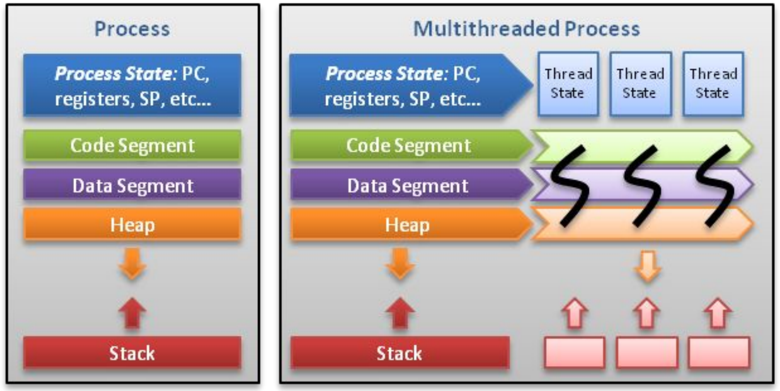
Multitasking & Real-Time Operating System

Programming and Development of Embedded Systems

Multitasking

- A task is a unit of work/execution which provides some system functionality
  - E.g. Watering system of the greenhouse
- Multitasking is the execution of multiple tasks over a certain period of time
- Multitasking is performed using processes or threads
- A process is an instance of a program that is being executed by one or more threads
  - Processes are execution units that don't share memory space.
- A thread is a sequence of instructions within a program that can be executed independently
  - Threads are like workers.
  - Threads are execution units that share memory space
- A thread has its own context
  - Information to allow a thread to be scheduled independently; such as the stack, registers and thread-specific data



Multitasking - Threads

- To create a new thread we can use pthread_create. This also starts the thread.

```
pthread_t thread_id;
pthread_attr_t* attr = NULL;
void* args = NULL;
pthread_create(&thread_id, attr, function, args);
```

- Thread Function: A task to run as a thread has to have the following interface:

```
void* function(void* args);
```

- The argument can be any data type, but needs to be casted to a void*
- And when the function returns it returns anything as long as it is casted to void*.

```
void* counter(void* startVal) {
    int *countVal = startVal;
    (*countVal)++;
    pthread_exit(countVal);
}
```

Multitasking - Threads

- Thread Exits
  - When a thread is completed there are different ways to exit it:
    - return
    - pthread_exit()
    - program exit
    - E.g. pthread_exit(void* retval);
  - The main program can wait for the threads to finish using thread_join()

    pthread_join(pthread_t id, void** retval);

    - This will wait until the thread is finished and returns the return value. Example:

```
void* retval;
pthread_join(thread_id, &retval);
int* value = retval;
printf("Return value: %d", *value);
```

Multitasking - Mutex

- To protect a shared resource from multiple access at the same time **mutex** is used
  - E.g. A printer connected to a PC. What happens if two tasks print at the same time
- A mutex (Mutual Exclusion) provides the lock/unlock mechanism to prevent multiple access to a shared resource to change/update it by multiple threads
- Example

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
…
pthread_mutex_lock(&mutex);
i++;
pthread_mutex_unlock(&mutex);
…
pthread_mutex_destroy(&mutex);
```

Example: Using two threads make a program to print "**Ping - Pong**" 10 times to the terminal. A thread shall print **Ping** and another thread shall **Pong** to the terminal. Ensure the right order so that the output looks like:

