

Real-Time Programming

Farhang Nemati
Örebro University
Spring 2016

1

Lecture 1

2

Real-Time Programming

- Teacher:
 - Farhang Nemati
 - Room T2232
 - Email: farhang.nemati@oru.se
- Lab Assistant :
 - Farhang Nemati
- Where to find course related stuff:
 - Blackboard

About Me

- BSc in Computer Engineering, University of Tehran, Iran, 1998
- Work in Industry, Iran, 1998-2003
- MSc in Computer Science, Uppsala University, Sweden, 2006
- PhD in Real-Time Systems, Mälardalen University, Sweden, 2012
- Software Engineer, Atlas Copco, Sweden, 2012-2013
- Software Specialist, BMW, Germany, 2013-now
- Senior Lecturer, Örebro University, Sweden, 2015-now



Lectures and Labs

- 14 lecture sessions
 - Each session 2 * 45 minutes
- 10 guided labs sessions
 - Each session 2 * 45 minutes
- Lab: T004
- 15 self-practice sessions. Lab T004 is booked for you during these sessions
- If you are not registered yet, talk to Jenny Tiberg as soon as possible!
- Find your group mate for labs now and register in one of the groups in Blackboard.

Criteria to Pass the Course

- Approved written exam
 - 4-6 questions, 40 points
 - Required: 20 points for grade 3, 30 for grade 4, and 35 for grade 5
- Approved labs
 - There are 4 labs
 - Required: all 4 labs are handed in, in time

Literature

- Real-Time Systems and Programming Languages. Alan Burns and Andy Wellings
- (In Swedish) Realtidsprogrammering, ISBN: 91-44-03130-0. Ola Dahl
 - Only electronic version exists
- (Optional) Hard Real-Time Computing Systems. Giorgio Buttazzo
- Lecture slides
- Lab instructions

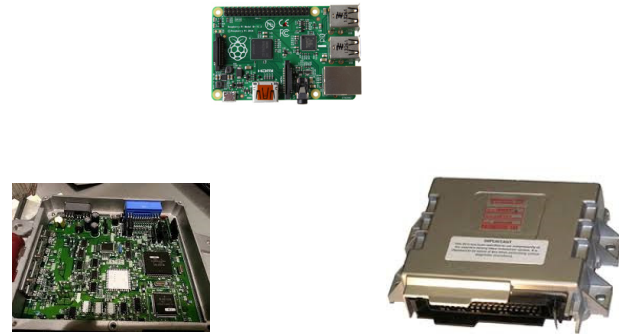
Preliminary Course Contents

- Course Introduction
 - Lab environment, programming tools, etc.
- Introduction
 - What are Embedded Systems?
 - What are Real-Time Systems?
- Programming for Embedded Systems and Real-Time Systems
 - Concurrent programming
 - Inter-process communication, Synchronization, etc.
 - Execution Time analysis
- Scheduling Algorithms
- Resource Sharing Protocols
- Design and Analysis of Real-Time Systems
 - Petri net
 - Response time analysis
- (Optional) Real-Time Systems on Multi-processors

Embedded Systems



Embedded Systems



Embedded Systems

- Interacting with physical world
- Long time running
- Concurrency
- Limited resources
- Limited access

Concurrency in Embedded Systems

- Increase reaction time to physical environment
- Parallelism, i.e., run on multiple processors
- Independence among different tasks, e.g., periodic execution of some tasks

Real-Time Systems

- Functional correctness as well as timing constraints have to be satisfied, e.g., deadline for doing some operation has to be met.



Types of Real-Time Systems

- Hard Real-Time System
 - Safety-Critical Systems
- Soft Real-Time Systems

Real-Time Systems

- Most of the embedded systems have timing constraints (Why?)
- Because of interaction with the physical world
- A Real-Time system is usually divided into tasks
 - Task?
- Execute tasks concurrently
- Who decides which task to run?
 - A scheduler
- How?
 - Fairness, priority, etc.

Real-Time Task

- A task is a set of operations that execute sequentially
 - A task can have a priority
 - Has an execution time: the total time it takes for the task to perform its operation
 - It might have a period (cycle)
 - It HAS a deadline: the latest time by which its execution has to be finished

Real-Time Task

- **Types of Real-Time Tasks:**
 - **Hard task:** its deadline should not be missed; missing a deadline might have severe or catastrophically consequences.
 - **Soft task:** missing deadline can be tolerated. Results after deadline can still be useful. However the performance could be degraded.
 - **Firm Task:** missing deadline can be tolerated. Results after deadline are useless.
 - A real-time system may contain a mixture of all types of tasks.

Execution of Real-Time Tasks

- **Monolithic**
 - Only one program. It runs all tasks sequentially
- **Real-Time Operating System (RTOS)**
 - An operating system suitable for real-time systems runs the tasks concurrently
 - E.g., VxWorks, RTLinux, FreeRTOS, etc.
- **Real-Time Language (RTL)**
 - A programming language suitable for real-time systems runs the tasks concurrently
 - E.g., Ada-95

Example1

- Assume we have a system that consists of 2 tasks which periodically perform some actions.

- **Monolithic solution:**

```
int mainProgram()  
{  
    while(1)  
    {  
        operationsOfTask1  
        operationsOfTask2  
        sleep(forSomeTime);  
    }  
}
```

Example1

- **RTOS solution:**

```
int mainProgram()  
{  
    createTask(task1, priority1);  
    createTask(task2, priority2);  
}  
void task1()  
{  
    while(1)  
    {  
        operationsOfTask1;  
        delay(forSomeTime);  
    }  
}  
void task2()  
{  
    while(1)  
    {  
        operationsOfTask2;  
        delay(forSomeTime);  
    }  
}
```

Example1

- RL solution, Ada-95:

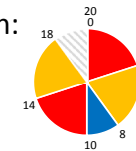
```
procedure mainProgram is
task TASK1;
... <declarations> ...
task body TASK1 is
begin
... < operationsOfTask1 > ...
end TASK1 ;
task TASK2
... <declarations> ...
task body TASK2
begin
... < operationsOfTask2 > ...
end TASK2;
begin
begin
null;
end mainProgram;
```

Example2; Car Control Tasks

- Assume we have 3 tasks that periodically control some parts of a car:
 - engine: runs every 10ms for 4ms
 - speed: runs every 10ms for 4ms
 - fuel: runs every 20ms for 2ms
- How do we know the **execution time** of each task?

- Each task has to finish its execution by end of its period (deadline)

- Monolithic solution:



- What happens if each task `engine` and `speed` takes 5ms to run?

Car Control Example, Monolithic Solution

```
//The source code in this example does not correspond to the syntax of any language!
int main()
{
    while(1)
    {
        //runs for 4 ms
        controlEngineOperations;
        //runs for 4 ms
        controlSpeedOperations;
        //runs for 2 ms
        controlFuelOperations;
        //runs for 4 ms
        controlEngineOperations;
        //runs for 4 ms
        controlSpeedOperations;

        delayMillisecond(2);
    }
}
```

Car Control Example, RTOS Solution

```
int controlEngine()
{
    while(1)
    {
        controlEngineOperations; //takes 4 ms
        delayMillisecond(6);
    }
}

int controlSpeed()
{
    while(1)
    {
        controlSpeedOperations; //takes 4 ms
        delayMillisecond(6);
    }
}
```

Car Control Example, RTOS Solution

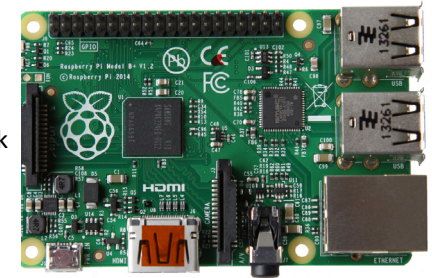
```
int controlFuel()
{
    while(1)
    {
        controlFuelOperations; //takes 2 ms
        delayMillisecond(18);
    }
}

int main()
{
    int engine, speed, fuel;
    engine = createTask(controlEngin, 10);
    speed = createTask(controlSpeed, 11);
    fuel = createTask(controlFuel, 12;

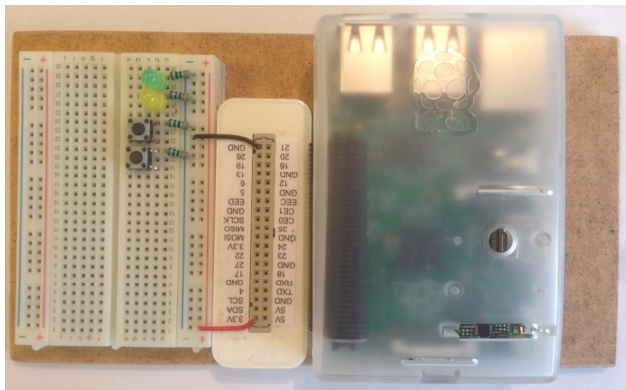
    waitForTasksToFinish(engine);
    waitForTasksToFinish(speed);
    waitForTasksToFinish(fuel);
}
```

Lab Equipment and Tools

- Raspberry Pi (Model B+)
 - 700MHz Broadcom BCM2835 CPU
 - 512 MB SDRAM @ 400MHz
 - 10/100 Ethernet RJ45 on-board network
 - Full size HDMI
 - 4 USB ports
 - Micro SD slot
 - 40 pins GPIO header

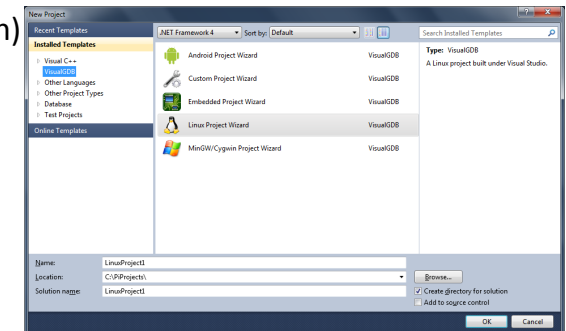


Lab Equipment and Tools



Development Environment

- Microsoft Visual Studio
- VisualGDB (Linux Edition)



Labs

- Lab 1: Hello World, Blinking LED
 - Familiarize yourself with programming and debugging the system
 - Using electronic devices, start with using LEDs
- Lab 2: Buttons, Breathing LEDs
 - Using Buttons
 - Pulse Width Modulation (PWM)
- Lab 3: Concurrency
 - Multiple tasks (threads), task priorities
 - Synchronization
- Lab 4: Car plate, Make a Car
 - 2 motors, buttons, lights, power supply etc.
 - Design of a complex system with concurrency constraints

Lecture 2

30

Properties of Real-Time Systems

- Complexity
 - Can vary from a small system with a few lines of code to several millions of code, e.g., automotive systems
- Reliability
 - Real-Time systems usually control some components in the physical world where a failure may have severe or catastrophic consequences, e.g., flight control system. They have to be reliable!
- Concurrency
 - Several things are done in parallel. The system is splitted into multiple tasks run in parallel.

Properties of Real-Time Systems

- Interactive with physical world
 - The system interacts with environment through sensors and actuators and special purpose hardware.
- Schedulability
 - The tasks have to finish their work in time. Each real-time task has a deadline, if all tasks could run in a way that none of them misses its deadline the system is schedulable. In hard real-time systems the schedulability of the system has to be guaranteed in advance.

Properties of Real-Time Systems

- Fault Tolerant
 - In the case of a fault the system should continue working, e.g., in a safe mode
- Predictability
 - The behavior of the system has to be known and deterministic in advance, i.e., before it is deployed on a real system.

Why Predictability is difficult to achieve?

- External events: Handling interrupts may not be deterministic; we don't know when the external events happen.
- Cache problems: knowing exact cache misses/hits in advance is not possible.
- Dynamic memory allocation methods are not deterministic. Static memory may not be sufficient.

Why Predictability is difficult to achieve?

- System calls are not deterministic.
- Priority inversion: a lower priority task runs while a higher priority task is waiting.
- To guarantee the schedulability of a real-time system the Worst Case Execution Time (WCET) of each task has to be known which is very difficult.

Why Concurrency?

- Distribute work among multiple tasks
- React to external events in time
- Decompose a system into functions that can run in parallel with each other.
- Run different operations independently
 - E.g., to be able to react to a sensor input in time, sensor acquisition has to be performed independently from other tasks.

Challenges with Concurrency

- Task communication
- Tasks may share resources/data
 - Mutually exclusive resources/data has to be protected to remain consistent.
- Developing concurrent programs is hard
- Debugging concurrent programs is much harder

Types of Task communication

- External communication
 - Interaction with environment, e.g., through sensors and actuators
- Synchronization on mutual exclusive resources
- Synchronization; task may wait for a condition to be fulfilled
- Data communication; tasks may send/receive data to/from other tasks.

Timing facilities that a RTOS has to provide

- Access to Clock, i.e., system time
 - A timer interrupts the processor in constant time intervals; jiffy. Each timer interrupt is called a tick
 - The clock rate (the number of ticks per second)
 - In Linux 2.4 a tick is 10ms (100 ticks/s; 100HZ) and in Linux 2.6 it is 1ms (1000 ticks/s; 1000HZ- 1kHz)
 - Any timing parameter of a task, e.g., deadline, has to be a multiplication of system tick to have exact precision
- Possibility for delaying the execution of a task for an interval of time

POSIX Standard

- POSIX = Portable Operating System Interface (for Unix)s
- A family of related standards
 - Provides standard Application Programming Interfaces (APIs) and command line shells and utilities for an operating system
 - Specified by IEEE
 - Facilitates developing applications portable to any operating system compatible with POSIX

POSIX Threads (PThreads)

- To work with threads (tasks) independently of language and operating system
- A thread has relatively many attributes
- The attributes of a thread are added to an attributes object (pthread_attr_t). The object is then used (passed as a parameter) when creating the thread.
 - For each attribute in the attributes object there are functions to get/set them

POSIX Threads (PThreads)

```
#include <pthread.h>
```

- Create a Pthread

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

- Wait for a Pthread to terminate

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- Cancel a Pthread

```
int pthread_cancel(pthread_t thread);
```

POSIX Threads (PThreads)

- Example; create a Pthread

```
void *start_function( void *funcArgs )  
{  
    struct *myArgs = funcArgs;  
    ...  
}  
struct threadArgs  
{  
    float d1;  
    int d2;  
    char *d3;  
    ...  
}  
void mainFunc()  
{  
    struct threadArgs args;  
    args.d1 = ...;  
    ...  
    pthread_t threadId;  
    int pthread_create(&threadId, NULL/*default attributes*/, start_function, (void *)&args);  
}
```

POSIX Threads (PThreads)

- Example; create a Pthread with attributes object

```
pthread_t threadId;  
  
pthread_attr_t attr;  
pthread_attr_init(&attr);  
pthread_attr_setstacksize(&attr, 4000);  
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);  
...  
  
int pthread_create(&threadId, &attr, start_function, (void *)&args);
```

POSIX Threads (PThreads)

- Get the Priority of a Pthread

```
int pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);  
policy: SCHED_FIFO, SCHED_RR, SCHED_OTHER  
Priority: param.sched_priority
```

- Set the Priority of a Pthread

```
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);
```

- SCHED_FIFO, SCHED_RR are Real-Time policies. The priority is only set with these policies. With other policies the priority has to be 0.

Delay a PThread

- sleep()

- The execution of the calling task (thread) is delayed for at least **sec** seconds

```
#include <unistd.h>  
  
unsigned sleep(unsigned sec)
```

Delay a PThread

- nanosleep()

- Delays the thread either for at least the duration of time specified in **rqtp* or until the delay is interrupted.

```
#include <time.h>
```

```
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);  
/*the time interval for which the task sleeps is put in rqtp*/
```

- The structure *timespec* is used to specify intervals of time with nanosecond precision:

```
struct timespec {  
    time_t tv_sec;        /* seconds */  
    long   tv_nsec;       /* nanoseconds */  
};
```

Car Control Example, Using PThreads

```
#include <pthread.h>  
#include <time.h>  
  
void *controlEngine(void *args)  
{  
    struct timespec ts1, ts2;  
    ts1.tv_sec = 0;  
    ts1.tv_nsec = 6000000; /*6 milliseconds*/  
    while(1){  
        /*controlEngineOperations here. We assume it takes 4 ms */  
        nanosleep(&ts1, &ts2);  
    }  
}
```

Car Control Example, Using PThreads

```
void *controlSpeed(void *args)
{
    struct timespec ts1, ts2;
    ts1.tv_sec = 0;
    ts1.tv_nsec = 6000000; /*6 milliseconds*/
    while(1){
        /*controlSpeedOperations here. We assume it takes 4 ms */
        nanosleep(&ts1, &ts2);
    }
}

void *controlFuel(void *args)
{
    struct timespec ts1, ts2;
    ts1.tv_sec = 0;
    ts1.tv_nsec = 18000000; /*18 milliseconds*/
    while(1){
        /*controlFuelOperations here. We assume it takes 2 ms */
        nanosleep(&ts1, &ts2);
    }
}
```

Car Control Example, RTOS Solution

```
int main()
{
    pthread_t engine, speed, fuel;
    struct sched_param eparam, sparam, fparam;
    eparam.sched_priority = 10;
    sparam.sched_priority = 11;
    fparam.sched_priority = 12;

    pthread_create(&engine, NULL, controlEngine, NULL);
    pthread_create(&speed, NULL, controlSpeed, NULL);
    pthread_create(&fuel, NULL, controlFuel, NULL);

    pthread_setschedparam(engine, SCHED_FIFO, &eparam);
    pthread_setschedparam(speed, SCHED_FIFO, &sparam);
    pthread_setschedparam(fuel, SCHED_FIFO, &fparam);

    pthread_join(engine, NULL);
    pthread_join(speed, NULL);
    pthread_join(fuel, NULL);
}
```

Lecture 3

POSIX Clocks and Timing Facilities

- **Clock:** A software entity to keep time, in seconds and nanoseconds
 - Is updated by system-clock ticks
 - It can be system-wide or local for a process (program)
 - `CLOCK_REALTIME` is a system-wide real-time clock. It represents seconds and nanoseconds since Epoch (1 January 1970 00:00:00)

• `#include <time.h>`

• **Get clock resolution**

```
clock_getres(clockid_t, struct timespec *)
```

Example:

```
struct timespec ts;
clock_getres(CLOCK_REALTIME, &ts);
// seconds in ts.tv_sec and nanoseconds in ts.tv_nsec
```

POSIX Clocks and Timing Facilities

- Get the current clock time

```
clock_gettime(clockid_t, struct timespec *)
```

Example:

```
struct timespec ts;  
clock_gettime(CLOCK_REALTIME, &ts);  
// seconds in ts.tv_sec and nanoseconds in ts.tv_nsec
```

POSIX Clocks and Timing Facilities

- Set the clock time

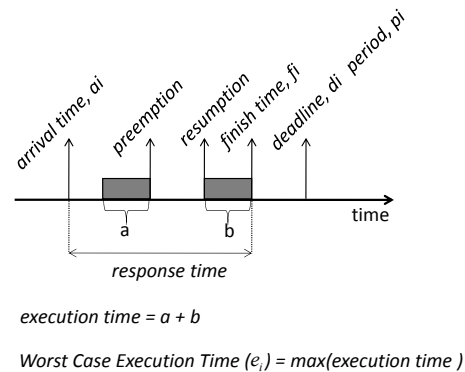
```
int clock_settime(clockid_t, const struct timespec *)
```

Example:

```
struct timespec ts;  
// put seconds in ts.tv_sec and nanoseconds in ts.tv_nsec  
ts.tv_sec = 100;  
ts.tv_nsec = 950;  
clock_settime(CLOCK_REALTIME, &ts);  
//Setting CLOCK_REALTIME requires appropriate privileges
```

Task Terminology

- Task parameters



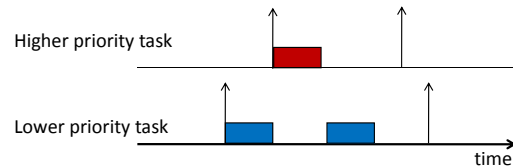
Task Model

- Task model includes a set of tasks with given assumptions.

$$T = \{\tau_1, \tau_2, \dots, \tau_n\},$$
$$\tau_i(e_i, \rho_i, d_i)$$

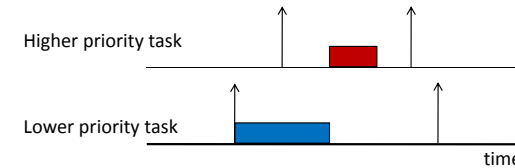
Preemptive, Non-preemptive Tasks

- **Preemptive:** A preemptive task can be interrupted by higher priority tasks at any time during its execution



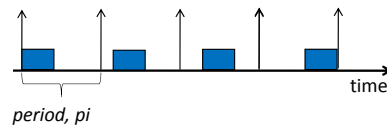
Preemptive, Non-preemptive Tasks

- **Non-preemptive:** a non-preemptive task cannot be interrupted by other tasks until end of its execution



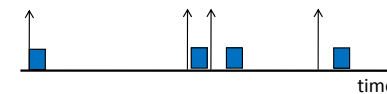
Periodic, Aperiodic, Sporadic Tasks

- **Periodic:** Execution of the task is repeated at constant time intervals. This constant interval is the period of the task



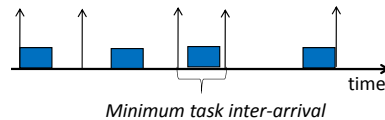
Periodic, Aperiodic, Sporadic Tasks

- **Aperiodic:** The task may execute at any time, e.g., in reaction to an external event.

