Microprocessors

Fall 2020

12. Real Time Operating Systems

Tentative Weekly Schedule

- Week x1 - Introduction to Course
- Week x2 - Architecture
- Week x3 - Assembly Language Introduction
- Week x4 - Assembly Language Usage, Memory and Faults
- Week x5 - Embedded C and Toolchain
- Week x6 - Exceptions and Interrupts
- Week x7 - GPIO, External Interrupts and Timers
- Week x8 - Timers
- Week x9 - Serial Communications I
- Week xA - Serial Communications II
- Week xB - Analog Interfacing
- **Week xC - Real Time Operating Systems**
- Week xD- DMA
- Week xE - Wireless Communications

Real-Time Systems

Real-time systems are characterized by the **consequences** that will happen if **logical** as well as **timing correctness** properties of the system are not met.

There are three types of real-time systems:

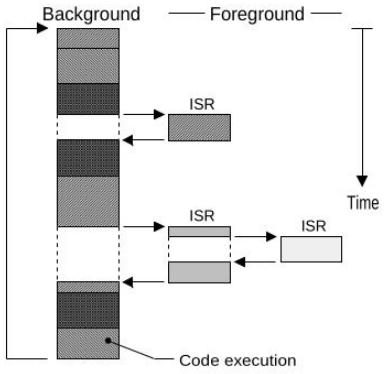
- **Soft real-time systems** - where tasks are performed by the system as fast as possible, but the tasks don't have to finish by specific times
- **Firm real-time systems** - where a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete system failure
- **Hard real-time systems** - where the tasks have to be performed both correctly and on time

Example Real-Time Systems

- *Avionics weapon delivery system in which pressing a button launches a missile*
  - **Hard** - missing the deadline to launch the missile after pressing the button may cause the target to be missed, resulting a catastrophe
- *Navigation controller for autonomous drone for irrigation*
  - **Firm** - missing a few navigation deadlines causes the drone to steer out of the planned path which would not damage the whole farm
- *Using a browser in an OS*
  - **Soft** - missing several deadlines will only degrade performance

Foreground/Background Systems

- Small systems of low complexity are generally designed as background/foreground (super-loops)
  - **Background** (task level) - Application consists of an infinite loop that calls modules (functions) to perform the desired operations.
  - **Foreground** - (interrupt level) ISRs handle asynchronous events.

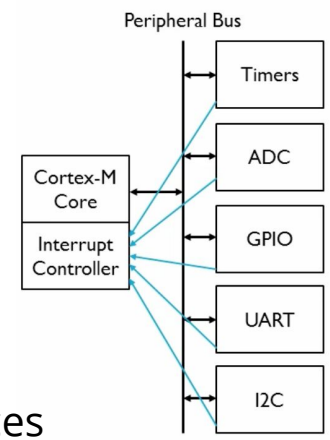


Foreground/Background Systems

- Critical operations must be performed by the ISRs to ensure that they are dealt with in a **timely fashion**.
- Any information for a background module made available by an ISR is not processed until the background routine gets its turn to execute (**task level response**)
- Most high volume microcontroller-based applications are designed as foreground / background systems.
- Also it might be desirable to put the microprocessor to sleep and perform all the processing in ISRs from a power consumption point of view.

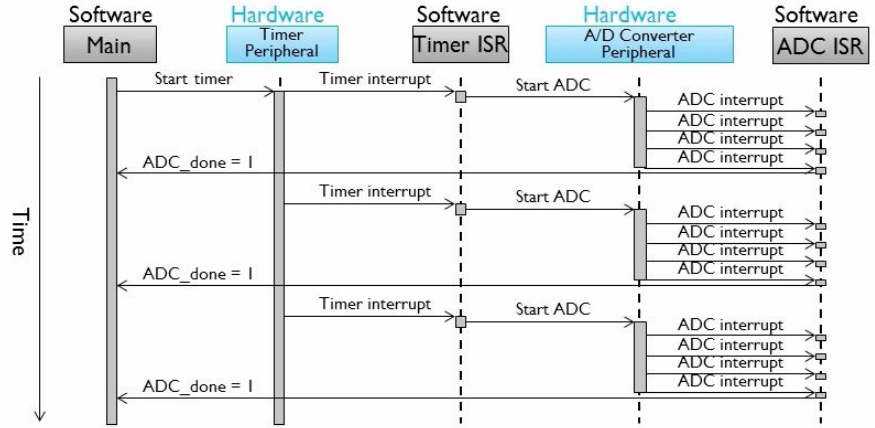
MCU Hardware & Software for Concurrency

- CPU executes instructions from one or more thread of execution
- Specialized hardware peripherals add dedicated concurrent processing
  - Watchdog timer
  - Analog interfacing
  - Timers
  - Communications with devices
  - Detecting external signal events
- Peripherals use IRQs to notify CPU of events



