11 - Real Time Operating System

CEG 4330/6330 - Microprocessor-Based Embedded Systems

Max Gilson

Operating Systems (OS)

- Kernel and user interface
  - Kernel implements underlying functionality
  - User interface implements the look and feel of the OS
- Arduino Uno does not use an OS
- An OS must perform a few key functions:
  - Task management
  - Memory management
  - Storage management

Task Management

- Task Management
    - A task can be a program or a part of a program
    - If you have 1 processor or thread, the computer is not useful unless it can switch between tasks and programs on the fly
    - Multi-tasking is the ability to switch between multiple programs
        - Display user interface
        - Execute a program
        - Background processing
    - The method used is called a CPU scheduling algorithm

Memory Management

- Memory Management
  - If there are multiple tasks running, they are loaded into RAM
  - Assume you are using a Raspberry Pi to watch YouTube, a program/task must interface with the WiFi device and another program (web browser) must display this information
    - In this case, two tasks are sharing the same memory
  - Virtual memory swapping
    - The OS can swap out task's portions of physical memory that are not currently needed
    - The memory that is not needed is saved to some local storage (hard drive, SSD, SD card)
    - This is only needed if the physical memory is not large enough for all tasks to be running

Storage Management

- Storage Management
    - A file system is almost always hierarchical
    - You are probably familiar with files and folders in Windows or MacOS
        - A folder can contain many folders, and that folder can contain folders, and so on…
        - Each folder can hold multiple files
    - A file system keeps data organized, without it, tasks would have a hard time finding external data
        - Imagine a file system without folders where everything is saved to the storage in a first come first serve basis
        - It would be very difficult to organize or find your data

Real Time Operating System (RTOS)

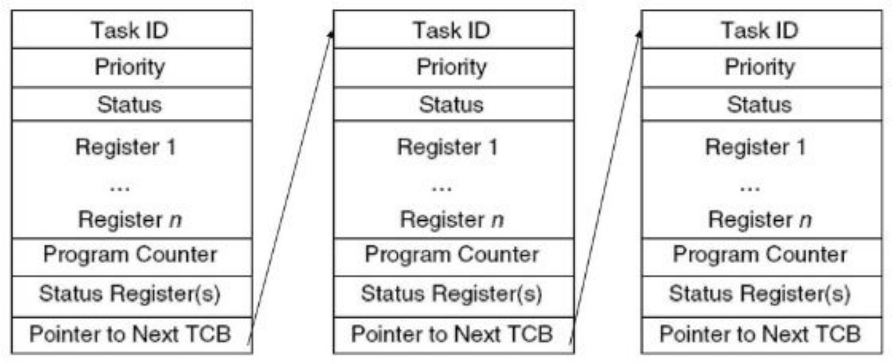
- An OS that handles multiple tasks in a timely manner or reacts to input within a specific time period
- Real time systems can be defined by
  - Hard (leads to system failure)
  - Firm (a low occurrence that can be tolerated)
  - Soft (leads to performance degradation)
- RTOS' fall into two categories
  - Hard RTOS (guaranteed to meet deadlines)
  - Soft RTOS (meets deadlines a percentage (maybe 90%) of the time

RTOS Basics

- Kernel is responsible for task scheduling
  - Tasks can have one of many states
    - Active (currently running on processor)
    - Ready (ready to execute)
    - Blocked (waiting for resources)
    - Sleeping (waiting for some time)
  - Tasks can have different priorities
    - Windows has priority levels from 0 to 31
    - Linux has priority from levels -20 to 19 (default 0)
    - Tasks with higher priority are allowed more time to run compared to lower priority

Task Control Block and Scheduling

- Task control block (TCB) contains all the information required for the execution of a task
- The TCB maintains the PC, CPU registers, virtual memory information, and more
- A task scheduler will organize the linked list of TCBs and provide timely execution
  - The task scheduler can be disabled if a task is required to finish executing to meet a deadline