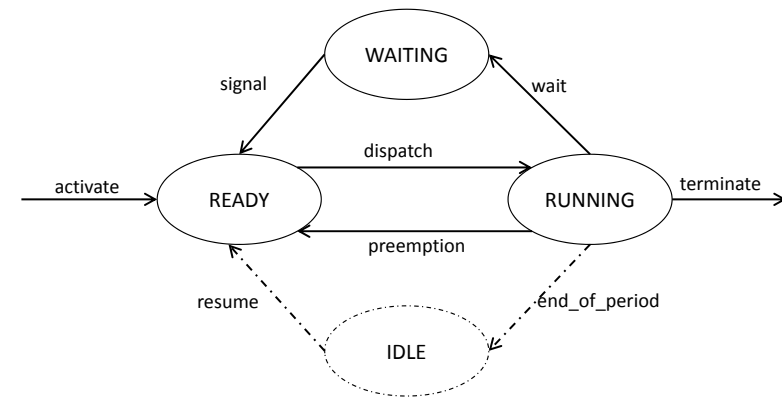


Periodic, Aperiodic, Sporadic Tasks

- **Sporadic**: An aperiodic task, however the time between two consecutive executions of the task shouldn't be less than a time interval, i.e., minimum time interval between two instances of the task.



Task States in RTOS



Task States in RTOS

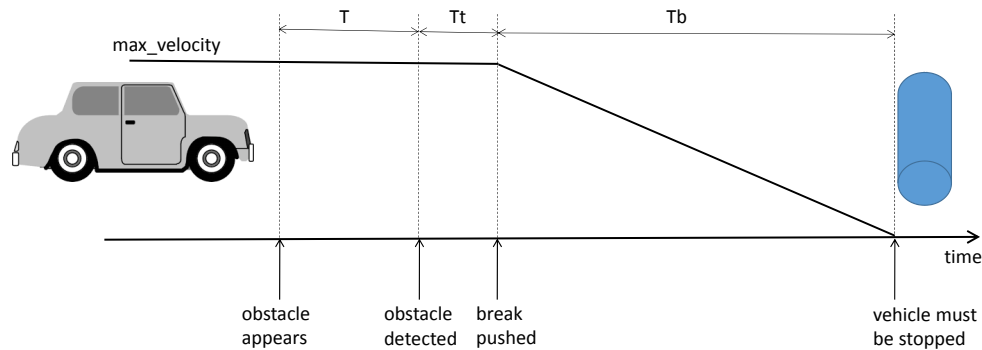
- **Ready**: When a task is ready to execute but it cannot execute because the processor is occupied by another higher priority task, it enters the Ready state. There could be multiple ready tasks, thus the RTOS puts these tasks in a *ready queue*.
- **Running**: When a task is assigned to the processor (starts executing), it enters Running state.
- **Waiting**: When a task during its execution has to wait for something that is not available at the moment, e.g., waits for some resource, data, a condition to be fulfilled, etc. In this case it enters Waiting state. When a task waits, for example on a resource, the RTOS puts the task into a queue associated with that resource.

Extracting Timing Restrictions

- Application requirements
- Physical World
 - Physical rules

Extracting Timing Restrictions

• Obstacle Avoidance System



Extracting Timing Restrictions

- The detection delay (period of sensor acquisition) is **T** time units; it should detect the obstacle and issue the brake command no later than T time units. Sampling period is less than T.
- Example:
 - The maximum velocity is 30 m/s (108 km/h)
 - The transition time, T_t, is 300 ms: the car will move for $0.3 \times 30 = 9$ meters during this time. This is the time it takes after sending the brake command until the brake is actuated.
 - After pushing the brake the car moves for 90 meters until it fully stops; the braking space = 90 meters

Extracting Timing Restrictions

- If the camera can first see the obstacle when it is 100 meters away from it, calculate the maximum time period for detection task.
- Solution:
 - The car moves for 9 meter during transmission
 - The maximum distance after which the brake command has to be sent equals to: $100 - (9 + 90) = 1$ meter. Thus the period of detection task should be less than $1/30 = 0.033$ s: **33 ms**

Task Synchronization and Communication

- Shared Variable Based
 - The objects in shared memory to which multiple tasks can have access
- Message Passing Based
 - Explicit exchange of data among tasks; directly or through some intermediate entity.

Shared Variable Based Synchronization and Communication

- Atomic (Indivisible) Operation
- Critical Sections
- Mutual Exclusion
- Condition Synchronization
- Busy Waiting
- Semaphores
- Mutexes
- Conditions
- Monitors
- Barriers

Shared Variable Based Synchronization and Communication

- Producer – Consumer

```
//Producer
while(1)
{
    x = calculate();
    globalVar = x;
}

//Consumer
while(1)
{
    y = globalVar;
    use(y);
}
```

Atomic (Indivisible) Operation

- An operation that from the rest of system's view is an Indivisible, uninterruptible, and instantaneously performed operation

```
x = x + 1;
```

```
MOVE X,D0
ADD #1,D0
MOVE D0,X
```

Shared Variable Based Synchronization and Communication

```
//Task1
while(1)
{
    x = x + 1;
    y = y + 1;
}

//Task2
while(1)
{
    x = x - 1;
    z = z + 1;
}
```

Critical Sections, Mutual Exclusion

- Critical Sections
 - A piece of code (sequence of statements) that no more than one task can execute at the same time.
- Mutual Exclusion
 - The synchronization needed to protect a critical section from being executed by more than one task.

Condition Synchronization

- A task stops proceeding until a condition is satisfied. Another task may fulfill the condition and let (signal) the waiting task know when the condition is satisfied.

```
//Task1          //Task2
...              ...
wait until flag = 1  falg = 1;
flag = 0;          signal Task1;
...                ...
```

Busy Waiting (Spinning)

- Is used for Condition Synchronization
- A task continuously (in a loop) checks a condition and proceeds when the condition is satisfied.

Busy Waiting

- Producer-Consumer

```
//Producer          //Consumer
while(1)             while(1)
{
    x = calculate();
    while (flag != 1)
    {
        //Just wait ...
    }
    flag = 0;
    globalVar = x;
}

                    {
                        while (flag != 0)
                        {
                            //Just wait ...
                        }
                        flag = 1;
                        y = globalVar;
                        use(y);
                    }
```

Busy Waiting

- Livelock: The tasks may stuck in a loop for condition checking and never exit the loop
- Inefficient: wasting processor with not useful work
- Developing and testing is difficult
- ...

Lecture 4

78

Busy Waiting

```
while (flag != 1)
{
    //Do nothing ...
}
```

- Livelock: The tasks may stuck in a loop for condition checking and never exit the loop
- Inefficient: wasting processor with not useful work
- Developing and testing is difficult

Exercise; Two Threads

- Create two threads. The first thread is supposed to increment a global integer and the second thread will put the odd values of the counter in a buffer. The program then checks if the buffer only contains odd numbers. Does the program have the results as expected?
- See the attached code (*twothread.c*)

Semaphores

- Used for synchronization
- Wraps a non-negative integer value
 - The value can only be accessed through 2 function calls
- It may have an internal queue in which tasks waiting for the semaphore are put

Semaphores

- Three operation are possible:
 - To create a semaphore with a value
 - To take/wait for the semaphore:
 - If the value is 0 the caller task suspends
 - If the value is not 0, the value is decremented by 1 and the caller proceeds
 - To give/signal the semaphore
 - The value is incremented by 1
 - Signals the tasks waiting (suspended) on the semaphore

Semaphores

- No need for busy waiting
- Binary Semaphores
 - Its value has only two different values; 0, 1
 - Used for cases with two cases, e.g., on/off, busy/free
- General (Counting) Semaphores
 - Its value can have any non-negative number
 - Suitable for handling resource sharing

Mutexes

- Similar to semaphore with a binary value
- Used for protecting mutual exclusion critical sections
- Suits for handling mutually exclusive resources shared among multiple tasks
- The task that takes a mutex, owns it

POSIX Semaphores

```
#include <semaphore.h>
```

- Initialize/open a semaphore

```
int sem_init(sem_t *sem, int pshared, unsigned value); //unnamed
sem_t *sem_open(const char *name, int oflag, ...); //named
```

- Take/wait for a semaphore

```
int sem_wait(sem_t *sem); //the caller task(thread) suspends if the semaphore is occupied
int sem_trywait(sem_t *sem); //the caller task(thread) continues even if the semaphore is occupied
```

- Give/Signal a semaphore

```
int sem_post(sem_t *sem);
```

- Delete a semaphore

```
int sem_destroy(sem_t *sem); //unnamed
int sem_unlink(const char *name); //named
```

POSIX Mutexes

- Like POSIX threads the attributes of a mutex are passed as an attributes object (pthread_mutexattr_t)
- There are functions to get/set each attribute of the attributes object

POSIX Mutexes

```
#include <pthread.h>
```

- Initialize a mutex

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Or

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

- Destroy a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

POSIX Mutexes

- Lock a mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex); //the caller task suspends if the mutex is locked
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex); //the caller task continues even if the mutex is locked
```

- Unlock a mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Exercise; Two Threads with Mutex

- Remove the inconsistency in the results
- See the attached code (*twothread_sync.c*)

Semaphores/Mutexes

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

//Task1
while(1)
{
    pthread_mutex_lock(&mtx);
    x = x + 1;
    pthread_mutex_unlock(&mtx);

    y = y + 1;
}

//Task2
while(1)
{
    pthread_mutex_lock(&mtx);
    x = x - 1;
    pthread_mutex_unlock(&mtx);

    z = z + 1;
}
```

Semaphores/Mutexes

- How many semaphores (or mutexes) are needed?

```
//Task1
while(1)
{
    x = x + 1;
    y = y + 1;
}

//Task2
while(1)
{
    x = x - 1;
    z = z + 1;
}

//Task3
while(1)
{
    x = x * 2;
    z = 0;
}
```

Semaphores/Mutexes

```
//Task1
while(1)
{
    sem_wait(&sem1);
    x = x + 1;
    sem_post(&sem1);

    y = y + 1;
}

//Task2
while(1)
{
    sem_wait(&sem1);
    x = x - 1;
    sem_post(&sem1);

    sem_wait(&sem2);
    z = z + 1;
    sem_post(&sem2);
}

//Task3
while(1)
{
    sem_wait(&sem1);
    x = x * 2;
    semGive(&sem1);

    sem_wait(&sem2);
    z = 0;
    sem_post(&sem2);
}
```

- Make critical sections as short as possible

Semaphores/Mutexes

- Producer-Consumer reading from/writing to a buffer

```
//Producer
while(1)
{
    x = calculate();

    pthread_mutex_lock(&mtx);
    buffer[index++] = x;
    pthread_mutex_unlock(&mtx);
}

//Consumer
while(1)
{
    pthread_mutex_lock(&mtx);
    y = buffer[--index];
    pthread_mutex_unlock(&mtx);
    use(y);
}
```

- What happens if Producer writes to the buffer when it is full? Or Consumer reads from the buffer when it is empty?

Semaphores/Mutexes; Exercise

- Improve the Producer-Consumer so that:
 - There is a maximum size for the buffer:
`int buffer[10];`
 - The producer waits (is blocked) if buffer is full
 - The consumer waits (is blocked) if the buffer is empty

Lecture 5

Semaphores/Mutexes; Solution

```
#include <pthread.h>
#include <semaphore.h>

const int MAX_BUF_SIZE = 10;
int buffer[MAX_BUF_SIZE];
void start()
{
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    sem_t bufFullSem, bufEmptySem;
    pthread_t prodThrd, consThrd;
    sem_init(&bufFullSem, 0, MAX_BUF_SIZE);
    sem_init(&bufEmptySem, 0, 0);
    pthread_create(&prodThrd, NULL, producer, NULL);
    pthread_create(&consThrd, NULL, consumer, NULL);
}
```

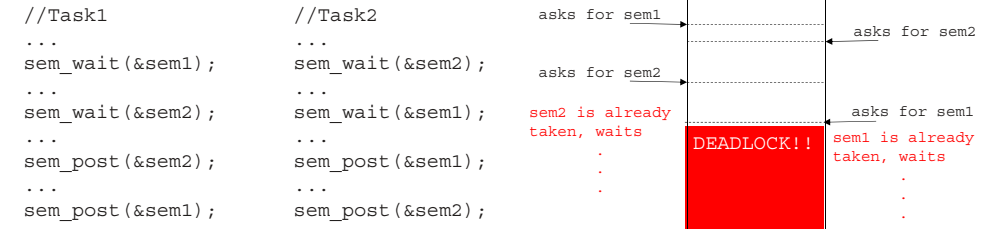

Semaphores/Mutexes; Solution

```
void *producer(void *arg)
{
    while(1)
    {
        x = calculate();
        sem_wait(&bufFullSem);
        pthread_mutex_lock(&mutex);
        buffer[index++] = x;
        pthread_mutex_unlock(&mutex);
        sem_post(&bufEmptySem);
    }
}

void *consumer(void *arg)
{
    while(1)
    {
        sem_wait(&bufEmptySem);
        pthread_mutex_lock(&mutex);
        y = buffer[--index];
        pthread_mutex_unlock(&mutex);
        use(y);
        sem_post(&bufFullSem);
    }
}
```

Problems with Semaphores and Mutexes

- Deadlock



- Task Starvation (Indefinite Postponement)
- Error prone: One mistake can take the whole system down.

Condition Variables

- In combination with a mutex
- Two Operations
 - Wait: Always waits (suspends) for a condition. Releases the associated mutex while suspending on the condition.
 - Signal: wakes up a waiting task. The waiting task is resumed and checks the condition again. Has no effect if no task is waiting for the condition.

POSIX Condition Variables

```
pthread_cond_t condition_cond = PTHREAD_COND_INITIALIZER;

pthread_cond_wait(pthread_cond_t*, pthread_mutex_t *);

pthread_cond_signal(pthread_cond_t*, pthread_mutex_t *);
```

Condition Variables

```
void start()
{
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

    pthread_t prodId, oddConsId, evenConsId;
    pthread_create(&prodId, NULL, producer, NULL);
    pthread_create(&oddConsId, NULL, oddConsumer, NULL);
    pthread_create(&evenConsId, NULL, evenConsumer, NULL);
}
```

Condition Variables

```
void *producer(void *arg)
{
    while(1)
    {
        x = calculate();
        pthread_mutex_lock(&mutex);
        buffer[index++] = x;
        pthread_cond_signal(&cond, &mutex);
        pthread_mutex_unlock(&mutex);
    }
}
```

Condition Variables

```
void *evenConsumer(void *arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        while(index == 0 || buffer[index]%2 != 0)
        {
            pthread_cond_wait(&cond, &mutex);
        }
        y = buffer[--index];
        pthread_mutex_unlock(&mutex);

        //Another consumer might be waiting ...
        pthread_cond_signal(&cond, &mutex);

        addEven(y);
    }
}
```

```
void *oddConsumer(void *arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        while(index == 0 || buffer[index]%2 == 0)
        {
            pthread_cond_wait(&cond, &mutex);
        }
        y = buffer[--index];
        pthread_mutex_unlock(&mutex);

        //Another consumer might be waiting ...
        pthread_cond_signal(&cond, &mutex);

        addOdd(y);
    }
}
```

Monitors

- All critical sections and operations needed for managing synchronization are encapsulated in modules that are called Monitors.
- Critical sections and variables to be accessed under mutual exclusion are hidden from the caller.