Concurrent Hardware & Software Operation

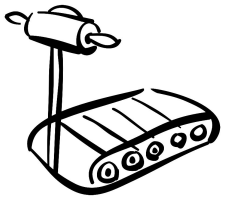
- Embedded systems rely on both MCU hardware peripherals and software to get everything done on time.



Event-Triggered Scheduling using IRQs

- Basic architecture, useful for simple low-power devices
  - Very little code or time overhead
- Leverages built-in task dispatching of interrupt system
  - Can trigger ISRs with input changes, timer expiration, UART data reception, analog input threshold crossing comparator
- Function types
  - Main function configures system and then goes to sleep
    - If interrupted, it goes back to sleep after servicing the interrupt
  - Only interrupts are used for normal program operation

Example: Event-triggered



- Treadmill Computer
  - rotation, mode key, clock, LCD
- main code:
  - configure timer, inputs and outputs
  - sleep in while loop
- ISR1 - rotation:
  - find the speed from length
  - increment rotations
- ISR2 - mode:
  - change the mode upon button press
- ISR3 - clock:
  - get time of the day from RTC
  - display it on LCD

CPU Scheduling

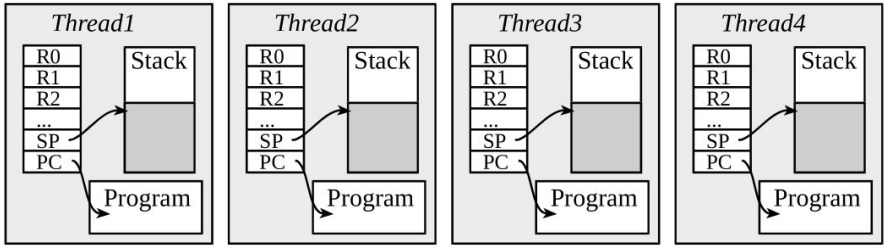
- MCU's interrupt system provides a basic scheduling approach for CPU
  - Run this subroutine everytime this hardware event occurs
  - Adequate for simple systems
- More complex systems need to support multiple concurrent independent threads of execution
  - Use task scheduler to share CPU
  - Different approaches to task scheduling
- How do we make the processor responsive?
- How do we make it do the right things at the right times?
  - If we have more software threads than hardware threads, we need to share processor.

Real-Time Operating Systems (RTOS)

- An **operating system** is a software that manages a computer's hardware, provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.
- An operating system usually covers functionality and correctness, but does not cover timeliness.
- **Real-Time Operating Systems (RTOS)** provide three essential functions with respect to software **tasks**
  - Scheduling
  - Dispatching
  - Inter-task communication and synchronization

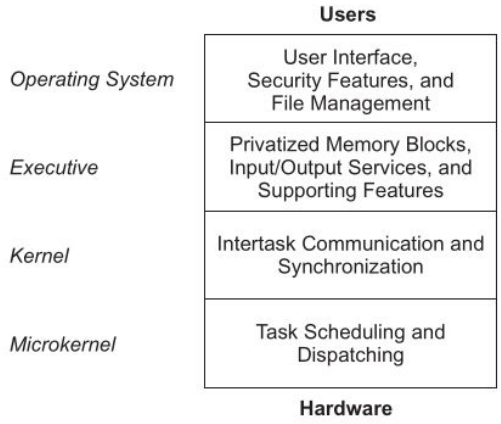
What is a Task (Thread)?

- A **task**, also called a **thread**, is an abstraction of a running program and is the logical unit of work schedulable by the operating system.
  - We can also describe it as a simple program that thinks it has the CPU all to itself.
- The design process for real-time application involves **splitting the work** to be done **into tasks** which are responsible for a portion of the problem.
- Each task is assigned a **priority**, its own set of **CPU registers**, and its **own stack area**.