```
Ping - Pong
Ping - Pong
Ping - Pong
...
```

What is the main challenge with making the right order?

Multitasking - Condition Variables

- Condition variables are used to synchronize threads
  - Mutexes are used to control access to data
  - But condition variables are used to synchronize threads based on the value of data
    - By providing wait/signal mechanism
  - A condition variable is always used in conjunction with a mutex lock.
- To create a condition variable:
  - Statically: ***pthread_cond_t condition = PTHREAD_COND_INITIALIZER;***
  - Dynamically: ***pthread_cond_t condition; pthread_cond_init (condition, attr);***
    - The optional **attr** object is used to set condition variable attributes.
    - There is only one attribute defined for condition variables:
      - process-shared, which allows the condition variable to be seen by threads in other processes.
    - The attribute object, if used, must be of type ***pthread_condattr_t***
      - May be specified as NULL to accept defaults

Multitasking - Condition Variables

- To destroy a condition variable ***pthread_cond_destroy (condition)*** is used.
- The ***pthread_condattr_init()*** and ***pthread_condattr_destroy()*** routines are used to
  - Create and destroy condition variable attribute objects.
- **Waiting** and **Signaling** on Condition Variables
  - ***pthread_cond_wait()*** blocks the calling thread until the specified condition is signalled
    - It should be called while **mutex is locked**
    - it will automatically **release the mutex** while it waits.
    - After signal is received and thread is awakened
      - The mutex will be automatically locked for the thread
    - You are responsible to unlock the mutex when you don't need it.
    - **Recommendation**: Use a **WHILE** loop instead of an **IF** statement to check the condition

Multitasking - Condition Variables

- **Waiting** and **Signaling** on Condition Variables …
  - **pthread_cond_signal()** signals (or wake up) a waiting thread on the condition
    - It should be called after mutex is locked
    - And then you must unlock the mutex
  - **pthread_cond_broadcast()** is used instead of pthread_cond_signal
    - If more than one thread is in a blocking wait state
  - Don't call **pthread_cond_signal**() before calling **pthread_cond_wait**() - logical error
  - Proper locking and unlocking of the associated mutex variable is essential.
- **Exercise 19**: Create a program with two threads and a condition variable
  - A thread prints "Ping" and the other thread prints "Pong".
  - Ensure the right order so that the output looks like =====>
  - The application shall do this 10 times before exit.

```
Ping - Pong
Ping - Pong
Ping - Pong
...
```

Multitasking - Exercise 20

- Multithreading using **TeensyThreads.h** on Teensy 3.5
  - Using a mutex, and threads.delay, threads.yield and threads.addThread functions make a program with 3 synchronized threads to print 1, 2 and 3 to a terminal in order.
  - To create a **mutex,** use Threads::Mutex. E.g. `static Threads::Mutex mutex;`
  - To create a thread, use threads.addThread. E.g. `threads.addThread(print_one);`
  - To make a context switch, use `threads.yield();`
  - To make a delay results in a context switch, use threads.delay. E.g. `threads.delay(500);`
  - To synchronize the threads, you need to emulate a condition variable
  - In the **loop** function, make the built-in LED blinking every 500ms using **threads.delay**.
  - The printed numbers shall be in order and look like:

```
1 - 2 - 3
1 - 2 - 3
1 - 2 - 3
...
```

Multitasking - Semaphore

- To provide synchronization of inter-task communications **semaphores** are used
- Semaphores provide **waiting/signaling** mechanism to synchronize tasks execution
  - They act like traffic lights or gates
  - They can be used to manage the execution order of tasks
  - They can also be used to protect shared resources and critical sections
- A Semaphore is an integer variable which can be changed atomically
  - An instance of a semaphore can be created as *sem_t semaphore;*
  - A semaphore can be initialized by *sem_init (sem_t *s, int pshared, unsigned int value);*
  - A semaphore can be destroyed by *sem_destoy(sem_t *s);*
- A Semaphore has two operations to manipulate its value
  - **sem_wait** which decrements its value; e.g. *sem_wait(&semaphore);*
  - **sem_post** which increments its value; e.g. *sem_post(&semaphore);*

Multitasking - Semaphore

- If the value of a semaphore is **0** and a **wait** is called the caller task gets suspended
- **sem_post** sends the semaphore to a waiting task and wakes it up
- Semaphore is a generalized mutual exclusion
  - If we initialize a semaphore with **1**, we have a binary semaphore and it acts like a **mutex**.
  - Mutual exclusion with more than one resource
    - Counting semaphore: X > 1; Initialize to the number of available resources

```
sem_init(s, 0, X); // X = 1
sem_wait(s);
// critical section
sem_post(s);
```

- A semaphore can manage execution order
  - A task can wait for another