Monitor Example

```
const int MAX_SIZE = 100;
typedef struct Monitor
{
    pthread_mutex_t mutex;
    pthread_cond_t bufFullSem;
    pthread_cond_t bufEmptySem;
    int buffer[MAX_SIZE];
    int index = 0;
} MyMonitor;

void initMonitor(MyMonitor* monitor)
{
    monitor->mutex = PTHREAD_MUTEX_INITIALIZER;
    monitor->bufFullSem = PTHREAD_COND_INITIALIZER;
    monitor->bufEmptySem = PTHREAD_COND_INITIALIZER;
}
```

Monitor Example

```
void takeValue(MyMonitor* monitor, int* x)
{
    pthread_mutex_lock(&(monitor->mutex));
    while(monitor->index == 0)
    {
        pthread_cond_wait(&(monitor-> bufEmptySem), &(monitor->mutex));
    }
    *x = monitor->buffer[--monitor->index];
    pthread_mutex_unlock(&(monitor->mutex));
    pthread_cond_signal(&(monitor-> bufFullSem), &(monitor->mutex));
}

void addValue(MyMonitor* monitor, int x)
{
    pthread_mutex_lock(&(monitor->mutex));
    while(monitor->index == MAX_SIZE)
    {
        pthread_cond_wait(&(monitor->bufFullSem), &(monitor->mutex));
    }
    monitor->buffer[monitor->index++] = x;
    pthread_mutex_unlock(&(monitor->mutex));
    pthread_cond_signal(&(monitor->bufEmptySem), &(monitor->mutex));
}
```

Barriers

- A synchronization point for a group of tasks
- Every task in the group stops at the assigned barrier until all other tasks from the group are reached the barrier

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr, unsigned count);

int pthread_barrier_wait(pthread_barrier_t *barrier);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Lecture 6

Message Passing Based Synchronization and Communication

- Synchronization and communication among processes
- Thread/Tasks within a process can also use it
 - With threads, using shared variable synchronization and communication has higher performance

Message Passing Based Synchronization and Communication

- Synchronization Model
 - Asynchronous
 - Synchronous
 - Remote Procedure Call
- Naming of Source/Destination
 - Direct Naming
 - Indirect Naming
 - Symmetry
- The Structure of Message

Synchronization Model; Asynchronous Message Passing

- The sender continues; it does not wait for the message to be received
- Example: Posting a letter

```
...
async_send (message);
//does not wait ...
```

- + No need to wait
- + Program decomposition; decreases dependency
- - It might need too many buffers for keeping messages
- - More complex compared to synchronous
- - Testing and verification is more difficult

Synchronization Model; Synchronous Message Passing

- The sender can proceed only if the message has been received by the receiver
- Example: Phone call; the caller waits until the receiver is verified before giving any message

```
//sender                                //receiver
sync_send (message);                    sync_receive (message);
//waits until message is sent ...      //waits until message is received ...
```

- + Simple
- - Introduces dependency among tasks/objects
- - Tasks might wait too long

Synchronization Model; Remote Procedure Call (RPC)

- The sender (caller) waits for a reply from receiver
- Example: Phone call if the receiver can reply immediately

Naming of Source/Destination; Direct, Indirect Naming

- Direct: The sender directly names the receiver

```
//task1
send_to (task2, message);
```

- Indirect: The sender names an intermediate entity, e.g., socket, channel, mailbox

```
//task1
send_to (mail_box1, message);
```

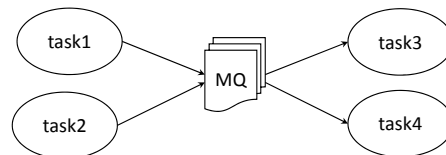
- Symmetry: Both sender and receiver name each other

```
//task1                      //task2
send_to (task2, message);    receive_from (task1, message);
```

Or:

```
//task1                      //task2
send_to (mail_box1, message); receive_from (mail_box1, message);
```

Message Queues



- A number of messages can be queued in a message queue
- Variable size of messages
- Tasks can send messages to a message queue or receive messages from it
- Multiple tasks can write to or read from same message queue

POSIX Message Queues

- Used for asynchronous message passing among tasks/processes
- The waiting tasks are queued in priority based manner
- The messages in a message queue can be prioritized
- The messages with the same priority level are queued in FIFO manner

POSIX Message Queues

```
#include <mqqueue.h>
```

- Open a message queue

```
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

- oflag (a bit mask):

- `O_RDONLY`: Can only receive messages from the queue
- `O_WRONLY`: Can only send messages to the queue
- `O_RDWR`: Can receive messages from the queue and send messages to it
- `O_CREAT`: Is together with two more arguments; `mode_t` and `mq_attr`. To create a message. It fails if `O_EXCL` is set and the queue already exists
- `O_EXCL`: If this flag is set and `O_CREAT` is not set undefined behavior!
- `O_NONBLOCK`: Whether the senders or receivers should block on the queue or not. Blocked tasks are waited in a priority based queue

POSIX Message Queues

- Open a message queue

```
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

- `mode` (bit mask): Access permissions like file access permissions in Linux, for example 666 means all users may read and write the file
- `attr` (attribute object): contains the attributes of the queue

POSIX Message Queues

- Message queue attributes

```
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

```
struct mq_attr {
    long mq_flags; /* Message queue description flags: for example O_RDWR */
    long mq_maxmsg; /* Maximum number of messages on the queue */
    long mq_msgsize; /* Maximum message size (in bytes) */
    long mq_curmsgs; /* Number of messages currently in queue */
};
```

POSIX Message Queues

```
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

- Create/Open message queue example:

```
mqd_t q1;
struct mq_attr q1Attr;
q1Attr.mq_msgsize = 20;
q1Attr.mq_maxmsg = 50;
q1 = mq_open("/myqueue", O_CREAT | O_WRONLY | O_NONBLOCK, 0700, &q1Attr);
```

- open an existing message queue

```
mqd_t q2;
q2 = mq_open("/myqueue", O_RDONLY | O_NONBLOCK); /* mode and attr not needed*/
```

POSIX Message Queues

- Retrieving attributes of a message queue

```
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

- Modifying message queue attributes

```
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr);
```

- The maximum message size and the maximum number of messages are not changeable

POSIX Message Queues

- Send a message to a message queue

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);
```

- Receive a message from a message queue

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);
```

POSIX Message Queues

- Close a message queue

```
int mq_close(mqd_t mqdes);
```

- Remove a message queue

```
int mq_unlink(const char *name);
```

- removes the queue and the queue is marked to be destroyed when there are no links to the queue

Message Queues; Exercise

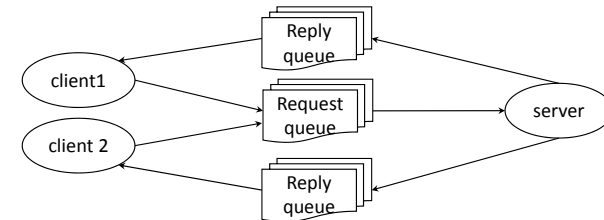
- Create two tasks; task1 and task2
- task1 creates a message queue which can contain at most 10 messages and the size of each message is at most 20 bytes.
- Every 100ms task1 generates a message and puts it into the queue. It should block if the queue is full
- task2 gets the messages from the queue and print it with period of 50ms. It should block if the queue is empty
- The task1 will send 100 messages and then it sends a final message (for example “stop”) and ends. task2 should end when it receives a stop message.
- Which task will be blocked mostly?

Message Queues; Solution

- See the attached program (msgqueue.c)

Client-Server Communication

- A special case of message passing based communication
- Clients issue requests to a server asking for some service
- Server accepts requests from clients and provides services to them
- Inter-task communication (usually message queues or pipes) are used to implement client-server communication



Client-Server Communication

- The server creates a request queue where clients can send their requests
- Each client creates its own reply queue. The client sends the queue id of its reply queue as a part of the request message. The server sends the reply to queue with this queue id.

Pipes

- An alternative for message queue
- A pipe has a read end and a write end
- Is treated as a (virtual) I/O device.
 - Uses I/O operations, e.g., the same operations that are used with files
 - `read()`, `write()` are used to read from, write to a pipe respectively
- Data written to the write end of a pipe can be read from the read end of the pipe

POSIX Pipes

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

- Create a pipe:

```
int pipe2(int pipefd[2], int flags);
```

- Creates a pipe which is a unidirectional data channel
- `pipefd[0]` refers to the read end of the pipe and `pipefd[1]` refers to the write end of the pipe

- Reading from and writing to a pipe:

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

- Close a pipe:

```
int close(int fd);
```

Sockets

- End points for communication over network
- Data streaming (TCP): Uses I/O operations, e.g., `read()`, `write()` are used to read/write.
- Data packets (UDP)

Signals

- Software interrupts
- There are 31 signals that a process or task can handle
- Mostly used for error and exception handling and not for normal communication
- To notify processes/tasks of occurrence of important events such as hardware exceptions, killing a process, etc.
- A routine is associated with a signal; if a signal is arrived the routine is executed
- There is a default routine for each signal, e.g., signal CTRL+C breaks the running process

Signals

- Signals can be disabled; the process will not receive them
- Signals received by a process is sent to one of its tasks that has enabled receiving of signals
- A task can send signals to
 - Itself
 - Any other task in its own process
 - Any system-wide public task
 - Its own process
 - Any other process in the system
- No history; no record of how many times a signal has arrived
 - POSIX queued signals record the history

Signals

- Associate a handler routine with a signal:

```
signal (int signalNumber, void(* function)())
```

- Send a signal to a task

```
int pthread_kill(pthread_t thread, int sig);
```

- Send a signal to the current task (itself)

```
raise (int signalNumber)
```

Signals

Example

```
void start()
{
    signal (SIGTERM, sighandler1);
    signal (SIGBUS, sighandler2);
}