Types of RTOS

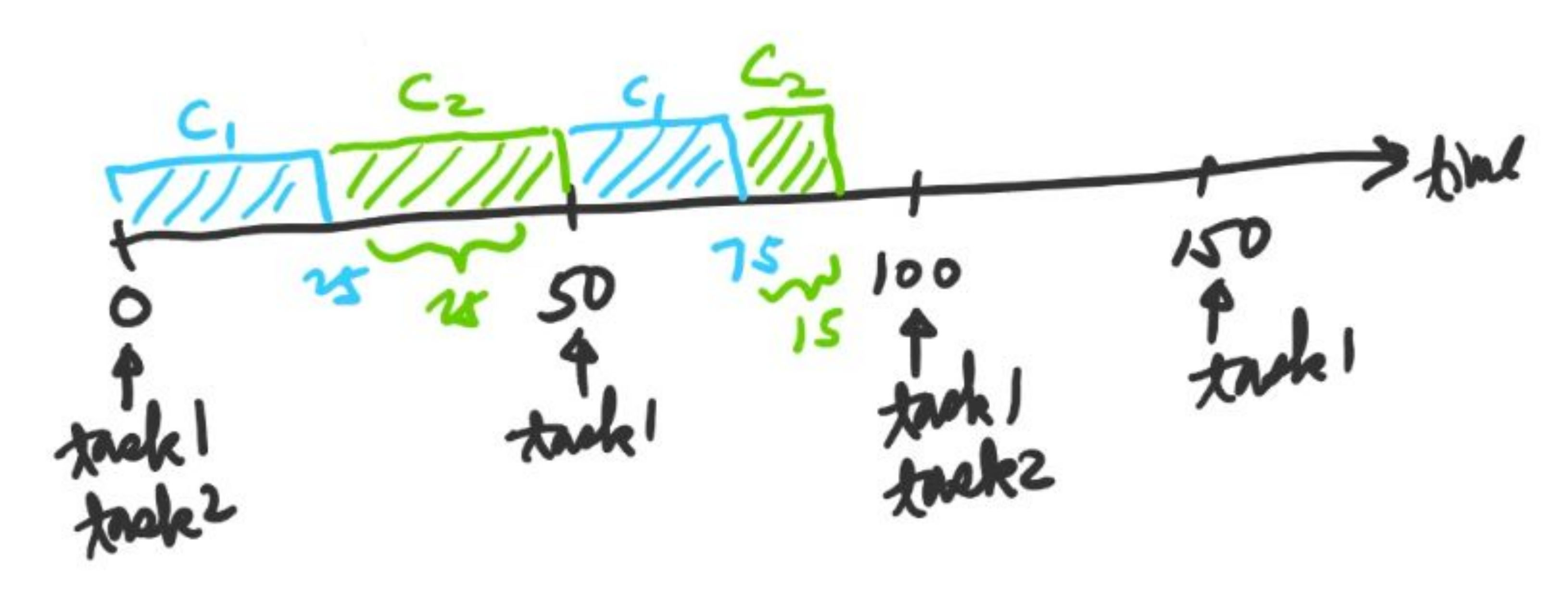
- Cooperative Multitasking
  - Tasks must voluntarily relinquish control back to OS
  - After a task relinquishes control, the highest priority task is scheduled
  - A low priority task may never get CPU time
- Preemptive Multitasking
  - Low priority tasks are switched out from CPU to make way for high priority tasks
  - The scheduler does not wait until the low priority task is finished (preemptive)
- Many embedded systems allow for switching between both types of multitasking

Task Scheduling Example

- Two periodic tasks:
- Ci: completion time (including context switching)
- Ti: task release period
- U = 0.9 (utilization) = C1/T1 + C2/T2
  - C1 = 25, T1 = 50
  - C2 = 40, T2 = 100
  - Priority with preemption
- Case 1: Task 1 has higher priority
  - All deadlines met
- Case 2: Task 2 has higher priority
  - Task 1 misses the first deadline