Kernel

- The kernel is the part of a multitasking system responsible for the **management** of tasks (managing the CPU's time) and **communication** between **tasks**.
- The fundamental service provided by the kernel is **context switching**.



Context (Task) Switch

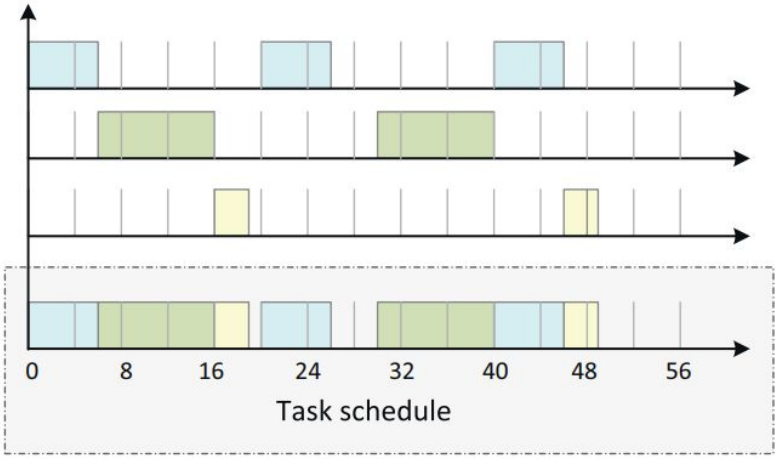
- When a multitasking kernel decides to run a different task, it **saves** the current **tasks context** (CPU registers) in the current tasks context storage area (it's stack)
- Once completed, the new task's **context** is **restored** from its storage area and the new task **resumes** to execution.
- Context switching adds **overhead** to the application.
- Written in Assembly language.

Scheduler

- The **scheduler** is part of the kernel responsible for determining which task will run next.
- Most real-time kernels are priority based. Each task is assigned a priority based on its importance, and the control of the CPU will always be given to the highest priority task read-to-run.
- When the highest-priority task gets the CPU is determined by the type of kernel used:
  - non-preemptive
  - preemptive

Example timing for task scheduling

Task A: period = 20 ms, execution time = 6 ms, deadline = 20 ms
Task B: period = 30 ms, execution time = 10 ms, deadline = 30 ms
Task C: period = 40 ms, execution time = 3 ms, deadline = 40 ms

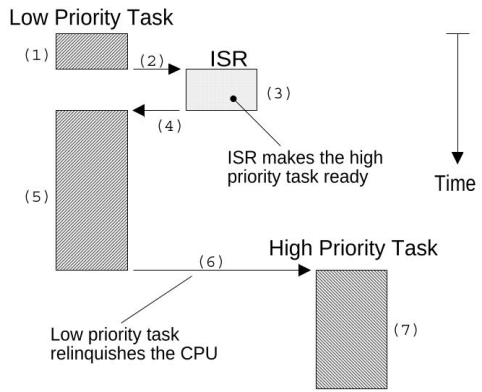


Non-Preemptive Kernel

- Non-preemptive kernels require that each task does something to explicitly give up control of the CPU (in other words each task **runs to completion**)
- To maintain the illusion of concurrency, this process must be done frequently.
- Non-preemptive scheduling is also called **cooperative multitasking** - tasks cooperate with each other to share the CPU.
- One of the advantages of a non-preemptive kernel is that the interrupt latency is typically low.
- Another advantage is the lesser need to guard shared data between tasks.
- Main disadvantage is responsiveness. A higher priority task that is ready to run may have to wait a long time for current task to give up the CPU

Non-Preemptive Kernel

- Asynchronous events are handled by ISRs.
- ISRs can make a higher priority task ready to run, but needs to give control back to the interrupted task.