void sighandler1(int sigNumber)
{
    ...
}

void sighandler2(int sigNumber)
{
    ...
}
```

Lecture 7

Basic Facilities Provided by a RTOS

- Timing Facilities
- Task Management
- Memory Management
- Error Handling
- I/O Services
- Interrupt Handling
- Task Synchronization and Communication
- Scheduling

Basic Services Provided by a RTOS; Timing Facilities

- Clocks, Timers
- Time resolution
- Handling the interrupt issued by system tick:
 - Store the information and state of the executing task
 - Update (virtual) timers
 - Wake up periodic tasks if any and if their period has passed
 - Call scheduler, i.e., sort the tasks in the ready queue
 - Load the information and state of the task at the top of ready queue and assign it to the processor

Basic Services Provided by a RTOS; Task Management

- Creating tasks
- Deleting Tasks
- API to change task attributes

Basic Services Provided by a RTOS; Memory Management

- Dynamic memory management
- No paging and virtual memory for real-time tasks
- Memory protection is often not provided; because of performance and determinism

Basic Services Provided by a RTOS; Error handling

- Deadlocks, missing deadlines, memory management errors, etc.
 - Error code of systems calls, e.g., error codes in errno.h (POSIX)
 - Exception handling
 - Fault tolerance

Basic Services Provided by a RTOS; I/O Services

- Files, devices, drivers
- Create/Remove
- Open/Close
- Read/Write

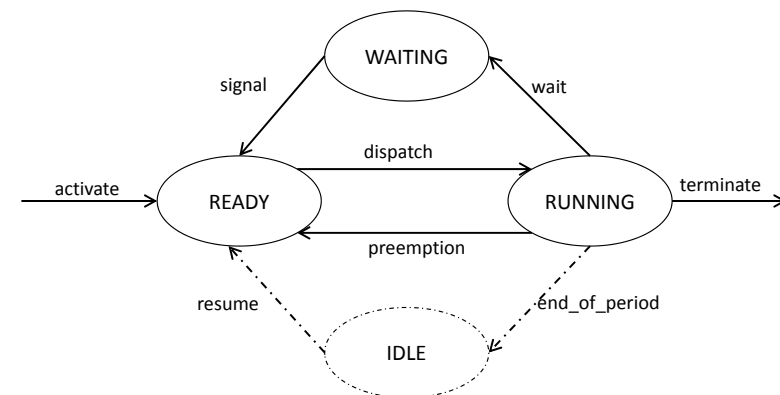
Basic Services Provided by a RTOS; Interrupt Handling

- An interrupt is usually used to communicate with hardware
- An interrupt is associated with an Interrupt Service Routine (ISR); the ISR is executed when the interrupt is arrived.
- In non real-time OS handling an interrupt is usually immediate to response to a device fast
 - In a RTOS this might interfere with hard real-time tasks
 - A task could be responsible for handling all interrupts

Basic Services Provided by a RTOS; Task Synchronization and Communication

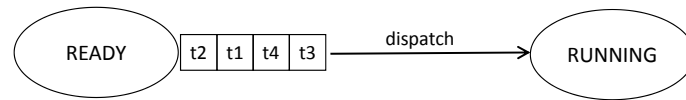
- Semaphores, mutexes
- Condition variables
- Monitors
- Message queues
- Signals

Real-Time Scheduling



Real-Time Scheduling

- How the tasks in the ready queue are ordered for getting the processor, i.e., transferring to Running
- The Scheduler provided by RTOS sorts the queue according to a scheduling algorithm



Real-Time Scheduling

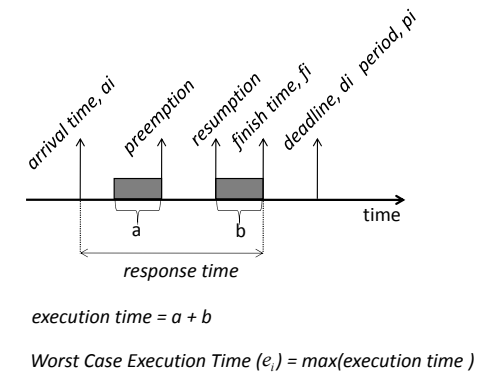
- Sort tasks in the ready queue based on:
 - Arrival time (FIFO)
 - Execution time
 - Priority
 - Deadline

Real-Time Scheduling

- Task Model
- Schedulability Analysis of a task set
- Scheduling Algorithms

Task Model

- Task parameters



Task Model

- A set of tasks
- Each task with specified parameters
 - Depends on whether the tasks are periodic, aperiodic, or sporadic

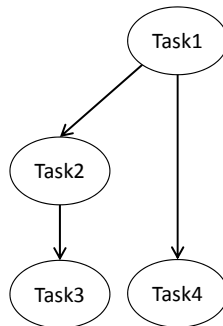
$$T = \{\tau_1, \tau_2, \dots, \tau_n\},$$
$$\tau_i(e_i, \rho_i, d_i, \dots)$$

Task Model

- Resource requirement constraints
 - Mutual exclusive resources
 - Resource sharing protocols are needed

Task Model

- Precedence constraints



Task Model; Aperiodic Tasks

- Each task is specified by 3 parameters:
 - Arrival time, a_i
 - Execution time, e_i
 - Deadline, d_i

$$\tau_i(a_i, e_i, d_i)$$

Task Model; Periodic Tasks

- Each task is specified following parameters:
 - Execution time, e_i
 - Deadline, d_i
 - Period, p_i
 - Utilization factor ($u_i = e_i / p_i$)

$$\tau_i(e_i, d_i, p_i)$$

Task Model; Sporadic Tasks

- Each task is specified by following parameters:
 - Execution time, e_i
 - Deadline, d_i
 - Minimum inter-arrival time, $e \min_i$

$$\tau_i(e_i, d_i, e \min_i)$$

Schedulability Analysis

- **Schedulability:** Given a task set and model, schedulability is to guarantee that all tasks will meet their deadline
- **Feasible Schedule:** Is a schedule that if the tasks are scheduled according to it, all tasks will meet their deadlines
 - There could be more than one feasible schedules
- **Schedulability Analysis:** Given a task set and model, to formally check whether there is any feasible schedule.

Schedulability Analysis

- Example, find a feasible schedule for following task set
 - with aperiodic model: $\tau_i(a_i, e_i, d_i)$
 - Assume tasks are independent (they don't share any resource other than processor)

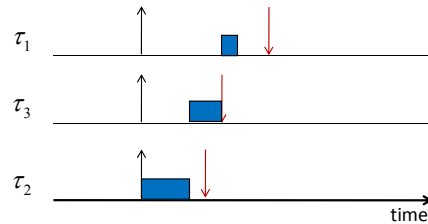
$$\tau_1(0, 1, 8)$$

$$\tau_2(0, 3, 4)$$

$$\tau_3(0, 2, 5)$$

Schedulability Analysis

- A feasible schedule: $\tau_2(0,3,4)$ $\tau_3(0,2,5)$ $\tau_1(0,1,8)$



- There could be more feasible schedules

Categories of Scheduling Algorithms

- Preemptive vs. Non-preemptive
- Static vs. Dynamic
- Offline vs. Online
- Optimal vs. Heuristic
- Time driven vs. Event driven

Preemptive vs. Non-preemptive

- **Preemptive Scheduling Algorithms:** A running task can be preempted by higher priority tasks at any time during its execution
- **Non-preemptive Scheduling Algorithms:** When a task is started it will run to its end without being preempted by other tasks. Any scheduling decision can only be done once the task is finished

Static vs. Dynamic

- **Static Scheduling Algorithms:** A static algorithm is based on parameters of tasks (e.g., priority, etc.) that are assigned to the tasks before their execution
- **Dynamic Scheduling Algorithms:** A dynamic algorithm is based on parameters of tasks that may change during run time, e.g., the priority of a task could be decided based on the current state ready queue

Offline vs. Online

- **Offline Scheduling Algorithms:** An algorithm is offline if all scheduling decisions are taken offline, i.e., before the system starts running
- **Online Scheduling Algorithms:** An algorithm is considered online if the scheduling decisions are taken during run-time based on task parameters

Optimal vs. Heuristic

- **Optimal Scheduling Algorithms:** An optimal algorithm is able to find a feasible schedule if there exists one, i.e., if an algorithm is optimal and it cannot find a feasible schedule no other algorithm can find a feasible schedule either
- **Heuristic Scheduling Algorithms:** A heuristic algorithm is guided by a heuristic function. Its goal is find a solution optimal or close to optimal, however it does not guarantee finding an optimal solution

Time driven vs. Event driven

- **Time driven Scheduling Algorithms:** Scheduling decisions are triggered based on system time, i.e., arrival times of tasks are known beforehand.
- **Event driven Scheduling Algorithms:** Scheduling decisions are triggered based on arrival of events, i.e., arrival of tasks are not known in advance. Schedulability cannot be guaranteed since the number of arrivals of tasks is not deterministic.

Lecture 8

Aperiodic Task Scheduling with Synchronous Arrival Times

- A set of aperiodic tasks that arrive at the same time
- Independent (No resource sharing)
- No precedence constraints
- Tasks are specified only by execution time and deadline:

$$\tau_i(e_i, d_i)$$

Aperiodic Task Scheduling; Earliest Due Date (EDD) – Jackson's Algorithm

- **Algorithm:** Execute tasks according to their non-decreasing deadlines. E.g., the task with the earliest deadline is executed first.
- Since all tasks arrive at the same time, preemption is not an issue

EDD; Example1

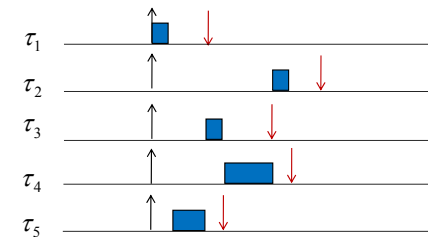
- Schedule the tasks according to EDD algorithm

	e_i	d_i
τ_1	1	3
τ_2	1	10
τ_3	1	7
τ_4	3	8
τ_5	2	5

EDD; Example

- The feasible schedule found by EDD

$\tau_1(1,3) \tau_5(2,5) \tau_3(1,7) \tau_4(3,8) \tau_2(1,10)$



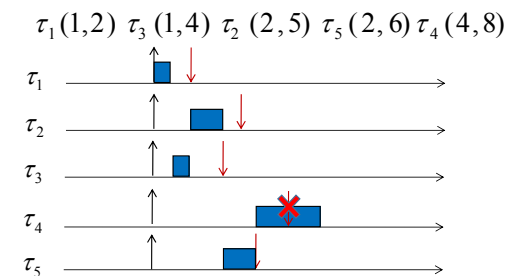
EDD; Example2

- Schedule the tasks according to EDD algorithm

	e_i	d_i
τ_1	1	2
τ_2	2	5
τ_3	1	4
τ_4	4	8
τ_5	2	6

EDD; Example

- Schedule according to EDD



EDD

- EDD is optimal
- EDD minimizes the maximum lateness of a task set even if it cannot schedule it
 - Lateness of task τ_i denoted by L_i : $L_i = f_i - d_i$
 - Maximum lateness of a task set:

$$L_{\max} = \max(L_i)$$

EDD Schedulability

- A task set is schedulable if:

$$\forall \tau_i = \tau_1 \dots \tau_n \quad f_i \leq d_i$$

- Suppose H_i denotes all tasks with their priority higher than or equal to τ_i

$$f_i = \sum_{\tau_k \in H_i} e_k$$

- Thus:

$$\forall \tau_i = \tau_1 \dots \tau_n \quad \sum_{\tau_k \in H_i} e_k \leq d_i$$

Aperiodic Task Scheduling with Arbitrary Arrival Times

- Preemptive Earliest Deadline First (EDF) - Horn's algorithm
- Tasks are preemptive
- Tasks can arrive at **arbitrary** time
- Tasks are specified as follows:

$$\tau_i(a_i, e_i, d_i)$$

Aperiodic Task Scheduling; Preemptive EDF

- Similar to EDD
- Independent (No resource sharing)
- No precedence constraint
- **Algorithm**: Any time a task arrives, sort tasks according to their non-decreasing deadlines and execute the task with the earliest deadline

Preemptive EDF

- EDF is optimal
- EDF minimizes the maximum lateness of a task set even if it cannot schedule it

$$L_{\max} = \max(L_i)$$

Preemptive EDF Schedulability

- At any time instant t:

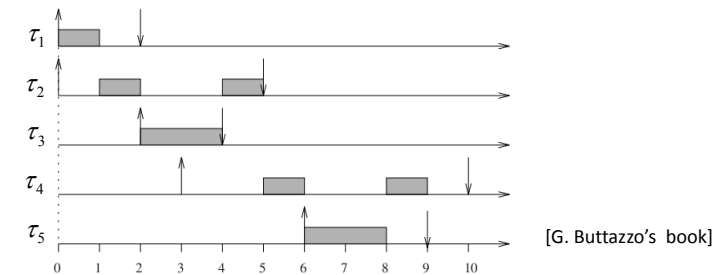
$$\forall \tau_i = \tau_1 \dots \tau_n \quad \sum_{\tau_k \in H_i} e_k \leq d_i$$

Preemptive EDF; Example

- Schedule the tasks according to Preemptive EDF algorithm

	a_i	e_i	d_i
τ_1	0	1	2
τ_2	0	2	5
τ_3	2	2	4
τ_4	3	2	10
τ_5	6	2	9

Preemptive EDF; Example



Aperiodic Scheduling; Least Slack Time First (LST)

- Slack time of a task: $s_i = d_i - e_i$
- Algorithm:** At any time a task arrives, order the tasks according to their non-decreasing slack times and execute the task with minimum slack time
- Objectives**
 - There is no gain in finishing a task much earlier than its deadline. As long as it does not miss its deadline its execution can be postponed
 - Soft tasks can have more chance to execute.

Non-preemptive EDF

- Tasks are non-preemptive
- Independent (No resource sharing)
- No precedence constraint
- Algorithm:** Whenever a task is finished order tasks according to their non-decreasing deadlines and execute the task with the earliest deadline until it is finished

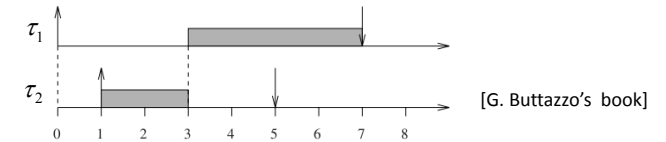
Non-preemptive EDF

- Non-preemptive EDF is **Not** optimal
- Example:

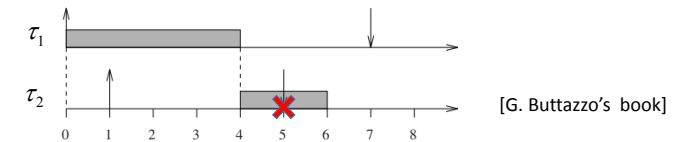
	a_i	e_i	d_i
τ_1	0	4	7
τ_2	1	2	5

Non-preemptive EDF; Example

- A feasible schedule:

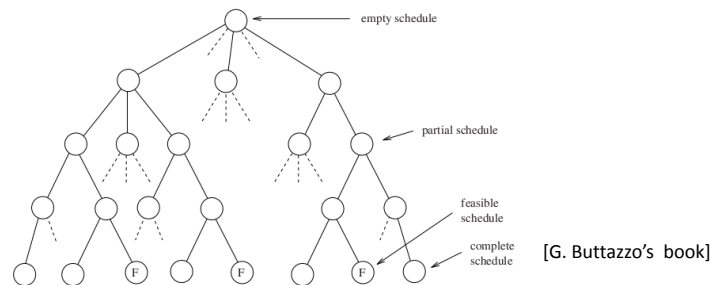


- Schedule according to Non-preemptive EDF:



Aperiodic Non-preemptive Scheduling

- If the arrival times are known in advance for a non-preemptive scheduling, a branch-and-bound algorithm can be used:



Aperiodic Non-preemptive Scheduling

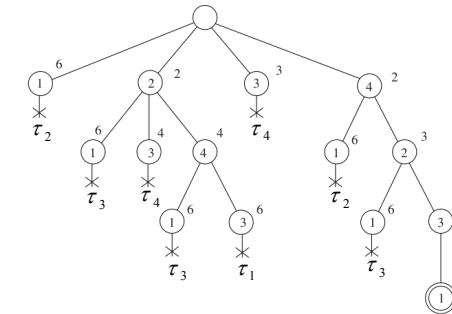
- To find a feasible schedule in the worst case an exhaustive search might be performed
 - Assuming n tasks, $n.n!$ combinations will be checked
 - Too expensive; not practical if the number of tasks is high.

Aperiodic Non-preemptive Scheduling; Bratley's Algorithm

- **Algorithm:** When searching in the tree a branch skipped whenever:
 - Adding a new task (node) causes an infeasible schedule
 - A feasible schedule is found
- The algorithm is simple and easy and in average case it's effective
- The worst case can still take $n.n!$ checks
- Not practical for online scheduling if the number of tasks are high

Bratley's Algorithm; Example

	a_i	e_i	d_i
τ_1	4	2	7
τ_2	1	1	5
τ_3	1	2	6
τ_4	0	2	4



Number in the node : the task scheduled
Number next to node: finishing time

[G. Buttazzo's book]

Aperiodic Non-preemptive Scheduling; The Spring Algorithm

- A Heuristic algorithm
 - Adding a task to a branch is guided by a heuristic function
- The objective of the algorithm is to find a feasible schedule for a task set with different constraints, e.g., non-preemptive, precedence constraints, arbitrary arrival times, resource sharing, etc.
- **Algorithm:** When searching in the search tree and adding a new task (node) to the tree: Add a task whose heuristic function has the minimum value among other tasks

The Spring Algorithm

- Heuristic function examples

$$H(\tau_i) = a_i$$

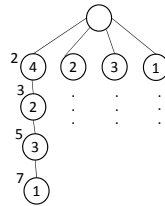
$$H(\tau_i) = d_i$$

$$H(\tau_i) = e_i$$

The Spring Algorithm; Example

	a_i	e_i	d_i
τ_1	4	2	7
τ_2	1	1	5
τ_3	1	2	6
τ_4	0	2	4

$$H(\tau_i) = d_i$$

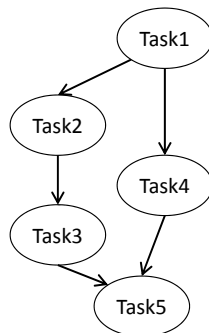


Aperiodic Scheduling; Summary of Presented Algorithm

Same Arrival Times	Different Arrival Times	
	Preemptive	Non-preemptive
•EDD	•EDF •LST	•Bratley •Spring

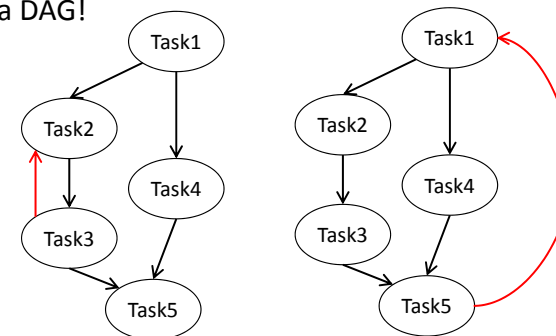
Scheduling with Precedence Constraints

- The precedence constraints are represented with Directed Acyclic Graph (DAG)



Scheduling with Precedence Constraints

- Not a DAG!



Scheduling with Precedence Constraints

- With same arrival times
- With arbitrary arrival times

Scheduling with Precedence Constraint; Latest Deadline First (LDF)

- With same arrival times
- **Algorithm:** Adding a task to a schedule queue:
 - Select from tail to head
 - Tasks without successors or those whose successors are selected. Select the task with the latest deadline

Lecture 9

Scheduling with Precedence Constraints

- With same arrival times
- With arbitrary arrival times

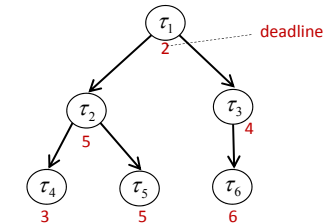
Scheduling with Precedence Constraints; Latest Deadline First (LDF)

- With same arrival times
- **Algorithm:**
 - Adding a task to a schedule queue:
 - Select from **tail to head**
 - Tasks without successors or those whose successors are selected, select the task with the latest deadline
 - To schedule then will be the reverse order of the queue; the task from the end of the queue will run first.
- Optimal

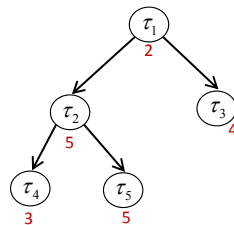
LDF; Example

- Schedule the following task set with LDF

	e_i	d_i
τ_1	1	2
τ_2	1	5
τ_3	1	4
τ_4	1	3
τ_5	1	5
τ_6	1	6

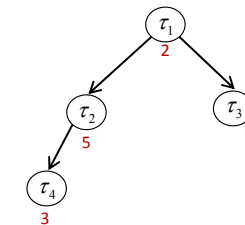


LDF; Example



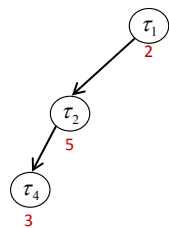
queue: τ_6

LDF; Example



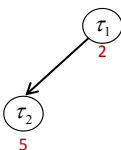
queue: $\tau_6 \tau_5$

LDF;
Example



queue: $\tau_6 \ \tau_5 \ \tau_3$

LDF;
Example



queue: $\tau_6 \ \tau_5 \ \tau_3 \ \tau_4$

LDF;
Example



queue: $\tau_6 \ \tau_5 \ \tau_3 \ \tau_4 \ \tau_2$

LDF;
Example

	e_i	d_i
τ_1	1	2
τ_2	1	5
τ_3	1	4
τ_4	1	3
τ_5	1	5
τ_6	1	6

queue: $\tau_6 \ \tau_5 \ \tau_3 \ \tau_4 \ \tau_2 \ \tau_1$ **➡** **Schedule:** $\tau_1 \ \tau_2 \ \tau_4 \ \tau_3 \ \tau_5 \ \tau_6$ **It's feasible!**

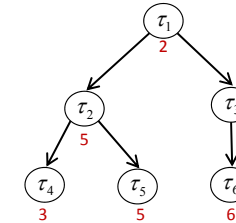
Scheduling with Precedence Constraints; with Same Arrival Times

- EDF
- **Algorithm:** Adding a task to a schedule queue:
 - Select from **root**
 - Among tasks that are without predecessor or their predecessors are already selected, select the task with the earliest deadline

EDF; Example

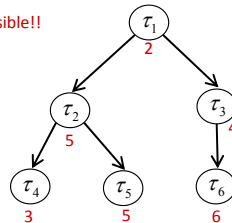
- Schedule the following task set with EDF

	e_i	d_i
τ_1	1	2
τ_2	1	5
τ_3	1	4
τ_4	1	3
τ_5	1	5
τ_6	1	6



EDF; Example

Schedule: $\tau_1 \ \tau_3 \ \tau_2 \ \tau_4 \ \tau_5 \ \tau_6$ NOT Feasible!!

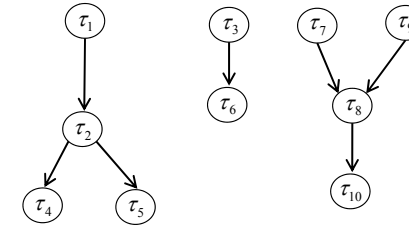


EDF with Precedence Constraints

- EDF with precedence constraints with same or arbitrary arrival times is **NOT** optimal!
- What to do?
 - Bratley's Algorithm
 - Spring Algorithm
- There is a better way

Scheduling with Precedence Constraints; with Arbitrary Arrival Times

- **EDF*** Algorithm
- Transform the dependent set of tasks into an independent task set by:
 - Modifying Arrival Times
 - Modifying Deadlines



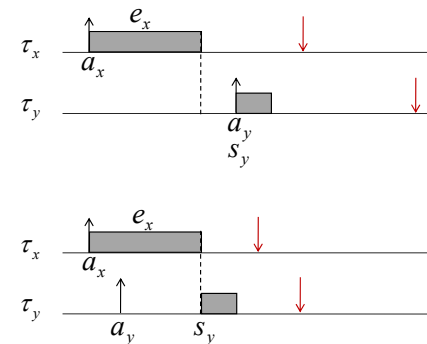
EDF*; Modification of Arrival Times

- Given two tasks τ_x and τ_y where $\tau_x \rightarrow \tau_y$ meaning that τ_y is an immediate successor of τ_x
 - The start time of τ_y (denoted by s_y) can not be earlier than its arrival time
 - τ_x should have enough time to finish before τ_y can start; s_y can not be earlier than minimum finishing time needed for τ_x

$$s_y \geq a_y$$

$$s_y \geq a_x + e_x$$

EDF*; Modification of Arrival Times



EDF*; Modification of Arrival Times

- The new arrival of task τ_y (start time) denoted by a^*_y is calculated as follows:

$$a^*_y = \max(a_y, a_x + e_x)$$

EDF*; Modification of Arrival Times

- The algorithm for modifying all arrival times
 - For each initial node in DAG set the arrival time

$$a^*_i = a_i$$

- Select a task τ_j whose arrival time is not yet modified but the arrival times of all its immediate predecessors have been modified. If such task does not exist, exit the algorithm
- Modify the arrival time of τ_j

$$a^*_j = \max(a_j, \max(a^*_h + e_h : \tau_h \rightarrow \tau_j))$$

- Continue from step 2

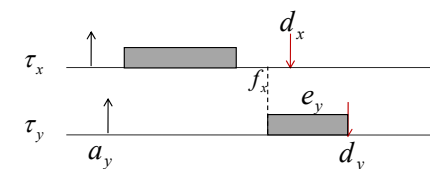
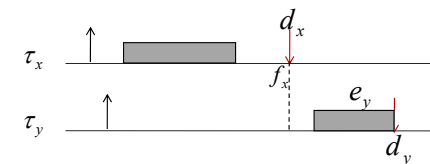
EDF*; Modification of Deadlines

- Given two tasks τ_x and τ_y where $\tau_x \rightarrow \tau_y$ meaning that τ_y is an immediate successor of τ_x
 - The finishing time of τ_x can not be later than its deadline
 - τ_y should have enough time to finish after τ_x is finished; f_x can not finish later than the latest time that τ_y can start

$$f_x \leq d_x$$

$$f_x \leq d_y - e_y$$

EDF*; Modification of Arrival Times



EDF*; Modification of Deadlines

- The new deadline of task τ_x (start time) denoted by d^*_x is calculated as follows:

$$d^*_x = \min(d_x, d_y - e_y)$$

EDF*; Modification of Deadlines

- The algorithm for modifying all deadlines
 - For each terminal node in DAG set the deadline

$$d^*_i = d_i$$
 - Select a task τ_j whose deadline is not yet modified but the deadlines of all its immediate successors have been modified. If such task does not exist, exit the algorithm
 - Modify the deadline of τ_j

$$d^*_j = \min(d_j, \min(d^*_k - e_k : \tau_j \rightarrow \tau_k))$$
 - Continue from step 2

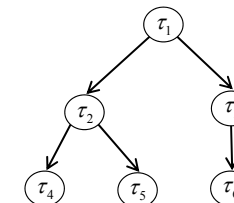
EDF*

- After modifying all arrival times and deadlines the task set has been transformed into a task set where there is no precedence constraints anymore, i.e., the tasks with the new arrival times and deadlines are independent.
- Now EDF can be used with the transformed task set

Modify Arrival Times and Deadlines; Example

- Schedule the following task set with EDF

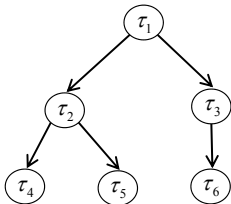
	a_i	e_i	d_i
τ_1	0	1	2
τ_2	1	1	5
τ_3	0	1	4
τ_4	2	1	3
τ_5	1	1	5
τ_6	0	1	6



Modify Arrival Times and Deadlines; Example

• Schedule the following task set with EDF

	a_i	e_i	d_i	a^*_i	d^*_i
τ_1	0	1	2	0	1
τ_2	1	1	5	1	2
τ_3	0	1	4	1	4
τ_4	2	1	3	2	3
τ_5	1	1	5	2	5
τ_6	0	1	6	2	6



Aperiodic Scheduling Algorithms; Summary

	Same Arrival Times	Different Arrival Times	
		Preemptive	Non-preemptive
Independent	•EDD	•EDF •LST	•Bratley •Spring
Precedence Constraints	•LDF	•EDF*	•Spring

Lecture 10

Periodic Task Scheduling

- A set of periodic tasks
- Independent (No resource sharing)
- No precedence constraints
- A periodic task is repeated in specific rate
 - Sensor acquisition
 - Monitoring
 - Control loops

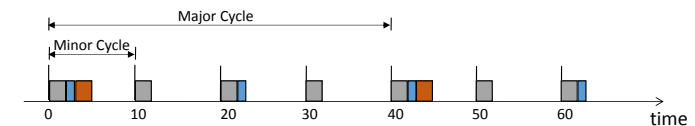
Periodic Task Scheduling; Timeline Algorithm (Cyclic Executive)

- Is used widely
- Simple and easy
- **Algorithm:** A global loop in equal iterates in equal time slots and in each time slot one or more tasks are executed

Timeline Algorithm (Cyclic Executive)

- Example

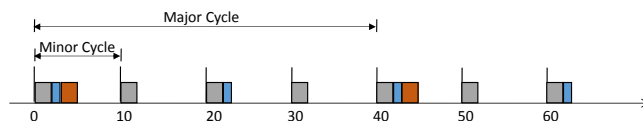
	e_i	p_i
τ_1	2	10
τ_2	1	20
τ_3	2	40



Major Cycle = Least Common Multiply (LCM) of task periods

Timeline Algorithm (Cyclic Executive); Schedulability Test

- A task set could be schedulable if: For each minor cycle the summation of execution times of tasks executing during that cycle \leq length of Minor Cycle



Timeline Algorithm (Cyclic Executive)

- + Simple.
 - A main program within each interval equal to the minor cycle will execute the appropriate tasks. This is repeated at each interval equal to the major cycle.
- + Sequential
 - No context switch
 - No need to protect shared data, resources, etc.
- - Not flexible; if the period or execution time of a task changes the whole schedule might need to be changed
 - If the periods are not multiple of each other the time table could be too big, e.g., major cycle for tasks with periods 7, 11, 27 is $7 \cdot 11 \cdot 27 = 2079$
- - If a task is malfunctioning it will affect other tasks

Fixed-Priority Scheduling Algorithms

- A fixed priority is assigned to each task offline
- The tasks are scheduled according to their fixed priorities
- Static (priority is fixed) online scheduling

Rate Monotonic Scheduling Algorithm (RMS)

- The rule for assigning priorities to the periodic tasks: Assign priorities to tasks according to their arrival rate; the shorter the period of a task is the higher its priority will be.
 - E.g., the task with the shortest period gets the highest priority and the task with the longest period will get the lowest priority.
- Online static (fixed priority) scheduling algorithm
- Among fixed-priority preemptive algorithms RMS is optimal

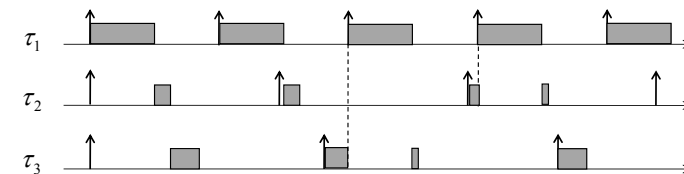
Rate Monotonic Scheduling Algorithm (RMS)

- Task Model
 - Independent tasks
 - Preemptive
 - Tasks are specified as follows:

$$\tau_i(e_i, p_i)$$

Rate Monotonic Scheduling Algorithm (RMS);

- Example



$$\rho_1 = 1, \rho_2 = 2, \rho_3 = 3$$

RMS

- Processor Utilization Factor, U:

$$U = \sum_{i=1}^n \frac{e_i}{p_i}$$

- Example

	e_i	p_i
τ_1	4	10
τ_2	6	20
τ_3	9	40

$$U = \frac{4}{10} + \frac{6}{20} + \frac{9}{40} = 92.5\%$$

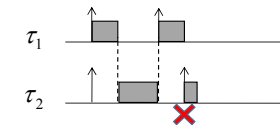
RMS; Schedulability Test

- $U < 1$ does not necessarily mean the task set is schedulable

- Example

	e_i	p_i
τ_1	2	5
τ_2	4	7

$$U = \frac{2}{5} + \frac{4}{7} \approx 97.14\%$$



RMS; Schedulability Test

- Assuming n is the number of tasks of a periodic task set, the task set is schedulable if:

$$U \leq n \times (2^{1/n} - 1)$$

- The upper bound is only dependent on the number of tasks

RMS; Schedulability Test

- The test is sufficient but not necessary. Let $U_{\text{lub}} = n \times (2^{1/n} - 1)$:
 - If $U \leq U_{\text{lub}}$ the task set is schedulable
 - If $U > 1$ the task set is not schedulable
 - If $U_{\text{lub}} \leq U \leq 1$ **no conclusion!**

RMS; Schedulability Test

- Example of upper (U_{lub}) bounds for RMS

n	U_{lub}
1	1
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
7	0.729
∞	≈ 0.693

RMS; Schedulability Test

- Shortest repeating cycle = Least Common Multiple (LCM) of periods
- We can test the schedule in LCM
 - Too difficult when the number of tasks is high
- It's enough if we only test the Critical Instant
 - **Critical Instant** of a **task**: The instant of a task where the task has worst response time
 - **Critical Instant** of a **task set**: The instant where all tasks arrive at the same time. The worst case scenario happens in the critical instant, i.e., all tasks have their worst response time [M. Joseph and P. Pandya, 1986]

RMS; Response Time

- Calculating Maximum Response Times
- Let denote response time of task τ_i with R_i and denote the set of tasks having priority higher than τ_i with H_i

$$R_i = e_i + \sum_{\tau_j \in H_i} \left\lceil \frac{R_i}{p_j} \right\rceil e_j$$

RMS; Exact Schedulability Test

- **Response Time Analysis**: For each task τ_i
 - 1) $R^{(0)}_i = e_i + \sum_{\tau_j \in H_i} e_j$
 - 2) $R^{(k+1)}_i = e_i + \sum_{\tau_j \in H_i} \left\lceil \frac{R^k_i}{p_j} \right\rceil e_j$
 - Using numerical calculations we continue the equation in step 2 until it either reaches a fix value (does not increase anymore) or it exceeds the deadline (d_i)
 - If the final response time exceeds the deadline the task is unschedulable
 - If the response time reaches a fix value before it exceeds the deadline the task is schedulable. This value is the maximum response time of the task.

RMS; Response Time Analysis

• Example

	e_i	p_i
τ_1	1	3
τ_2	1	4
τ_3	2	6
τ_4	1	20

$$R^{(1)}_1 = R^{(0)}_1 = e_1 = 1$$

RMS; Response Time Analysis

• Calculate R_2

- Step 0: $R^{(0)}_2 = e_2 + e_1 = 1 + 1 = 2$
- Step 1: $R^{(1)}_2 = e_2 + \left\lceil \frac{R^{(0)}_2}{p_1} \right\rceil e_1 = 1 + \left\lceil \frac{2}{3} \right\rceil 1 = 2$
- $R^{(1)}_2 = R^{(0)}_2 = 2 \leq d_2 = 4 \Rightarrow \tau_2$ is schedulable

	e_i	p_i
τ_1	1	3
τ_2	1	4
τ_3	2	6
τ_4	1	20

RMS; Response Time Analysis

• Calculate R_3

- Step 0: $R^{(0)}_3 = e_3 + e_2 + e_1 = 2 + 1 + 1 = 4$
- Step 1: $R^{(1)}_3 = e_3 + \left\lceil \frac{R^{(0)}_3}{p_1} \right\rceil e_1 + \left\lceil \frac{R^{(0)}_3}{p_2} \right\rceil e_2 = 2 + \left\lceil \frac{4}{3} \right\rceil 1 + \left\lceil \frac{4}{4} \right\rceil 1 = 5$
- Step 2: $R^{(2)}_3 = e_3 + \left\lceil \frac{R^{(1)}_3}{p_1} \right\rceil e_1 + \left\lceil \frac{R^{(1)}_3}{p_2} \right\rceil e_2 = 2 + \left\lceil \frac{5}{3} \right\rceil 1 + \left\lceil \frac{5}{4} \right\rceil 1 = 6$
- Step 3: $R^{(3)}_3 = e_3 + \left\lceil \frac{R^{(2)}_3}{p_1} \right\rceil e_1 + \left\lceil \frac{R^{(2)}_3}{p_2} \right\rceil e_2 = 2 + \left\lceil \frac{6}{3} \right\rceil 1 + \left\lceil \frac{6}{4} \right\rceil 1 = 6$

	e_i	p_i
τ_1	1	3
τ_2	1	4
τ_3	2	6
τ_4	1	20

RMS; Response Time Analysis

- $R^{(3)}_3 = R^{(2)}_3 = 6 \leq d_3 = 6 \Rightarrow \tau_3$ is schedulable

• Exercise: Calculate R_4

	e_i	p_i
τ_1	1	3
τ_2	1	4
τ_3	2	6
τ_4	1	20

Earliest Deadline First(EDF)

- At arrival times of tasks sort the ready queue according their deadlines; earliest deadline first
- Online dynamic scheduling algorithm
- EDF is optimal

Earliest Deadline First(EDF)

- Task Model
 - Independent tasks
 - Preemptive
 - Tasks are specified as follows:

$$\tau_i(e_i, d_i)$$

EDF; Schedulability Test

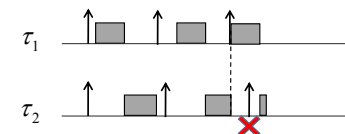
- Assuming n is the number of tasks of a periodic task set, the task set is schedulable if and only if:

$$U \leq 1 \quad \text{where } U = \sum_{i=1}^n \frac{e_i}{p_i}$$

- The condition is sufficient and necessary

Jitter

- In practice it can happen that a task does not arrives precisely at it beginning of its period



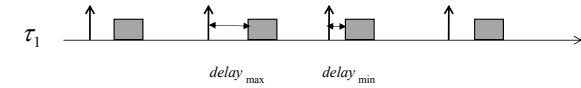
Jitter

- Let
 - $delay_{\max}$ = maximum delay of task τ_i from its period start
 - $delay_{\min}$ = minimum delay of task τ_i from its period start

$$jitter_i = delay_{\max} - delay_{\min}$$

Jitter

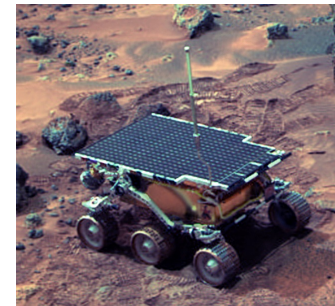
- Example



Lecture 11

Resource Access Protocols

- Mars Pathfinder



Mars Pathfinder

- An American spacecraft landed on Mars 1997. Its mission was to send meteorological data from Mars
- Its computing system was running on VxWorks
- A few days after it started its mission the system was experiencing resets causing the meteorological data being lost
- What was the problem?

Mars Pathfinder

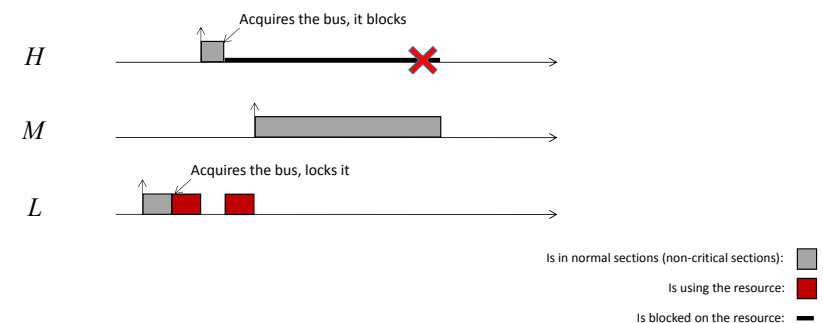
- The problem of Mars Pathfinder
 - There was a communication bus on the system used to transfer data among different components on Mars Pathfinder which could be used by one task at a time (mutual exclusive resource)
 - 3 tasks with different priorities were using the bus which was protected by a Mutex:
 - A **bus management** task with **high priority**
 - A not frequent but long running **communication task** with **medium priority**
 - A not frequent task for sending meteorological data with **low priority** which would acquire the mutex when using the bus

Mars Pathfinder

- The problem of Mars Pathfinder (Cont'd)
 - Most of the time the combination of the 3 tasks was working fine
 - However, in very infrequent times while the low priority task (meteorological data task) was using the bus (locking the mutex), the higher priority task (bus management task) would arrive and ask for the mutex and thus would block and wait. In this situation the medium priority task (communication task) would arrive and preempt lower priority task. The medium priority task would run for a long time and keeping lower priority task from releasing the bus. Thus the higher priority task would be blocked too long.
 - After sometime a watchdog timer would notice that the higher priority task has been running for too long time and would conclude that some thing had gone drastically wrong and would reset the system.
 - This problem is known as **Priority Inversion**

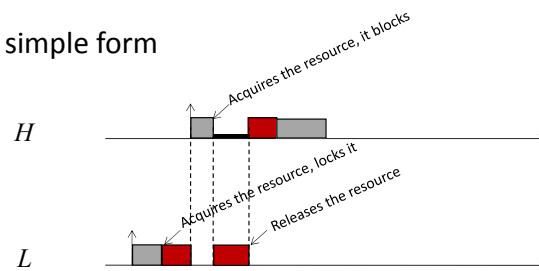
Mars Pathfinder

- The problem of Mars Pathfinder (Cont'd)



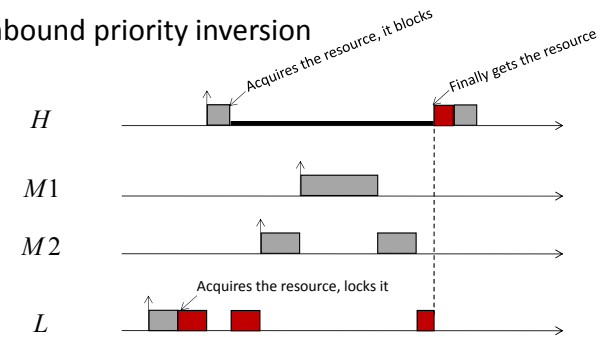
Priority Inversion

- The simple form



Priority Inversion

- Unbound priority inversion



Resource Access Protocols

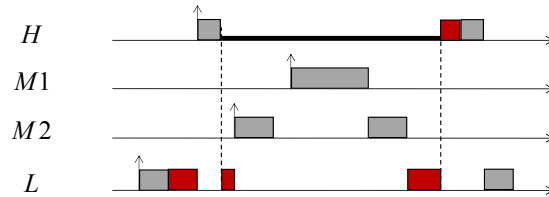
- How to overcome unbound priority inversion?
- Solution: Put some rules when tasks lock/unlock semaphores or mutexes
- Resource Sharing Protocols are used to apply the rules

Non-Preemptive Protocol (NPP)

- Protocol Rules
 1. Whenever a task locks a semaphore/mutex it runs non-preemptive, i.e., by boosting its priority to the highest priority
 2. Whenever a task releases the semaphore/mutex it becomes preemptive again, i.e., by returning to its original priority

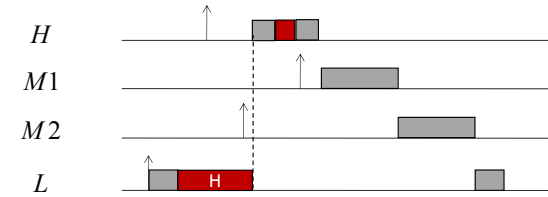
Non-Preemptive Protocol (NPP)

- Example
- **Without** the protocol:



Non-Preemptive Protocol (NPP)

- Example
- **With** the protocol:



Non-Preemptive Protocol (NPP)

- + Simple
- + Deadlock free
- + A task can be blocked by at most one lower priority task
- - Non preemption blocks all tasks including those that do not use any resource

Priority Inheritance Protocol (PIP)

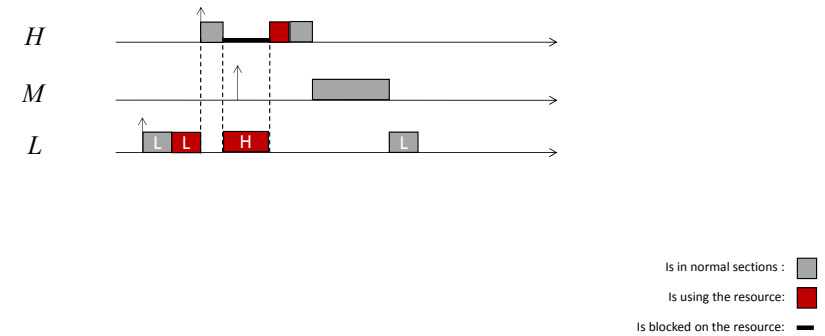
- Protocol Rules
 1. A task, τ_l , locks a semaphore S
 2. While τ_l is using the resource another task, τ_h , with higher priority arrives preempts τ_l , and at some time instant acquires semaphore S too. It will block since S is locked
 3. τ_l **inherits** the priority of τ_h , and runs with the priority of τ_h . I.e., the priority of lower priority task is boosted to the priority of the higher priority task
 4. When τ_l releases semaphore S, it will get back its original priority
 5. PIP is transitive meaning if τ_l blocks τ_m and τ_m blocks τ_h then τ_l inherits the priority of τ_h

Priority Inheritance Protocol (PIP)

- To sum up: Any time a lower priority task blocks a higher priority task it inherits the priority of the higher priority task

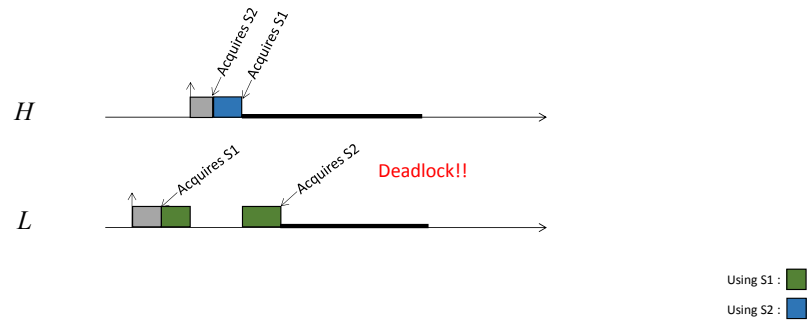
Priority Inheritance Protocol (PIP)

- Example



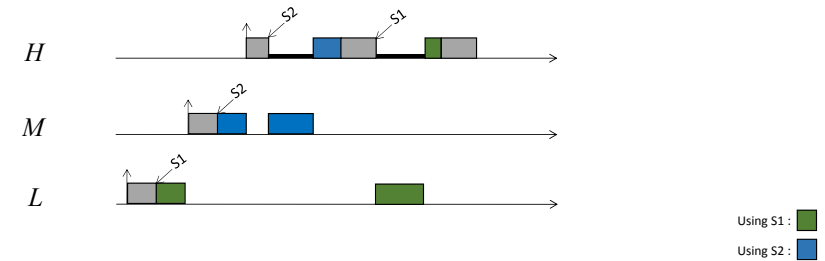
Priority Inheritance Protocol (PIP); Problems

- Deadlock



Priority Inheritance Protocol (PIP); Problems

- Multiple blockings



Priority Inheritance Protocol (PIP)

- + Has bounded priority inversion
- + No need to know resource usage of tasks in advance
- + Relatively good performance
- - Deadlock possible
- - Multiple blockings (Chained blocking)

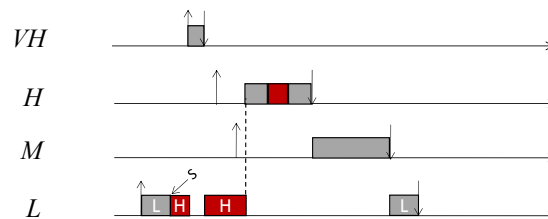
Highest Locker Priority Protocol (HLP)

- Also known as Immediate Priority Ceiling Protocol (IPC)
- Protocol Rules
 1. A ceiling is assigned to each resource (semaphore) which is the highest priority of all tasks that may use it

$$ceil(R) = \max \{ \rho_i \mid \tau_i \text{ uses } R \}$$
 2. Whenever a task starts using the resource (locks the semaphore) its priority is immediately boosted to the ceiling of the resource

Highest Locker Priority Protocol (HLP)

- Example: Tasks with priority H and L use S: $ceil(S) = H$



Highest Locker Priority Protocol (HLP)

- Maximum Blocking time of a task τ_i denoted by B_i
- τ_i can be blocked by at most one lower priority task that has priority ceiling higher or equal to τ_i
 - B_i equals to the duration of largest critical section belonging to any lower priority task that has priority ceiling higher than or equal to τ_i
 - Let $CS_j(R)$ denote the duration of longest critical section of task τ_j in which it uses resource R. Let ρ_i denote the priority of task τ_i

$$B_i = \max \{ CS_j(R) \mid \rho_j < \rho_i \text{ and } \rho_i \leq ceil(R) \}$$

Highest Locker Priority Protocol (HLP)

- Maximum Response Time

$$R_i = \underbrace{B_i}_{\text{priority inversion}} + e_i + \sum_{\tau_j \in H_i} \left\lceil \frac{R_i}{p_j} \right\rceil e_j$$

Highest Locker Priority Protocol (HLP)

- + Has bounded priority inversion
- + No Deadlock
- + No Multiple blockings; a task is blocked at most once
- + Relatively good performance

PIP and HLP in POSIX