```
// Thread 0
printf("Ping - ");
sem_post(s);
```

```
// Thread 1
sem_wait(s)
printf("Pong\n ");
```

- **Exercise 21**: Producer-Consumer
  - A Producer produces products and a consumer consumes the products with different rates
  - The producer is faster than the consumer and a storage which can hold max. 5 products is used
  - If the storage is empty, the consumer should wait
  - If the storage is full, the producer should wait

Real-Time Operating System

- An operating system is a system program which manages resources in a computer
  - Resources like processor, I/O devices, memory, user access and etc.
- A Real-time system is a system whose behavior depends on real time
  - Meets specified deadlines and its response should be guaranteed within a specified time.
- Types of real-time system
  - Critical real-time system (also called hard real-time system)
    - Is a safety-critical system. No tolerance for missed deadlines.
    - Missing deadlines cause failure and catastrophe. E.g. an emergency braking system
  - Non-critical real-time system
    - **Firm real-time system**: Needs to follow the deadlines. But missed deadlines are accepted, not desired
      - Missing a deadline may result in unacceptable software quality. E.g. multimedia applications
    - **Soft real-time systems**: Deadlines are handled softly. But a small amount of delay is acceptable.
      - Missed deadline can be recovered. E.g. online transaction systems

Real-Time Operating System

- Real-time Operating System
  - An OS used in real-time systems to manage resources and fulfill timing requirements
    - Commercial examples like VxWorks, LynxOS-POSIX and etc.
    - Open source examples like Chibios, freeRTOS and etc.
- Offers multitasking and priority-based preemptive scheduling
  - Allows us to separate critical tasks from non-critical tasks
- Provides services and API functions for applications
- Offers modular task-based development, which allows modular task-based testing.
- Is time-sharing and event-driven with no time wastage on events which are not occurred
- Occupies very less memory and consumes fewer resources
- Is deterministic and response times are highly predictable
- Guarantees correct computations at correct times

Real-Time Operating System

- Real-time computing does not mean it’s faster, it means only guaranteed timing.
  - Normally an RTOS is slower than a GPOS
- Uses timer interrupt to schedule and manage the tasks to achieve timing
- Supports event-triggered and time-triggered executions
- Uses context switch and each task has its own stack; and even possibly its own heap
- Supports static and dynamic memory allocations
- Is complex and expensive
- The basic functionalities of an RTOS are:
  - **RTOS Services** like interrupt handling services, time services, memory management and etc.
  - **Synchronization and messaging** like mutexes, semaphores, pipes and mailboxes and etc.
  - **Scheduler** to schedule multiple tasks and control multiple access to resources like the processor

Real-Time Operating System - Task Scheduler

- Tasks should prioritized and usually are run cyclically
- Tasks can have three states:
  - **Ready**: When a task has all the resources to get run, but still it is not in running state
  - **Running**: when CPU is executing a task, then it is in running state
  - **Blocked**: when a task has not all required resources for execution
- Types of Scheduling: **Cooperative** (aka. Non-Preemptive) and **Preemptive**
- Cooperative Scheduling
  - The scheduler never initiates a context switch from a running task to another task
  - A task holds the CPU until it gets terminated or it reaches a blocked state
  - All tasks run at specific intervals and tasks must be designed to not steal all the processor time
  - Can not guarantee timing between tasks
  - Tasks have no task-specific memory; all shared
  - E.g. Shortest job first, priority non-preemptive algorithms

Real-Time Operating System - Task Scheduler

- Preemptive Scheduling
  - The scheduler does context switching
  - A task can switch from running state to ready state or from blocked state to ready state
  - The processor is allocated to a task for a limited amount of time
    - If the task is not completed, it is again placed back in the ready state
    - The task stays in ready state until it gets next chance to execute
  - Tasks can be prioritized
    - A higher priority task requires CPU service first than a lower priority task
  - E.g. Priority preemptive, Round Robin and etc.
    - In Round Robin, time sharing; and context switching is done after a fixed time quantum

Multitasking & Real-Time Operating System

- Some useful links
  - Multithreaded Programming (POSIX pthreads Tutorial)
  - POSIX Threads Programming
  - Chapter 4 Programming with Synchronization Objects
  - Using Condition Variables
  - Mutex vs Semaphore
  - Real-time operating system (RTOS)
  - What is an RTOS?
  - What is Real Time Operating System (RTOS)- How it works?
  - What is real-time operating system (RTOS)?
  - What is an RTOS?
  - Real-Time Operating System (RTOS) Concepts
  - ChibiOS - Arduino
  - FreeRTOS Kernel