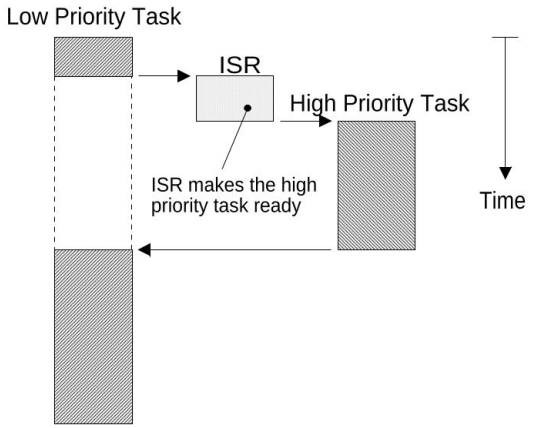


Preemptive Kernel

- A preemptive kernel is used when system responsiveness is important
- The highest priority task ready to run is always given control of the CPU.
- When a higher priority task is ready to run, the current task is preempted (suspended) and the higher priority task is immediately given control of the CPU.
- Execution of highest priority task is deterministic.
- Need to be careful about shared data, and atomic operations.
- Most commercial kernels are preemptive.

Preemptive Kernel

- If an ISR makes a higher priority task ready, when the ISR completes, the interrupted task is suspended and the new higher priority task is resumed.



Reentrancy

- **Reentrant function** is a function that can be used by more than one task **without fear of data corruption**.
- **Can be interrupted at any time** and resumed at a later time without loss of data.
- Use either local variables (CPU registers, or variables on the stack) or protect data when global variables are used.

Example reentrant function

```
void strcpy(char *dst, char *src) {
   while (*dst++ = *src++);
   *dst = NULL;
}
```

- An example reentrant function is given above.
- Because the arguments of the function are placed on the task's stack, the function can be invoked by multiple tasks without fear that the tasks will corrupt each other's pointers.

Example non-reentrant function

```
int Temp;
void swap(int *x, int *y) {
   Temp = *x;
   *x = *y;
   *y = Temp;
}
```

- An example non-reentrant function is given.
- Because the temp is a global, tasks can and will corrupt each other's data.
- It is hard to catch / solve problems arise from reentrancy

Example non-reentrant function

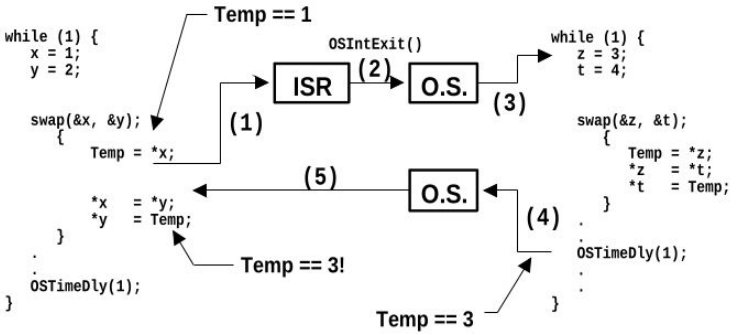
```
int Temp;
void swap(int *x, int *y) {
   Temp = *x;
   *x = *y;
   *y = Temp;
```

LOW PRIORITY TASK

```
while (1) {
   x = 1;
   y = 2;

   swap(&x, &y);
      {
         Temp = *x;

         *x   = *y;
         *y   = Temp;
      }
   .
   .
   OSTimeDly(1);
}
```

HIGH PRIORITY TASK

```
while (1) {
   z = 3;
   t = 4;

   swap(&z, &t);
      {
         Temp = *z;
         *z   = *t;
         *t   = Temp;
      }
   .
   .
   OSTimeDly(1);
   .
   .
}
```



Example non-reentrant to reentrant

```
int Temp;
void swap(int *x, int *y) {
   Temp = *x;
   *x = *y;
   *y = Temp;
}
```

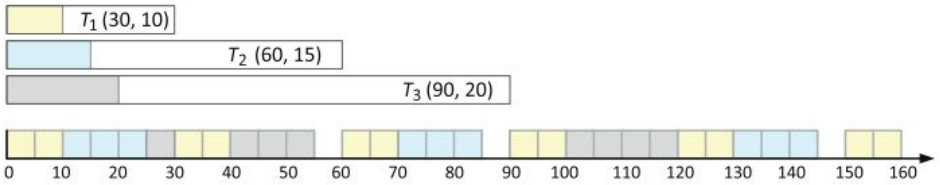
- We can make the **swap** function reentrant by using one of the following techniques:
- declare **Temp** local to **swap**
- **disable interrupts** before the operation and enable them after
- use a **semaphore**