```
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t  
    *restrict attr, int *restrict protocol);
```

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,  
    int protocol);
```

- protocol:
 - PTHREAD_PRIO_NONE
 - PTHREAD_PRIO_INHERIT: PIP
 - PTHREAD_PRIO_PROTECT: HLP

Priority Ceiling Protocol (PCP)

- Protocol Rules
 1. A ceiling is assigned to each resource (semaphore) which is the highest priority of all tasks that may use it
$$ceil(R) = \max \{ \rho_i \mid \tau_i \text{ uses } R \}$$
 2. During run-time the System Ceiling is the highest ceiling among all resources that are currently locked
$$sysceil = \max \{ ceil(R) \mid R \text{ is locked} \}$$
 3. Whenever a task τ_i acquires a resource it can lock the resource only if its priority is strictly higher than the system ceiling; $\rho_i > sysceil$
 4. If $\rho_i \leq sysceil$ then τ_i is said to be blocked by task τ_j that has locked the resource with ceiling equal to $sysceil$. In this case τ_j inherits the priority of τ_i

Priority Ceiling Protocol (PCP)

- + Has bounded priority inversion
- + No Deadlock
- + No Multiple blockings; a task is blocked at most once
- + Better response times for higher priority tasks
- - Complex implementation

Lecture 12

278

Petri Net

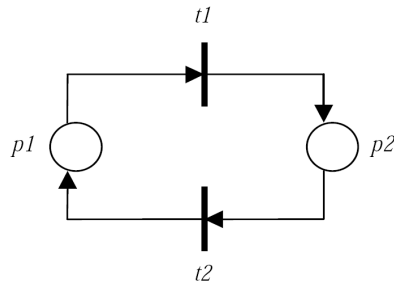
- A graphical and formal (mathematical) method for modeling and analyzing systems. They are specially useful for modeling systems that contain concurrent and asynchronous processing
- Model and analysis a system in design phase
- It's graphical, thus a system can be visualized
- It's mathematical, thus analysis and simulation can be done using tools
- Describes different states of a system and the conditions and events that transit the system from a state to another one

Places, Transitions, Arcs

- A Petri net contains arcs and two types of nodes; places and transitions and
- **Place**: Shown by a circle
 - Represents a condition, e.g., a resource is available, some data is available, a signal is arrived, a buffer is empty/full, etc.
- **Transition**: Shown by a solid bar or a rectangle
 - Represents an event, task, computation, processing, etc.
- **Arc**: Shown by a directed arc
 - Connects a place to a transition or a transition to a place. It does NOT connect two places or two transitions!

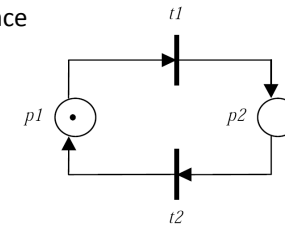


Example

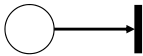
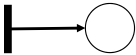


Token

- **Token**: Shown by a solid dot: •
 - To describe the behavior of a Petri net. Represents the fulfillment of a condition, e.g., a resource is available, data is ready, a signal is available, etc.
 - A place can contain any number of tokens. If the number of tokens in one place is too high a number is written in the place showing the number of tokens in that place



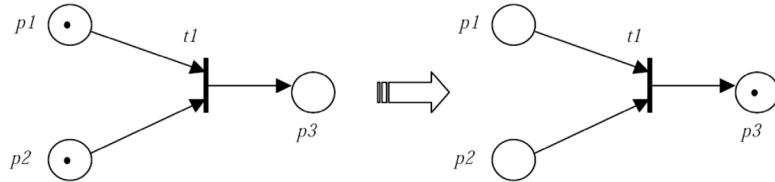
Input-Places, Output-Places, Generator, Stop

- A place that is connected by an arc **to** a transition is **input-place** for the transition
 
- A place that is connected by an arc **from** a transition is **output-place** for the transition
 
- A transition can have multiple input-places and/or output-places
- A transition without any input-place is called **Generator (Source)**, and a transition without any output-place is called **Stop (Sink)**

Enabled Transition, Firing a Transition

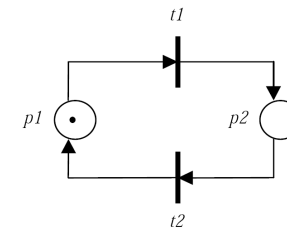
- A transition is **enabled** if all its input-places contain token
- **Firing** a transition: if a transition is enabled it can be fired
- When a transition is fired the tokens are removed from all of its input-places and tokens are added to all its output-places
 - The number of tokens taken from input-places might be different from the number of tokens added to output-places

Firing Example



Firing Sequence

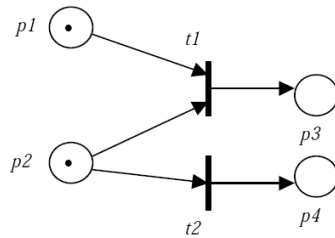
- **Firing Sequence:** A sequence of firing transitions



- Example: (t1, t2, t1, t2)

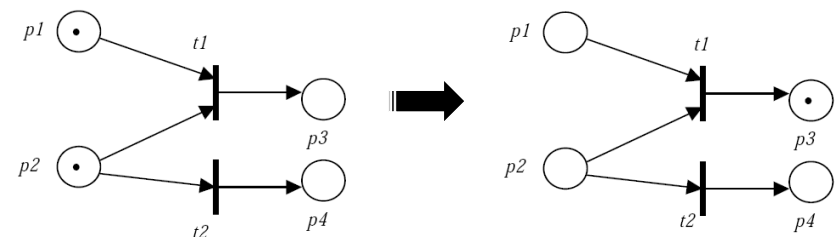
Firing Sequence

- If multiple transitions are enabled at the same time, firing sequence is not deterministic



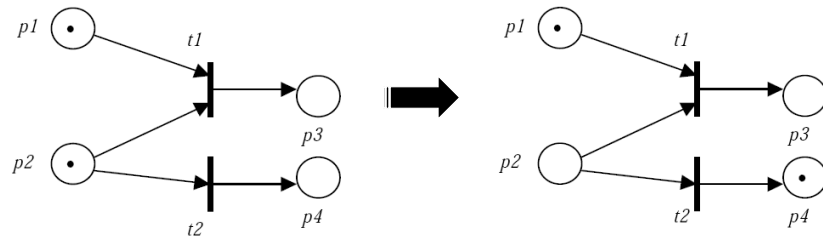
Firing Sequence

- Firing sequence is not deterministic
- 1)



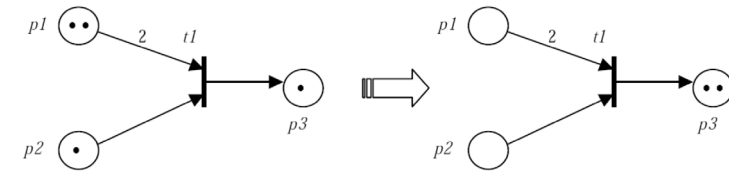
Firing Sequence

- Firing sequence is not deterministic
- 2)



Weighted Arcs

- **Weighted Arcs:** An arc can be weighted, i.e., by a number is written next to it. The number shows how many tokens it will take from input-place or will add to a output-place:

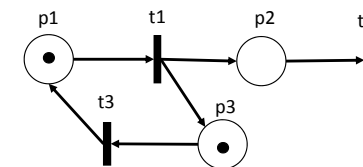


Marking

- **Marking:** A marking, M , of a Petri net is the distribution of tokens over the places, i.e., how many tokens are in each place. It shows the current state of the system
- A marking M can be shown by a tuple, $(M(p1), M(p2), \dots)$ where $M(pi)$ is the number of tokens in place pi : $(0,1,1,0)$

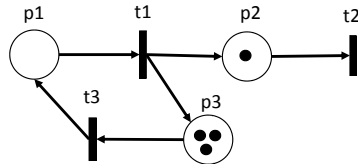
Marking Example

- Example1: $(1,0,1)$



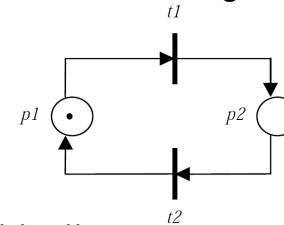
Marking Example

- Example2: (0,1,3)



Firing Sequence with Markings

- A firing sequence of the following Petri net:



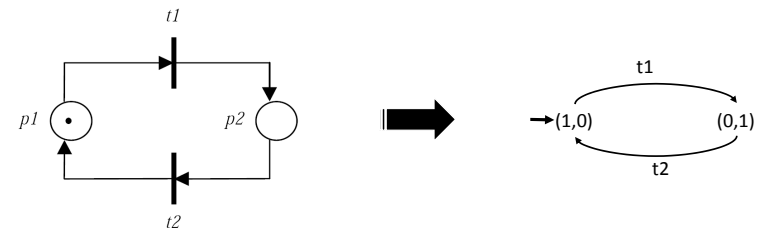
- ((1,0) (0,1) (1,0) (1,0))
- The marking at initial state of a Petri net is called Initial Marking, e.g., (1,0) of the example above.

Reachability Graph

- To be able to analyze a Petri net all possible markings have to be extracted
- Different possible markings of a Petri net drawn from an initial marking is shown by a **Reachability Graph**.
 - In a reachability graph nodes represent markings and the arrows connecting them represent transitions

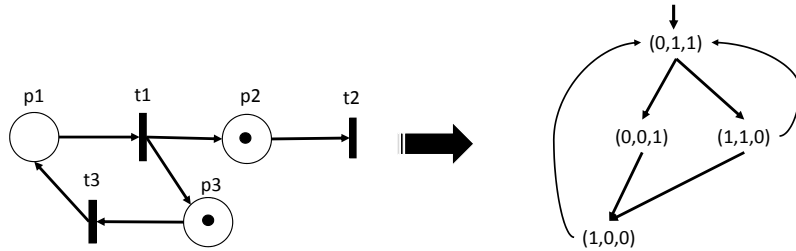
Reachability Graph Example

- Example1



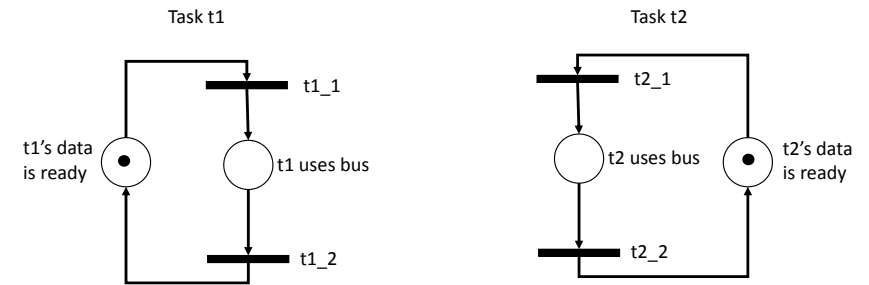
Reachability Graph Example

- Example2

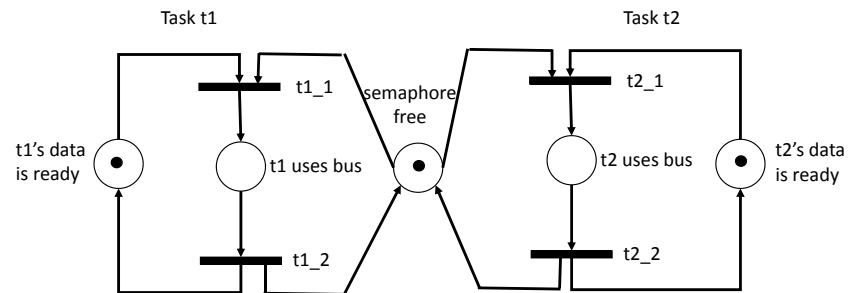


Mutual Exclusion

- Assume two tasks that use a bus to transfer some data. Only one task at a time is allowed to use the bus

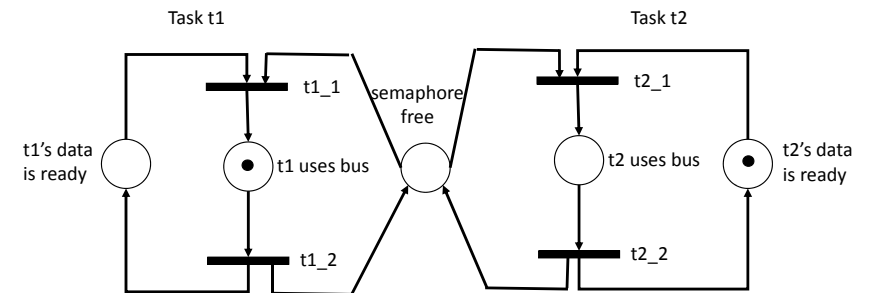


Mutual Exclusion



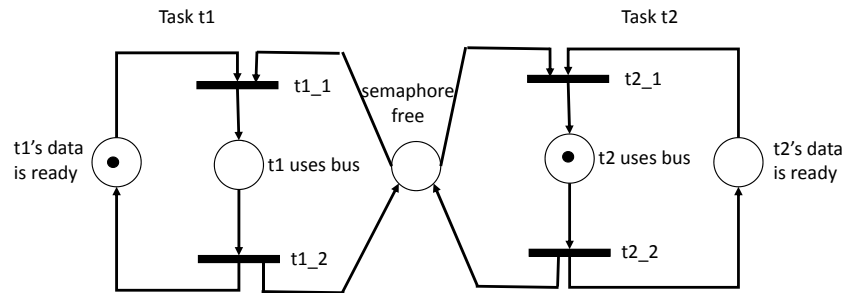
Mutual Exclusion

- A possible marking



Mutual Exclusion

- Another possible marking



Properties of Petri Net

The benefit of modeling a system using a formal modeling method like Petri net is to be able to analyze some properties of the system. The following properties of a system can be analyzed using a Petri net model:

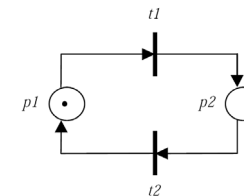
- Reachability
- Liveness
- Boundedness
- Fairness

Reachability

- Given an initial Marking M_0 , a marking M is said to be reachable if there exists a firing sequence that transforms M_0 to M
- To check reachability of marking from an initial marking the reachability graph has to be drawn from the initial marking
- This property is useful to check
 - If the system can arrive in a forbidden (erroneous) state, e.g., deadlock
 - If the system can arrive in a desired state

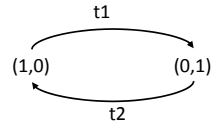
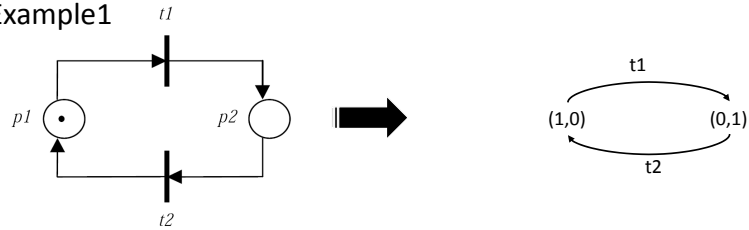
Reachability

- Example1: is (0,1) reachable from the following Petri net? How about (1,1)?



Reachability Example

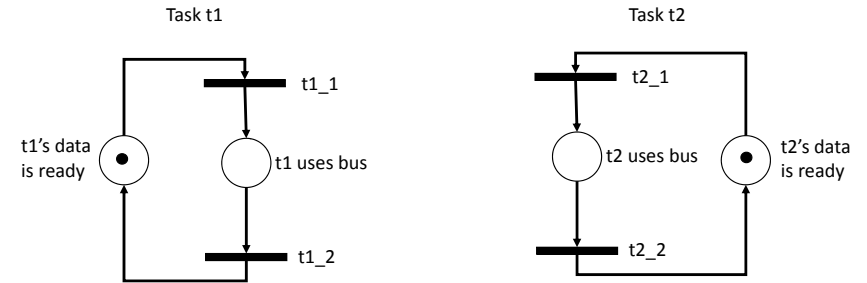
- Example1



- $(0,1)$ is reachable but $(1,1)$ is not reachable

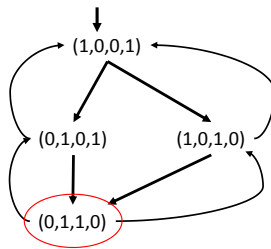
Reachability Example

- Example2: is $(0,1,1,0)$ which is a forbidden state reachable?