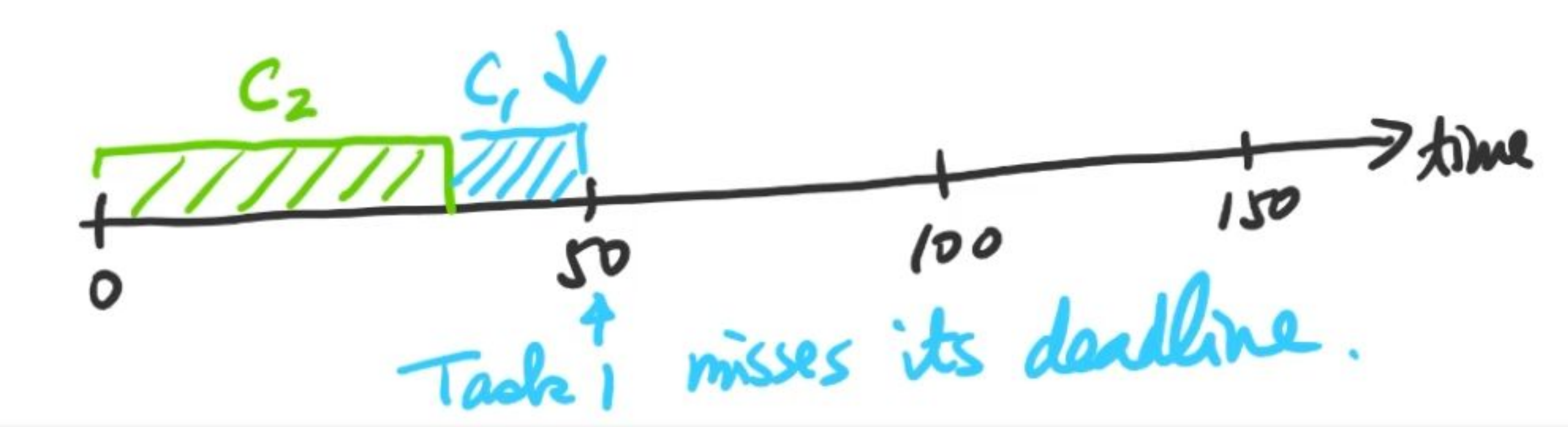
Task Scheduling Example Case 1

- Task 1 has higher priority:



Task Scheduling Example Case 2

- Task 2 has higher priority:



Rate Monotonic Scheduling

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \le n(\sqrt[n]{2} - 1) = \begin{cases} 0.8284 & when\ n = 2 \\ 0.6931 & when\ n = \infty \end{cases}$$

n: no. of tasks, C_i: service time, T_i : period

- Periodic Tasks: higher release rate (i.e. lower Ti) assigned higher priority with preemption
- Tasks (1 to n) are guaranteed to meet deadlines if the total utilization is lower than the upper utilization bound (U)
    - There may be situations where you can still meet deadlines above this utilization, like the previous example
- Non-periodic/non-real time tasks use the remaining time
    - Reading from the keyboard