Thread Scheduler

- List possible thread states
- Possible scheduling algorithms
  - What (order of threads), round robin, weighted round robin, priority
  - How (when to decide) - static, dynamic, deterministic, fixed
  - Why (when to run), cooperative preemptive
- When to run scheduler
- Performance measures
  - Utilization, Latency, Bandwidth, Response Time

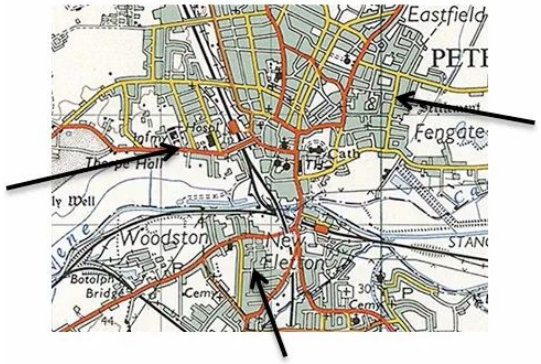
Worst-Case Response Time

- The performance of a real-time system is typically expressed in terms of the **(worst-case) response time**
- The response time of a job is defined as the length of time from the release time of the job to the instant when it finishes.
- The response time of a job depends heavily on its execution time, as well as on how jobs are scheduled by the system.



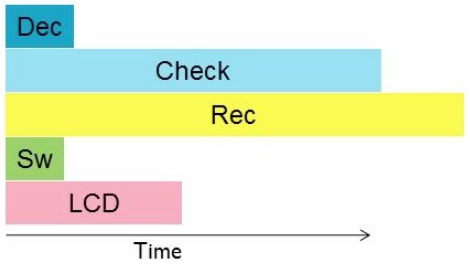
Example: GPS-Based hole alarm w/ map

- Sounds alarm when approaching a hole
- Displays vehicle position on LCD
- Logs driver's position information



Example: GPS-Based hole alarm w/ map

- **DEC**: Decode GPS data to find the vehicle position
- **CHECK**: Check to see if approaching any hole locations. Will increase as the number of holes in the database increases
- **REC**: Record position to flash memory
- **SW**: Read user input switches (~ 10 times per sec)
- **LCD**: Update LCD with map data (~ 4 times per sec)

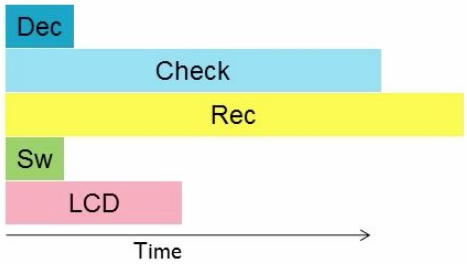


What are our choices?

- **Bare metal** - No OS at all - the easiest starting point, the hardest to get right
- **Simple scheduler** - Define tasks and have them go around in an order
- **RTOS** - Define tasks and all the other data communication and synchronization mechanisms
- **General Purpose OS** - For embedded systems, usually Linux. Comes with all the extra stuff for keeping it running.

How do we schedule these tasks?

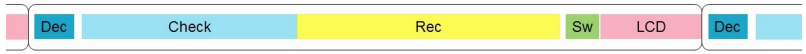
- Task scheduling: Deciding which task should be running now
- Do we run tasks in the same order every time?
  - Yes - static schedule. Cyclic, round-robin
  - No - dynamic schedule. Priority
- Can one task preempt another, or must it wait for completion?
  - Yes - preemptive
  - No - non-preemptive (cooperative, run-to-completion)



Static schedule (Cyclic Executive)

- Very simple
- Always run the same schedule regardless of changing conditions and relative importance of tasks
- All tasks run at the same rate. Changing rates requires adding extra calls to the function
- Maximum delay is sum of all task run times. Polling / execution rate is 1/max delay

```
while(1) {
   Dec();
   Check();
   Rec();
   Sw();
   LCD();
}
```



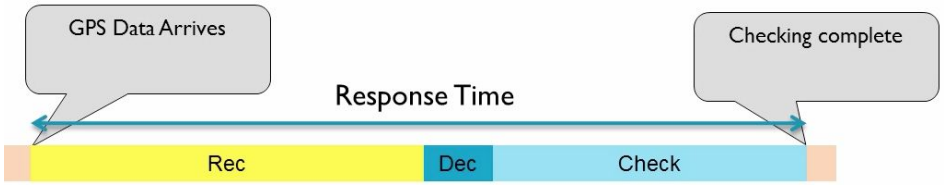
- ●

Dynamic Scheduling

- Allow schedule to be computed on-the-fly
  - Based on importance or something else
  - Simplifies creating multi-rate systems
- Schedule based on importance
  - Prioritization means that less important tasks don't delay more important ones
- How often do we decide what to run?
  - **Coarse grain** - After a task finishes. Also called run-to-completion (RTC) or non-preemptive
  - **Fine grain** - Any time. Called preemptive, since one task can preempt another

