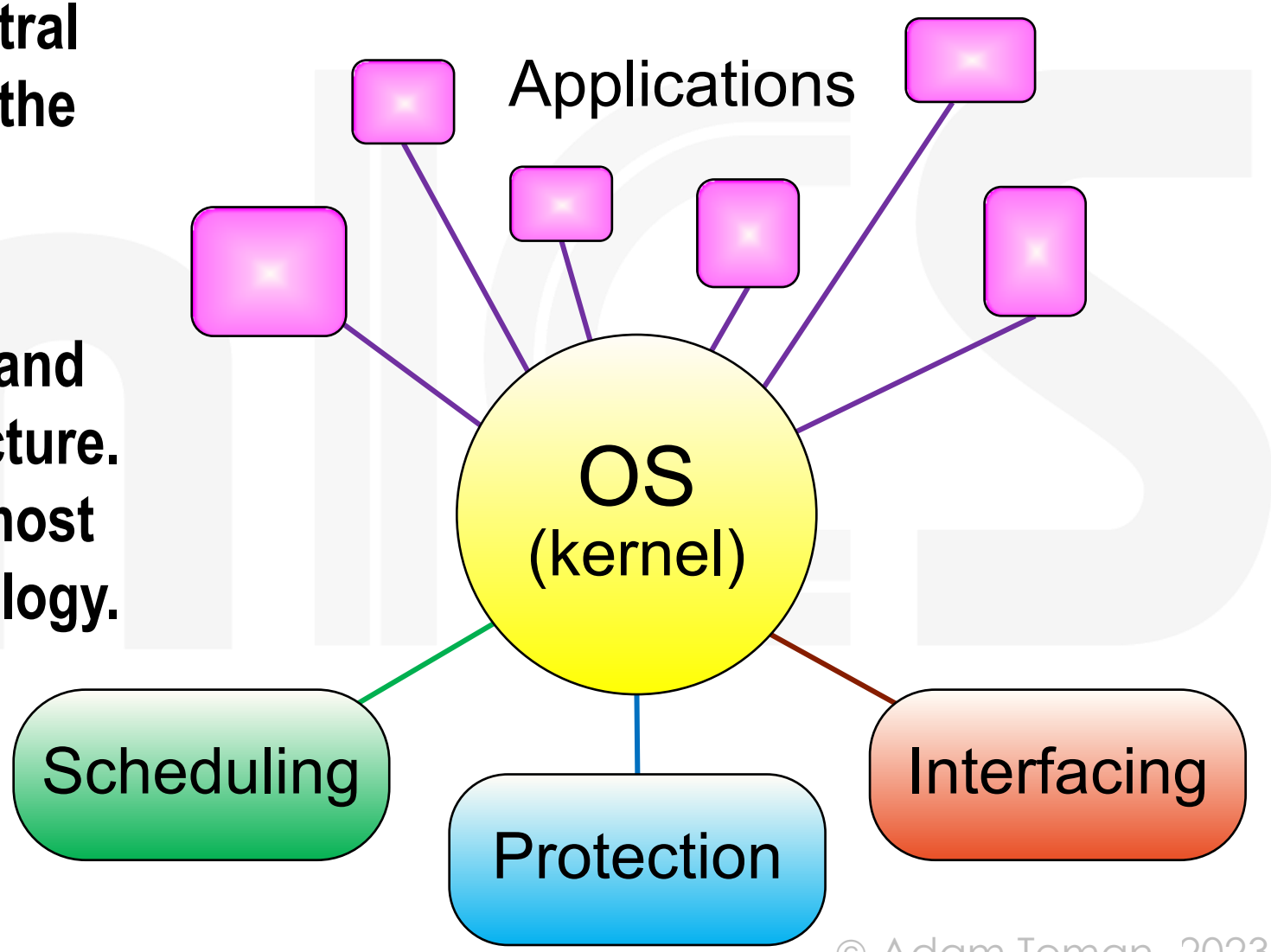
- **The RTOS itself is code and thus requires more Flash**
  - Typically, between 6-20K bytes
- **An RTOS requires extra RAM**
  - Each task requires its own stack
    - The size of each task depends on the application
  - Each task needs to be assigned a Task Control Block (TCB)
    - About 32 to 128 bytes of RAM
  - About 256 bytes for the RTOS variables
- **You have to assign task priorities**
  - Deciding on what priority to give tasks is not always trivial
- **The services provided by the RTOS consume CPU time**
  - Overhead is typically 2-10% of the CPU cycles
- **There is a learning curve associated with the RTOS you select**

Do You Need an RTOS?

- **Do you have some real-time requirements?**
- **Do you have independent tasks?**
  - User interface, control loops, communications, etc.
- **Do you have tasks that could starve other tasks?**
  - e.g. updating a graphics display, receiving an Ethernet frame, encryption, etc.
- **Do you have multiple programmers working on different parts of your project?**
- **Is portability and reuse important?**
- **Does your product need additional middleware components?**
  - TCP/IP stack, USB stack, GUI, File System, Bluetooth, etc.
- **Do you have enough RAM to support multiple tasks?**

Summary

- The operating system is the central piece of software that connects the programmer to the hardware.
- Operating system practicalities and research go well beyond this lecture. But we've learned some of the most important concepts and terminology.



Main References

- Gribble, Lazowska, Levy, Zahorjan, CSE 451, University of Washington
- Patterson, Hennessy “Computer Organization and Design – The RISC-V Edition”
- Berkeley CS-61C
- Null, Lobur, “The Essentials of Computer Organization and Architecture”
- Silberschatz, Gagne, Galvin, "Operating System Concepts, Ninth Edition"
- Venkatasubramanian, UC Irvine ICS 143
- Waterman, Asanovic, “The RISC-V Instruction Set Manual”
- Marilyn Wolf, “Computers as Components”
- UMASS CS-377
- Silicon Labs “Everything You Need to Know about RTOSs in 30 Minutes”
- Shawn Hymel, DigiKey “Introduction to RTOS”