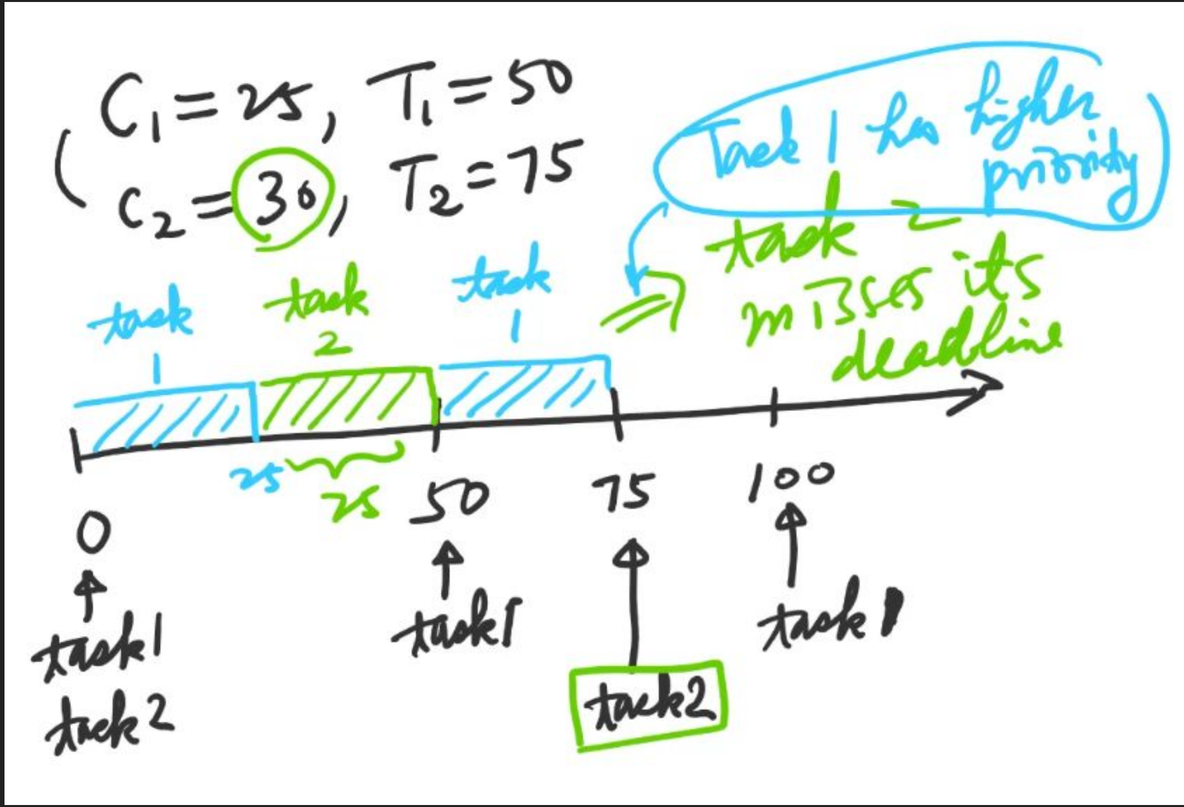
CPU Utilization

- C1/T1 is the percentage of time that Task 1 occupies the CPU
- C2/T2 is the percentage of time that Task 2 occupies the CPU
- The sum shows the total percentage of usage of the CPU
  - 0% means CPU has no tasks to execute
  - 100% means CPU is alway executing a task
  - If utilization is > 100%, tasks will always miss deadlines
- Sometimes, an OS on a processor with multiple cores will sum the percentage of each core in its utilization
  - Example: 400% utilization on a quad-core processor means every core is fully utilized, 100% means either only one core is fully utilized or each core is 25% utilized

Task Scheduling Example 2

- Two periodic tasks:
- Ci: completion time
- Ti: task release period
- U = 0.9 (utilization) = C1/T1 + C2/T2
  - C1 = 25, T1 = 50
  - C2 = 30, T2 = 75
  - Priority with preemption
- Case 1: Task 1 has higher priority
  - Task 2 misses deadline
- Case 2: Task 2 has higher priority
  - Task 1 misses deadlines
- In both cases the deadlines cannot be met
  - There is no configuration where these tasks can meet their deadlines

Task Scheduling Example 2



Task Scheduling Example 3

- Two periodic tasks:
- Ci: completion time
- Ti: task release period
- U = 0.8 (utilization) = C1/T1 + C2/T2
  - C1 = 20, T1 = 50
  - C2 = 30, T2 = 75
  - Priority with preemption
- Case 1: Task 1 has higher priority
  - Both tasks meet deadlines
- Case 2: Task 2 has higher priority
  - Both tasks meet deadlines
- In both cases the deadlines are always met
  - By satisfying the rate monotonic scheduling condition, we know no matter what, these tasks will meet their deadlines

Earliest Deadline First Scheduling

- The task with the earliest deadline is scheduled first
- This is a type of dynamic priority scheduling with preemption
  - Tasks may switch back and forth between higher and lower priorities based on how which task has the nearest deadline
- Guaranteed to meet deadlines if the utilization (U) is less than or equal to 1.0 (100% utilization)
- This scheduling algorithm is often impractical and difficult to implement and when U > 1.0, behavior is difficult to predict
  - It is an optimal algorithm, and if we can easily approximate its performance, that is ok

Concurrency

- Assume two tasks want to modify some register Rn
  - Initially: `Rn = B00000000`
  - Task A wants to set bit 3: `Rn |= BXXXXX1XX`
  - Task B wants to set bit 5: `Rn |= BXXX1XXXX`
  - After both tasks execute we expect `Rn = B00010100`
  - These are read-modify-write operations
    - Read Rn, Modify the read value, Write the modified value to Rn
    - If this read-modify-write process is interrupted, results may be undesirable
      - If Task A is switched out for Task B after modify, only bit 3 will be set in Rn at the end of both tasks
    - The read-modify-write operation must be labeled as a critical section or atomic (in C use a mutex to lock/unlock resources)
      - Can also use semaphores to wait before the critical section and signal afterwards

Semaphores

- Semaphores can be use to hold a section of code until it is completed by a task
- Pseudocode:

```
S1.wait()
//critical section of code goes here
S1.signal()
```

  - S1 is a resource that locks a section of code that other tasks should not be allowed to modify until S1.signal() has been reached
  - Example: Task 1 gets suspended during its critical section, if Task 2 uses the same semaphore (S1) it will wait() until Task 1 gets enough CPU time to reach signal()