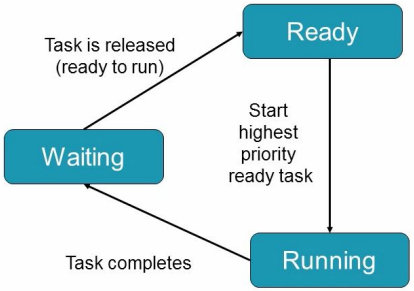
Dynamic Non-preemptive Scheduling

- If data arrives while in Rec state, we can return to Dec / Check states after completing Recording
- Still has extra Record time for responsiveness.



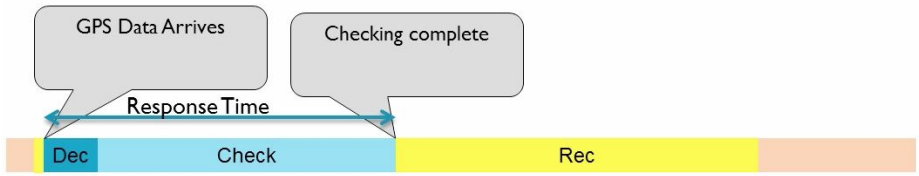
Dynamic Non-preemptive Scheduling

- Scheduler chooses among Ready tasks for execution based on priority
- Scheduling Rules
  - If no task is running, scheduler starts the highest priority ready task
  - Once started, a task runs until it completes
  - Tasks then enter waiting state until triggered or released again



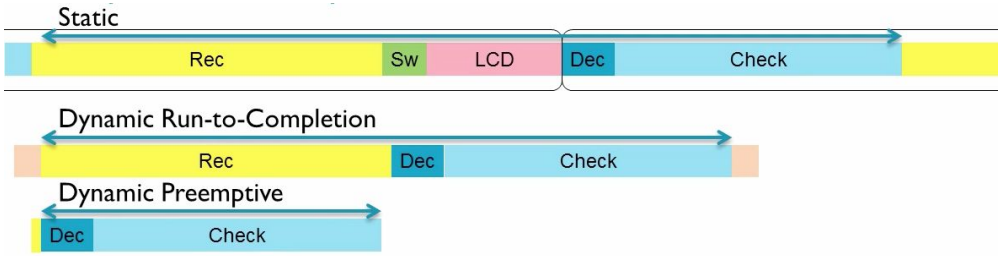
Dynamic Preemptive Scheduling

- If data arrives while in Rec state, it will preempt the current running task and start the Dec / Check states
- After completion of check, it will return to the preempted task.



Comparison of Response Times

- Preemption offers best response time
- Can do more processing, lower speed and save power
- Requires more complicated programming (reentrancy, memory)
- Introduces vulnerability to data race conditions



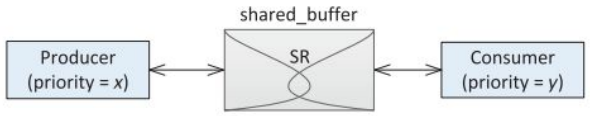
Resource Sharing

Shared Variables - shared access using global variables through which data can flow back and forth among them.

Shared Memory - An address space can be made accessible by the tasks

These shared approaches will lead to problems with the data if not properly handled.

- Data integrity
- Data loss
- Data duplication



Semaphore

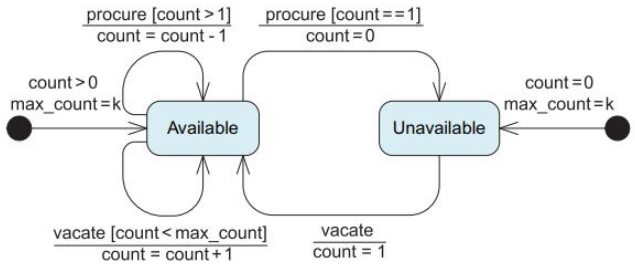
- When multiple tasks gain access to a shared resource at the same time, the integrity of the resource is endangered.
- This block of code that accesses a shared resource is often called a **critical section**.
- Two critical sections that access the same exclusive resource are called **contenting critical sections**.
- **Semaphore** is a mechanism to protect shared resources such that no two contending critical sections can be executed at the same time.