Reachability Example

- Example2: Let extract the reachability graph

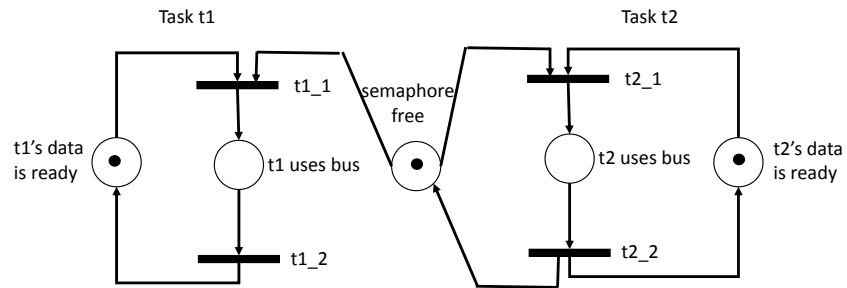


Liveness

- A Petri net is live if it can never stuck in a deadlock state, i.e., there is no marking where no transition can be fired
 - There is no marking in which a transition is disabled permanently, i.e., a transition can be fired infinite times
- This property is used to check whether the system will arrive in a deadlock state

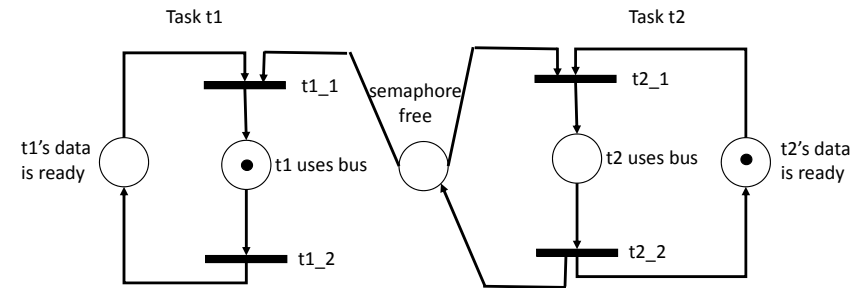
Liveness Example

- Example (t1 does not release the semaphore). Is the following Petri net live?



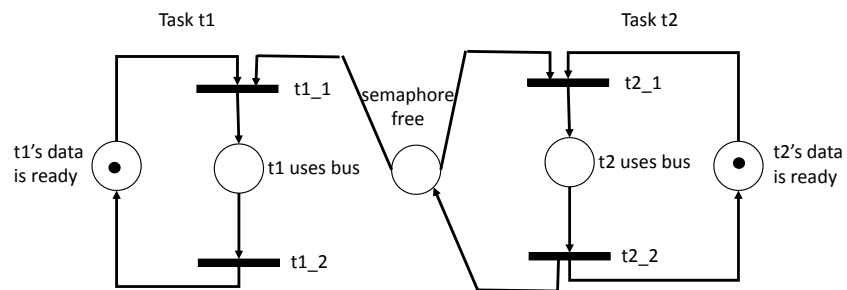
Liveness Example

- A possible marking sequence (t1_1, t1_2):

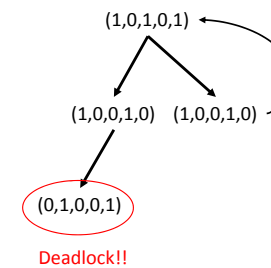


Liveness Example

- It can not proceed anymore, i.e., Deadlock!



Liveness Example

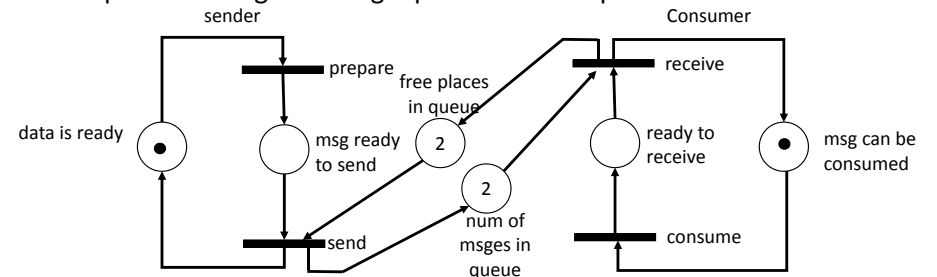


Boundedness

- A Petri net is bounded if it has no marking where the number of tokens in any place is more than k , where k is a positive integer
- If $k=1$ the Petri net is said to be **safe**
- This property is used to check if the maximum limits of resources are exceeded

Boundedness Example

- Example. Modeling a message queue used in a producer-consumer

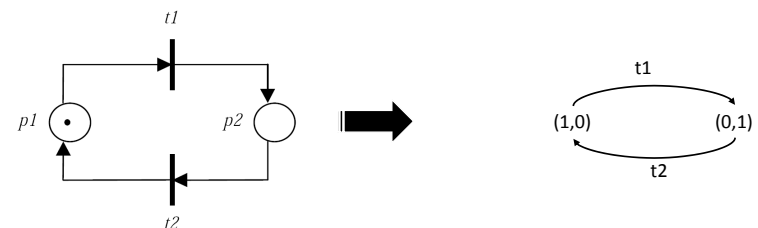


- What is the Bound (limit) of the model? Can at any time the queue have more than 4 free spaces or contain more than 4 messages?

Fairness

- Two transitions are said to be **mutually limited fair** if there is a limited number of firings for each one before the other one is fired, i.e., one of them can be fired up to a limit until the other one is fired
- **Global Fairness:**
 - An infinite firing sequence is said to be globally fair if every transition is fired infinite times
 - A Petri net is globally fair if any firing sequence from any initial marking is globally fair

Fairness Example



- $(t1, t2, t1, t2, t1, \dots)$: every transition is fired infinite times, thus the Petri net is globally fair
- $t1$ and $t2$ are mutually limited fair because none of them can be fired more than once unless the other one is fired

Coverable Marking

- A marking M is said to be **coverable** by a marking M_i if from an initial marking M_0 marking M_i is reachable and for every place in M_i the number of tokens are equal or greater than the same place in M , i.e. for every place p : $M[p] \leq M_i[p]$
- Example:

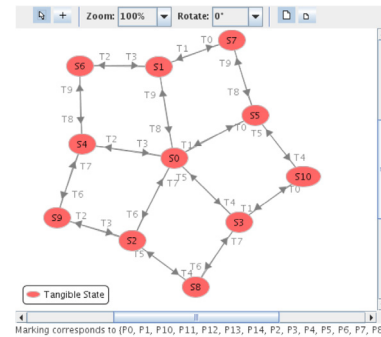
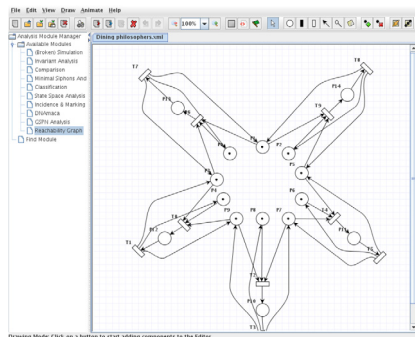
$\rightarrow (1,0,3,0) \longrightarrow (0,1,1,3) \longrightarrow (1,0,1,2) \longrightarrow (1,1,2,4)$

$(0,1,1,3)$ is coverable by $(1,1,2,4)$

Tools for Modeling Petri Net

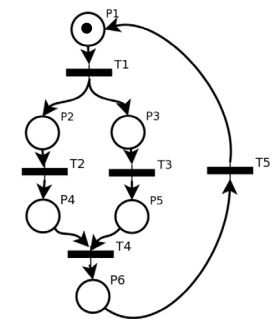
- There are many programs available for modeling, analyzing and simulating a system using Petri net models.
- PIPE 2: <http://pipe2.sourceforge.net/>
 - A free editor program for Petri net
 - Platform independent (written in Java)

PIPE 2



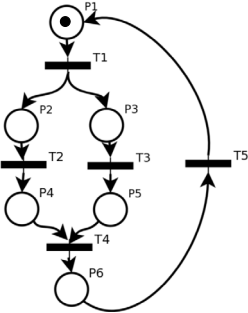
Petri Net Properties Example

- Modeling of a program where two tasks are repeatedly created, run and joined.
- Is the Petri net:
 - Bounded?
 - Live?
 - Fair?



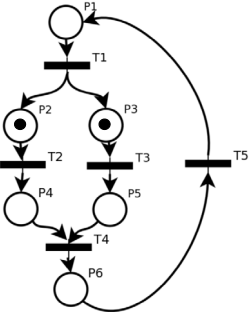
Petri Net Properties Example

\downarrow
(1,0,0,0,0,0)



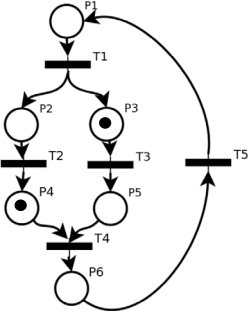
Petri Net Properties Example

\downarrow
(1,0,0,0,0,0)
 \downarrow_{T1}
(0,1,1,0,0,0)



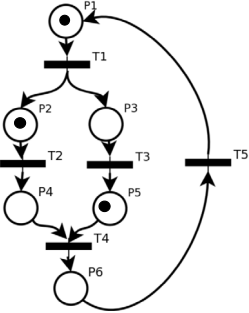
Petri Net Properties Example

\downarrow
(1,0,0,0,0,0)
 \downarrow_{T1}
(0,1,1,0,0,0)
 \swarrow_{T2}
(0,0,1,1,0,0)

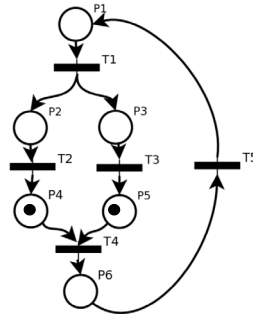
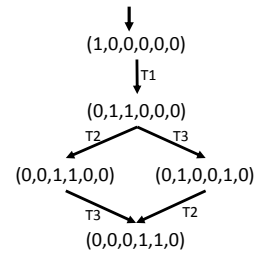


Petri Net Properties Example

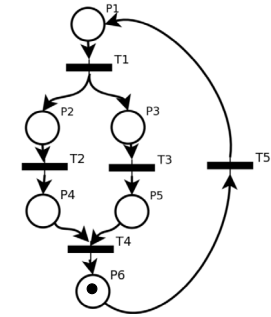
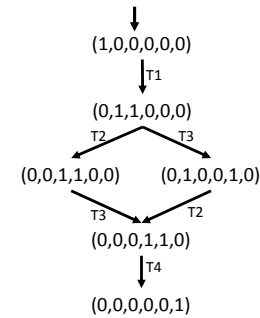
\downarrow
(1,0,0,0,0,0)
 \downarrow_{T1}
(0,1,1,0,0,0)
 \searrow_{T3}
(0,1,0,0,1,0)



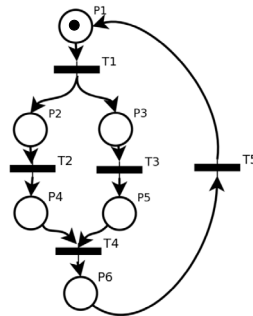
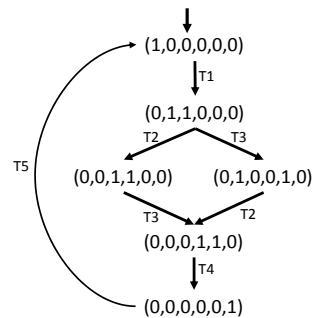
Petri Net Properties Example



Petri Net Properties Example



Petri Net Properties Example



Lecture 13

Timed Petri Nets

- Extend the Petri net model with time to model timing attributes of a system. Either of the followings:
 - P-Timed**: Associate a constant number with every place. The number indicates the time interval that the place takes, i.e., the time a token has to stay in the place before it can be available
 - T-Timed**: Associate a constant number with every transition. The number indicates time interval that the transition takes
- Any of P-Timed or T-Timed Petri net models can be transformed to the other one. Thus we only consider one of them (P-Timed)

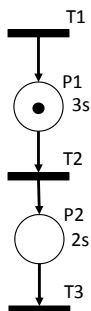
P-Timed Petri Nets

- Assume a place is associated with time interval d . When a token is added to the place it has to stay for d time units in the place until it can be available

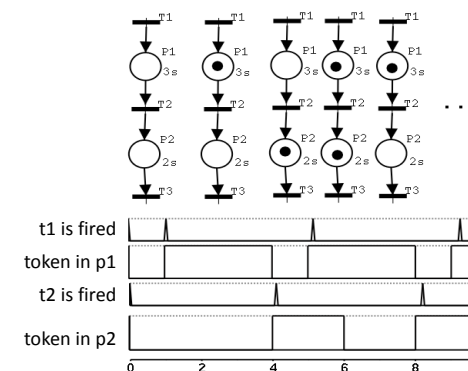


P-Timed Petri Nets

- Example



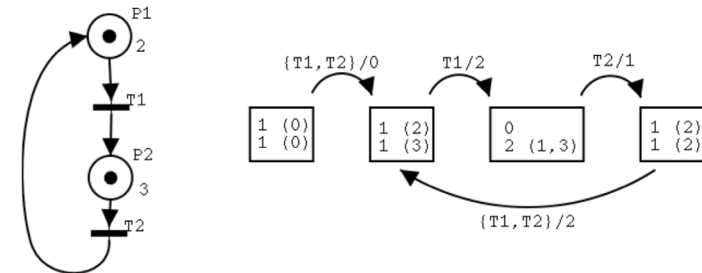
P-Timed Petri Nets Example



Reachability Graph for P-Timed Petri Nets

- For each individual token in a place the time it needs to spent in the place has to be recorded
- On each arrow we write the followings:
 - The transitions that are fired
 - The minimum time that takes to transfer from one marking to another one
- A marking is shown by a rectangle in which the followings are written:
 - The number of tokens in each place
 - The remaining time for each token to be ready

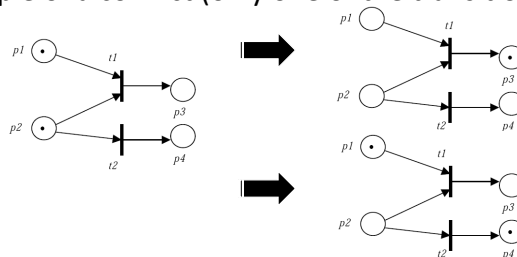
Reachability Graph for P-Timed Petri Nets; Example



- $\{T1, T2\}/0$ means that both T1 and T2 are fired at the beginning

P-Timed Petri Nets

- Every limited P-Timed Petri net without effective conflicts has a periodic behavior meaning that after a constant period it is repeated. Such a Petri net is said to have a **stationary behavior**
- Example of a conflict (only one of the transitions can be fired):

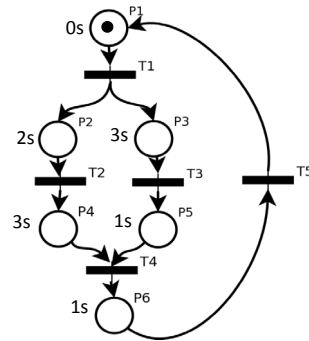


P-Timed Petri Nets

- **Maximum Speed:** If any transition is fired as soon as tokens in its input-places are available, the Petri net is executed in maximum speed
- **Firing Frequency:** The firing frequency of a transition is the number firings it performs in one time unit

P-Timed Petri Nets; Exercise

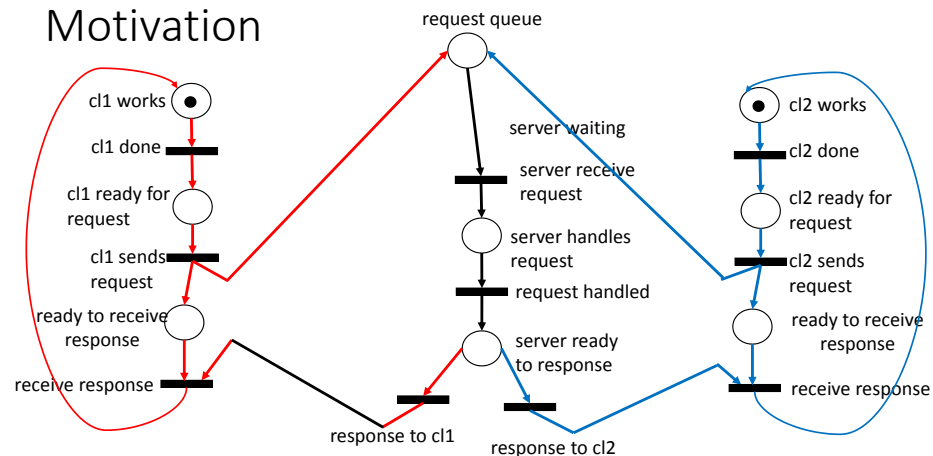
- Assuming the following Petri net executes with maximum speed, draw its reachability graph
- What is the period of the net?
- What is the firing frequency of T1?



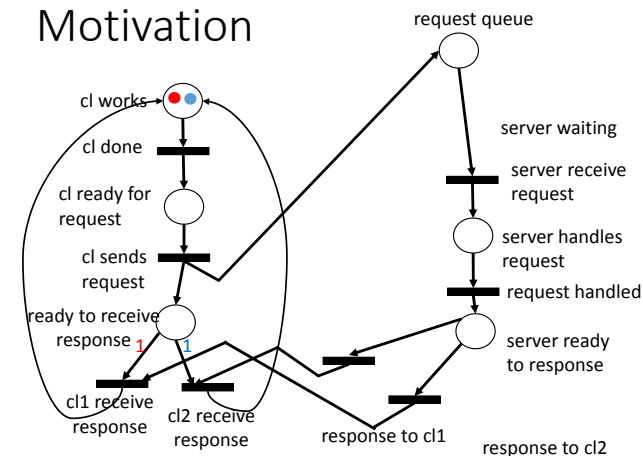
Colored Petri Nets

- So far all tokens have been identical
- Let look at an example

Colored Petri Nets; Motivation



Colored Petri Nets; Motivation

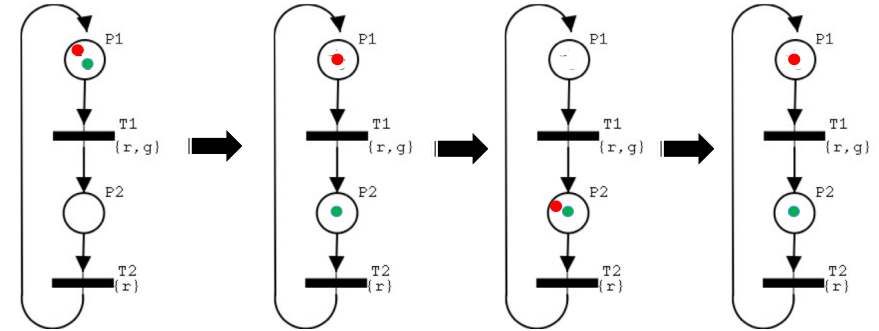


Colored Petri Nets

- In a colored Petri net:
 - Each token may have a color.
 - The color is a metaphor to differentiate values of tokens, i.e., a color represents the value of a token
 - Transitions are sensitive to colors, i.e., they distinguish between different colors. A transition might be enabled by **some** of the colored tokens
 - An arc is associated with a function with colored tokens as its parameters, e.g., $f(\text{red}) = \text{green}$ or $f(\text{green}) = 2\text{reads}$. If no function is specified, each token remains at it is, e.g., $f(\text{red}) = \text{red}$, $f(\text{green}) = \text{green}$, etc.