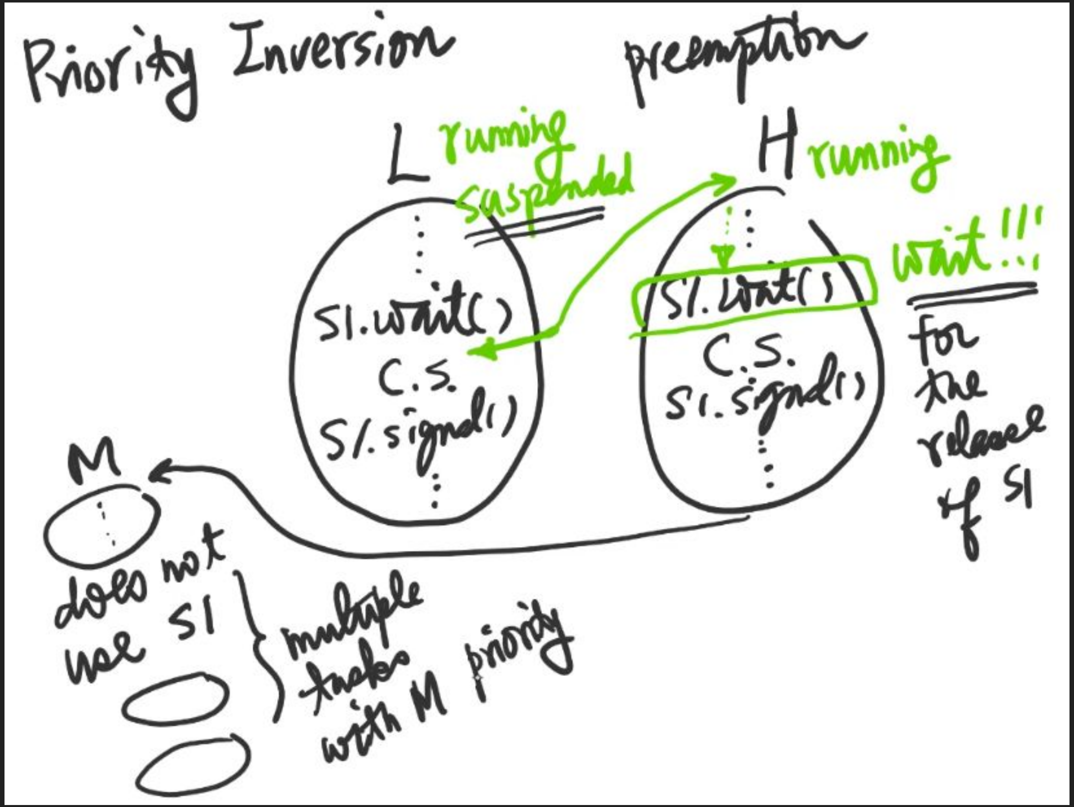
Priority Inversion

- A problem due to priority inversion caused the spacecraft to reset itself several times after it was landed on Mars on July 4, 1997.
- Resetting the spacecraft resulted in significant delay in capturing scientific data, which was critical for the mission given that the lifetime of the spacecraft was limited.

Priority Inversion (cont.)

- Using preemption with critical sections:
  - Assume low, medium, and high priority tasks L, M, H
  - L gets suspended by H during a critical section
  - H executes until its critical section is reached (must wait for L to finish its critical section)
  - M take over while H is waiting
    - Many M tasks may execute without ever giving L a chance to free its critical section
    - Execution order: L -> H -> M -> M -> M -> …
    - The highest priority task gets locked, and is not receiving any CPU time (BAD!)
- This is called priority inversion because the high priority task is being treated like a low priority task (never getting CPU time)

Priority Inversion (cont.)



Priority Inheritance

- To prevent priority inversion on a preemptive system, priority inheritance can be used
  - L is raised to the same priority as H, to ensure that it gets CPU time to release its critical section
  - This prevents the M tasks from hogging the CPU time from L, locking H

Reentrancy

- A reentrant function behaves correctly if executed simultaneously by several tasks
- A non-reentrant function is incapable of multiple tasks executing it simultaneously without issues
  - print(), malloc(), free()
- Printing slowly populates a buffer to print data out
  - If you are printing, and it gets interrupted with another print, the results will be undesirable
- Malloc allocates memory in the next available space, if malloc is called while malloc is running, the allocated spaces will be mixed within each other

Reentrancy (cont.)