Semaphore

- A semaphore can be specified with an initial value denoted by count.
  - At any time, a semaphore is available only if count > 0
  - it is unavailable when count <= 0.
- As a resource lock, a semaphore has a task waiting list containing those tasks that are blocked because of the unavailability of the semaphore.
- A semaphore value (count) can be viewed as the number of keys to a lock protecting a shared resource.
- Each key represents a privilege to access the protected resource.
- Depending on the OS convention procure / vacate, wait / post, acquire / release, pend / post can be used to get and release locks.

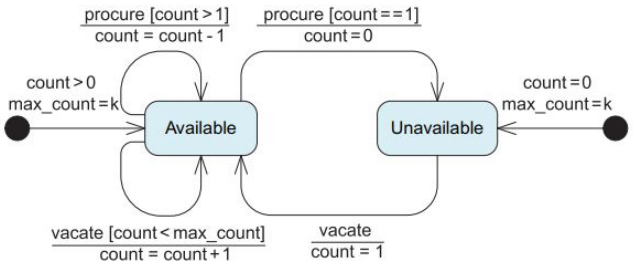
Semaphore - Procure

- A task executes the procure operation on a semaphore to request a privilege.
  - If currently available (i.e. count > 0) the semaphore value is decreased by 1, and the task has granted access to the protected resource.
  - If not available (count = 0), the task is added to the task waiting list.



Semaphore - Vacate

- A task executes the vacate operation on a semaphore to request a privilege.
  - If there are no tasks in the waiting list, this operation increases the semaphore value by 1.
  - If there are tasks in the waiting list, one task blocked waiting for the semaphore is unblocked and returns successfully from its procure operation



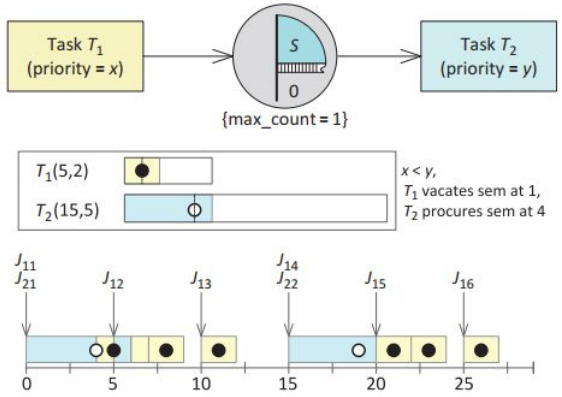
Semaphore use cases

A semaphore can be used in different scenarios

- **Task synchronization** - to sync one task to another
- **Task rendezvous synchronization** - to sync both tasks to the same point
- **Flow control** - to organize data transfer from one task to another
- **Resource protection** - to protect the integrity of shared resource

Example: Task synchronization

- Semaphores can be used as a tool for **inter-task synchronization.**
- For example two tasks can use a semaphore with maximum count as 1 to synchronize their operation
- This type of semaphore is called a **binary semaphore.**



Mutex

- Semaphores have the limitation that they have **no ownership** which poses two main problems
  - A rouge task might procure or vacant a semaphore that will change the operation.
  - A task might have several places that need to gain access to the same resource protected by a semaphore.
- **Mutex** is another type of **resource lock.**
- As a synchronization or resource protection tool, mutex is like an **unnamed binary semaphore.**

Intertask Communication

- In a multitasking system there may exist tasks that need to cooperate with each other by exchanging messages.
- This is called **intertask communication**.
- Classified into two types:
  - **Synchronous communication** - the sender and receiver need to wait for each other until the message transmission is complete
  - **Asynchronous communication** - the sender may continue to execute its next instruction while the message is being delivered to receiver

Message Queues

- A **message queue** is a buffer-like object that acts as a liaison between a message sender and a message receiver.
- Messages sent to a queue are stored in the queue and can be retrieved at any time by a receiver.