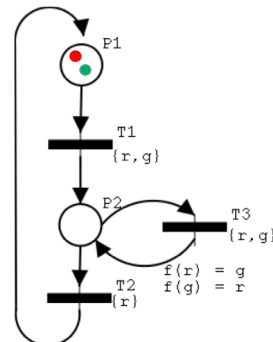
Colored Petri Nets

- Example1, the arcs are not specified with function



Colored Petri Nets

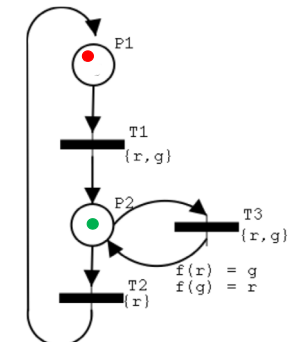
- Example2, an arc is specified with a function that transforms the colors



Colored Petri Nets

- Example2, an arc is specified with a function that transforms the colors

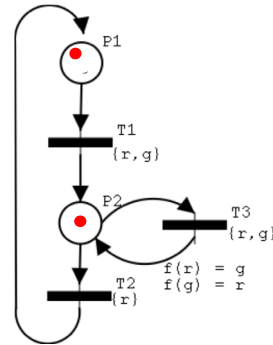
- $T1(g)$ is fired
 - Remove $f(g)=g$ from P1
 - Add $f(g)=g$ to P2



Colored Petri Nets

- Example2, an arc is specified with a function that transforms the colors

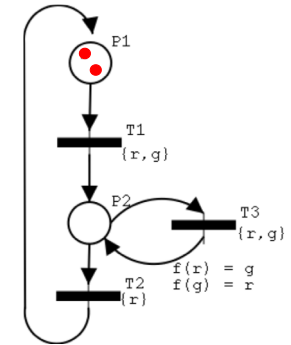
- T1(g) is fired
 - Remove $f(g)=g$ from P1
 - Add $f(g)=g$ to P2
- T3(g) is fired
 - Remove $f(g)=g$ from P2
 - Add $f(g)=r$ to P2



Colored Petri Nets

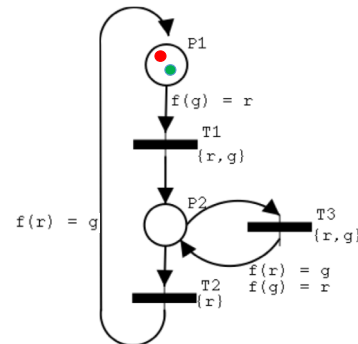
- Example2, an arc is specified with a function that transforms the colors

- T1(g) is fired
 - Remove $f(g)=g$ from P1
 - Add $f(g)=g$ to P2
- T3(g) is fired
 - Remove $f(g)=g$ from P2
 - Add $f(g)=r$ to P2
- T2(r) is fired
 - Remove $f(r)=r$ from P2
 - Add $f(r)=r$ to P1



Colored Petri Nets

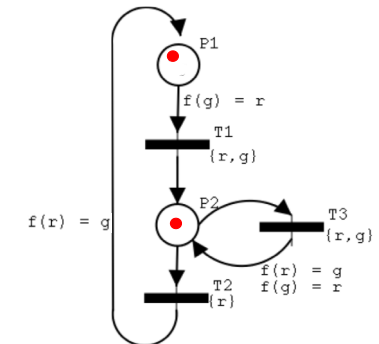
- Example3, three arcs are specified with functions that transform the colors



Coloured Petri Nets

- Example3, two arcs are specified with functions that transforms the colours

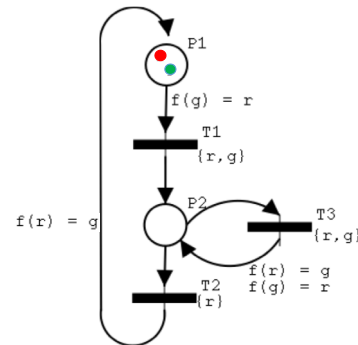
- T1(g) is fired
 - Remove $f(g)=r$ from P1
 - Add $f(r)=r$ to P2



Coloured Petri Nets

- Example3, two arcs are specified with functions that transforms the colours

- T1(g) is fired
 - Remove $f(g)=r$ from P1
 - Add $f(r)=r$ to P2
- T2(r) is fired
 - Remove $f(r)=r$ from P2
 - Add $f(r)=g$ to P1



350

Embedded System, Real-Time Systems

- Embedded Systems
- Real-Time Systems
 - Hard Real-Time System
 - Soft Real-Time Systems
- Real-Time Task
 - Hard Task
 - Soft Task
 - Firm Task

351

Lecture 14

Properties of Real-Time Systems

- Complexity
- Reliability
- Concurrency
- Interactive with physical world
- Schedulability
- Fault Tolerant
- Predictability

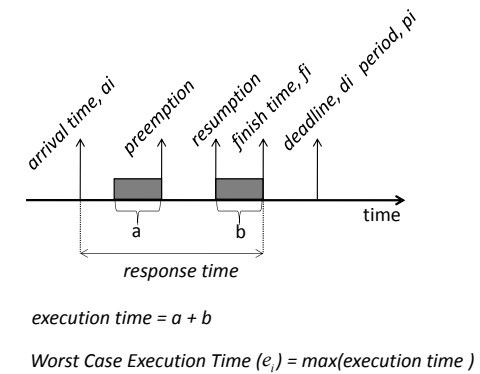
352

Real-Time Systems

- Timing facilities of a RTOS
 - Clock
 - Delaying the execution of a task
- Task Communication
 - External communication
 - Synchronization
 - Data communication
- Extract Timing Requirements
 - Application requirements
 - Physical rules

353

Task parameters



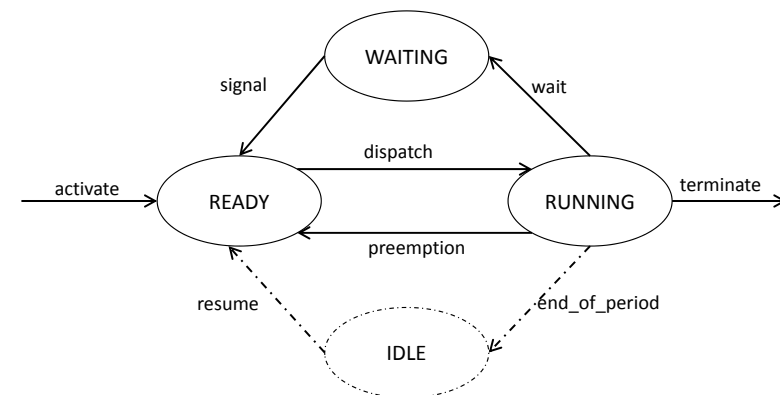
354

Real-Time Tasks

- Preemptive, non-preemptive
- Periodic
- Aperiodic
- Sporadic

355

Task States in RTOS



356

Task Synchronization and Communication

- Shared Variable Based
- Message Passing Based

357

Shared Variable Based Synchronization and Communication

- Atomic (Indivisible) Operation
- Critical Sections
- Mutual Exclusion
- Busy Waiting
- Semaphores
- Mutexes
- Condition Variables
- Monitors
- Barriers

358

Message Passing Based Synchronization and Communication

- Synchronization Model
 - Asynchronous
 - Synchronous
 - Remote Procedure Call
- Naming of Source/Destination
 - Direct Naming
 - Indirect Naming
 - Symmetry
- The Structure of Message

359

Task Synchronization and Communication

- Message Queues
- Client-Server Communication
- Pipes
- Sockets
- Events
- Signals

360

POSIX Standard

- POSIX = Portable Operating System Interface (for Unix)s
- A family of related standards
 - Provides standard Application Programming Interfaces (APIs) and command line shells and utilities for an operating system
 - Specified by IEEE
 - Facilitates developing applications portable to any operating system compatible with POSIX

361

Basic Facilities Provided by a RTOS

- Timing Facilities
- Task Management
- Memory Management
- Error Handling
- I/O Services
- Interrupt Handling
- Task Synchronization and Communication
- Scheduling

362

Real-Time Scheduling

- How the tasks in the ready queue are ordered
- The Scheduler provided by RTOS sorts the queue according to a scheduling algorithm
- A scheduling algorithm sorts tasks; decides which task should run next

363

Real-Time Scheduling

- Task Model
 - Task parameters
 - Resource requirement constraints
 - Precedence constraints
- Schedulability Analysis
 - Schedulability
 - Feasible Schedule
 - Schedulability test
- Scheduling Algorithms

364

Categories of Scheduling Algorithms

- Preemptive vs. Non-preemptive
- Static vs. Dynamic
- Offline vs. Online
- Optimal vs. Heuristic
- Time driven vs. Event driven

365

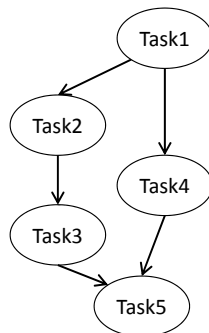
Aperiodic Scheduling Algorithms

- Same arrival times
 - Earliest Due Date (EDD) – Jackson's Algorithm
- Arbitrary arrival times
 - Earliest Deadline First (EDF) - Horn's algorithm
 - Least Slack Time First (LST)
 - Non-Preemptive:
 - Bratley's Algorithm
 - The Spring Algorithm (A heuristic algorithm)

366

Scheduling with Precedence Constraints

- Directed Acyclic Graph (DAG)



367

Scheduling Algorithms with Precedence Constraints

- Same arrival times
 - Latest Deadline First (LDF)
- Arbitrary Arrival times
 - EDF (not optimal)
 - Bratley's Algorithm
 - Spring Algorithm
 - EDF* (Transform arrival times and deadlines)

368

Periodic Scheduling Algorithms

- Timeline Algorithm (Cyclic Executive)
- EDF
- Rate Monotonic Scheduling Algorithm (RMS)
 - Critical instant of a task
 - Critical instant of a task set
 - Schedulability analysis
 - Sufficient but not necessary: $U_{\text{lub}} = n \times (2^{1/n} - 1)$
 - Response time analysis:

$$R^{(k+1)}_i = e_i + \sum_{\tau_j \in H_i} \left\lceil \frac{R^k_i}{p_j} \right\rceil e_j \quad R^{(0)}_i = e_i + \sum_{\tau_j \in H_i} e_j$$

369

Jitter

$$jitter_i = delay_{\max} - delay_{\min}$$

where

$delay_{\max}$ = maximum delay of task τ_i from its period start

$delay_{\min}$ = minimum delay of task τ_i from its period start

370

Resource Sharing

- Priority Inversion
- Deadlock
- Chained blocking (Multiple blockings)
- Resource Access Protocols:
 - Non-Preemptive Protocol (NPP)
 - Priority Inheritance Protocol (PIP)
 - Highest Locker Priority Protocol (HLP). Also known as Immediate Priority Ceiling Protocol (IPC)
 - Priority Ceiling Protocol (PCP)

371

Petri Net

- A graphical and formal (mathematical) method for modeling and analyzing systems
- Modeling systems with concurrent and asynchronous processing
- Model and analysis in design phase

372

Petri Net

- Place
- Transition
- Arc
- Token
- Input-Places, Output-Places, Generator (Source), Stop (Sink)
- Enabled Transition, Firing a Transition
- Firing Sequence
- Weighted Arcs
- Marking
- Reachability Graph

373

Properties of Petri Nets

- Reachability
- Liveness
- Boundedness
- Fairness

374

Advanced Petri Nets

- Timed Petri nets
 - P-timed, T-timed
 - Stationary behavior
 - Maximum speed
 - Firing frequency
- Colored Petri Nets
 - Different values for tokens
 - Transitions sensitive to token colors (values)
 - Arcs associated with functions

375

Criteria to Pass the Course

- Approved written exam
 - 4-6 questions, 40 points
 - Required: 20 points for grade 3, 30 for grade 4, and 35 for grade 5
- Approved labs
 - Required: All 4 labs are handed in and demonstrated.

376

Example Exam

1. (10 points)

Briefly explain the following concepts:

- a) Task
- b) Semaphore
- c) Priority Inversion
- d) Message queue
- e) Periodic, Aperiodic, Sporadic task
- f) Deadlock
- g) Monitor
- h) Hard Real-Time Systems
- i) Critical section
- j) POSIX Standard

Example Exam

2. (12 points)

We want to implement a system which contains 3 periodic tasks; t1, t2, and t3. t1 and t2 run with period 10ms and t3 with period 8ms. Each task performs some work and send some data (an integer number) on a network bus. Sending on the bus take 1ms and only one task at a time can have access to the bus (mutually exclusive access).

- a) Design the system using a P-timed Petri net
 - b) Show how the system can be implemented in a safe way
- You can use the following function calls:

Example Exam

```
void work(int i); // performs the work for task i
void sendOnBus(int i); // sends data on the bus for task i
int nanosleep(const struct timespec* t1, struct timespec* t2); // the task for a given time in t1
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *),
void *arg); // creates a task
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
// initializes a mutex
int pthread_mutex_lock(pthread_mutex_t *mutex); // locks a mutex
int pthread_mutex_unlock(pthread_mutex_t *mutex); // unlocks a mutex
```

Example Exam

3. (6 points)

Show how the system in question 2 can be implemented

- a) Using a monitor
- b) Using an extra task, t4, that performs all the sending on the bus. I.e., the tasks t1, t2, and t3 put their data into a queue and t4 sends the data (an integer number) on the bus.

You may use the following function calls:

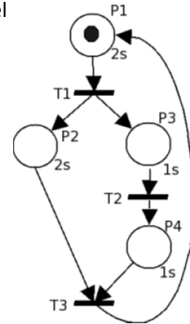
```
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
//creates/opens a message queue
//O_RDONLY open the queue to receive messages only, O_WRONLY open the queue to
//send messages only, and O_RDWR open the queue to both send and receive messages.
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);
//sends a message to the given queue
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
//reads a message from the given queue
```

Example Exam

4. (6 points)

Assume that a system is designed with the following Petri net model

- a) Draw the reachability graph of the Petri net
- b) Assuming the model executes with maximum speed what is the
 - 1) Period of the Petri net
 - 2) The firing frequency of the transitions



381

Example Exam

5. (6 points)

- a) Given a periodic preemptive task set, how the following scheduling algorithm would schedule the task set?
 - 1) Earliest Deadline First (EDF)
 - 2) Rate Monotonic Scheduling (RMS)
- b) What is a Resource Access Protocol? How the following resource access protocols work?
 - 1) Priority Inheritance Protocol (PIP)
 - 2) Highest Locker Priority Protocol (HLP)

382

The End!

- Labs
 - Away: May 8th – 17th, June 2 weeks
- The optional lab
- Please do the course evaluation!

383

Lecture 14 Optional

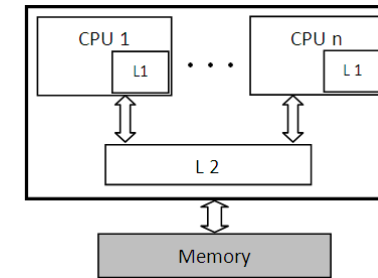
384

Introduction

• Multi-core Architectures

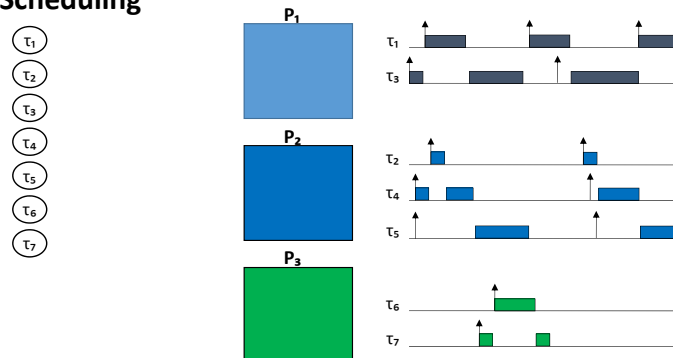
- Two or more independent processors (cores) on one chip (Single-chip multiprocessor).
- Processors may have independent (L1) and/or on-chip shared (L2) cache.
- Actual parallelism.
- Very fast inter-core communication.
- Shared Memory and BUS.
- Cores may have identical or different performance.
- Overcome thermal and power consumption problems.

Introduction



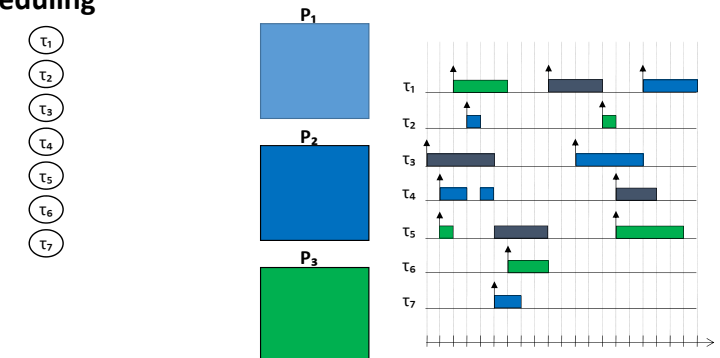
Scheduling on Multiprocessors

• Partitioned Scheduling



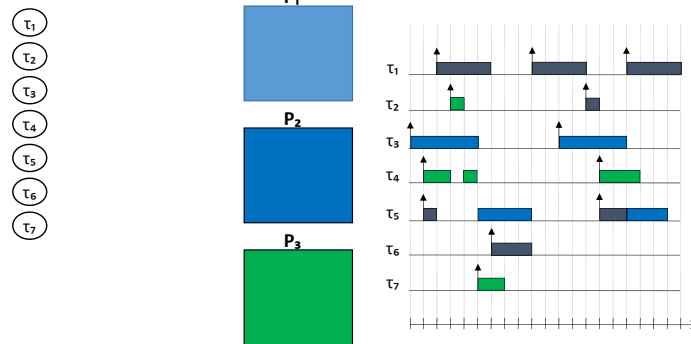
Scheduling on Multiprocessors

• Global Scheduling



Scheduling on Multiprocessors

• Hybrid Scheduling



Assign Tasks to Cores (Linux)

```
#include <pthread.h>
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);
int pthread_getaffinity_np(pthread_t thread, size_t cpusetsize, cpu_set_t *cpuset);
```

Some macros for working with **cpu_set_t**:

```
void CPU_ZERO(cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);
int CPU_ISSET(int cpu, cpu_set_t *set);
```

Assign Tasks to Cores (Linux)

Set the attribute object of thread to assign the thread to specific processors at the time the thread created creating:

```
int pthread_attr_setaffinity_np(pthread_attr_t *attr, size_t cpusetsize, const cpu_set_t *cpuset);
```

Assign Tasks to Cores; Example

- See the attached code (setAffinityExample.c)