- To create a reentrant function you must disable interrupts, use only local variables, do not use global variables, do not use registers or other shared data
- Any shared variables, registers, or hardware opens up the possibility for creating a function that causes problems when it is reentered

Inter-task Communication

- A mailbox can be used or message passing
  - A mailbox is a shared memory location, that multiple tasks can put information in and take out of
  - Only one task has a key to the mailbox at a time
  - This can allow tasks to communicate with each other
  - Message passing works like a queue, where multiple tasks can put messages in the queue for other tasks to read

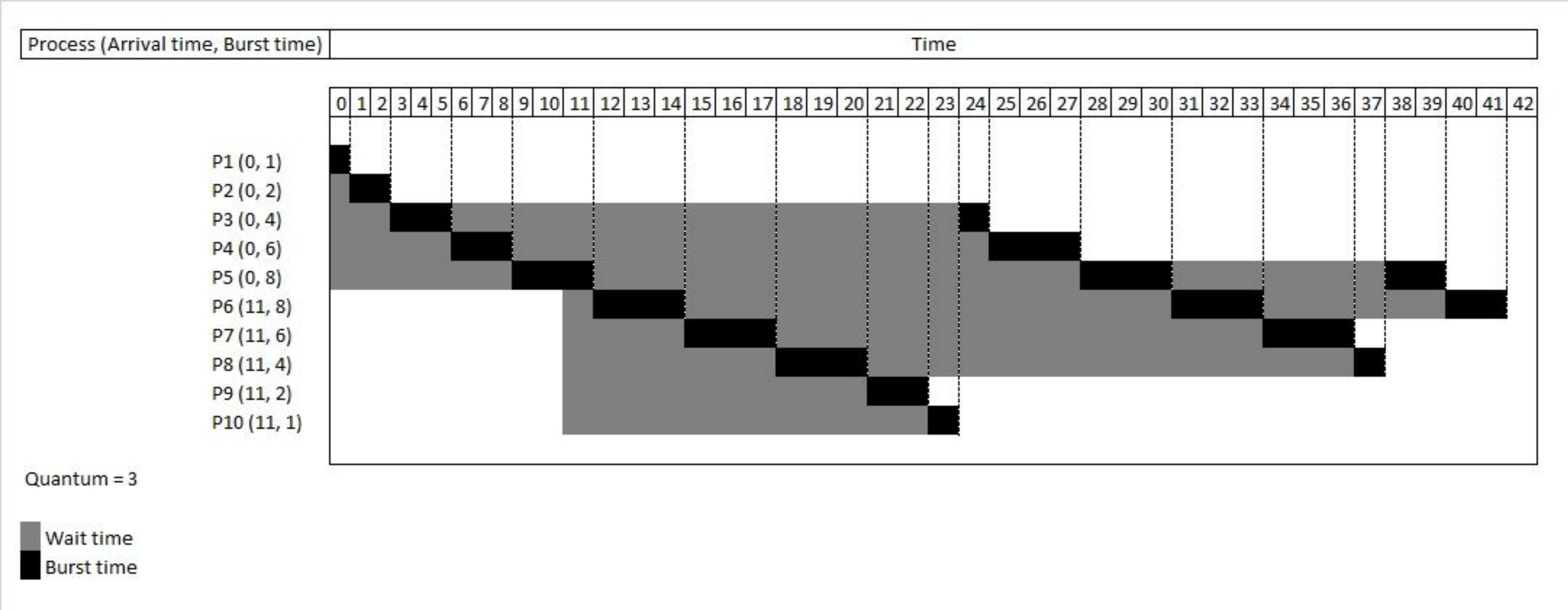
Fail-Safe Operation

- When a system fails you want to ensure the system enters into a safe condition
  - When a traffic light control system fails, revert to flashing red lights as the fail safe

Multitasking without an RTOS

- Round-Robin: similar to polled loop; tasks are allowed to run to completion; may use time-slicing (Special case: Polled Loop, poll inputs one by one)
- Round-Robin with interrupts (Hybrid): interrupts (foreground) for time-sensitive tasks, round-robin for mundane tasks (background)
- Interrupt-Driven: all tasks are inside ISRs

Round Robin


Process (Arrival time, Burst time)
Time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
P1 (0, 1)
P2 (0, 2)
P3 (0, 4)
P4 (0, 6)
P5 (0, 8)
P6 (11, 8)
P7 (11, 6)
P8 (11, 4)
P9 (11, 2)
P10 (11, 1)
Quantum = 3
Wait time
Burst time