1 Explain Concepts

1.1 Task

- Periodic: Task is executed in a repeated constant interval.
- Aperiodic: Task is executed at any time, such as an event.
- Sporadic: Task is executed at any time but has a minimum time interval when it cannot be executed again. Only after the minimum has passed, is it allowed to execute.

1.2 Semaphore

A variable or abstract data type which is used to controlling access, by multiple processes, to a common resource in a concurrent system.

1.3 Priority Inversion

When a task with lower priority preempties a task with a higher priority.

1.4 Message Queue

A queue that holds elements. It can be written to and read from and send messages with different priorities. Highest priority messages are read before the lower ones.

1.5 Deadlock

When task 1 waits for task 2 and task 2 waits for task 1.

1.6 Monitor

A module which encapuslates critical sections and operations needed for managing synchronization. Critical sections and variables to be accessed under mutual exclusion are hidden from the caller.

1.7 Hard Real-Time Systems

A real-time system which are not allowed to miss its deadline.

1.8 Critical section

A piece of code (sequence of statements) that no more than one task can execute at the same time.

1.9 Barrier

Syncronization point for a group of tasks. Every task stops at the assigned barrier until everyone has reached the barrier.

1.10 Jitter

In practice it can happen that a task does not arrive precisely at the beginning of its period. A jitter can solve this by introducing artifical delay from the task delays themselves.

delay(max) = **maximum** delay of a task from its period start
delay(min) = **minimum** delay of a task from its period start
jitter = **delay(max) - delay(min)**

1.11 Task States

Different states that a task can have are: RUNNING, READY, BLOCKED and SUSPENDED.

- **RUNNING**: When a task is executing it's in a RUNNING state. If the processor has a single core there can be only one task in this state.
- **READY**: A task that is able to execute (not in blocked or suspended state) but is also not currently executing because a different task is in the RUNNING state.
- **BLOCKED**: A task can be blocked from entering the RUNNING state by waiting for a temporal or external event. Such as a semaphore, queue etc. Usually has a timer which unblocks it after a certain amount of time if a specific event has not occured.
- **SUSPENDED**: Cannot enter RUNNING state until it is explicitly commanded through specific functions (vTaskSuspend(), xTaskResume()). This state does not have a timer.

1.12 POSIX Standard

(P)ortable (O)perating (S)ystem (I)nterface Uni(X)

1.13 Client-Server Communication

Server accepts requests of services from clients and provides it. Server creates a request queue where clients can send their requests. Each client has its own reply queue.

2 Implement a system

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

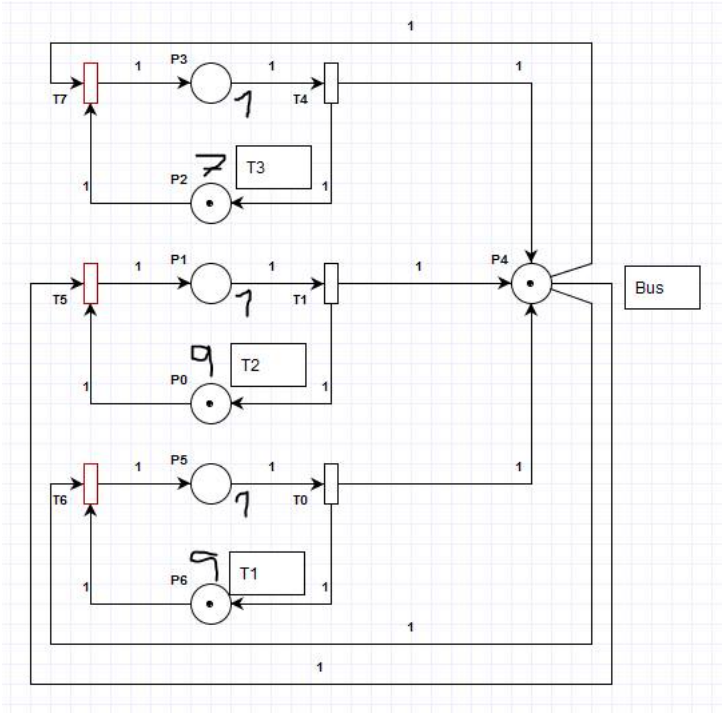

Figure 1: P-timed petri net

- b) Show how the system can be implemented in a safe way

You can use the following function calls:

```
void work(int i);ă// performs the work for task i
voidăsendOnBus(int i);ă// sends data on the bus for taski
intnanosleep(const struct timespec*ăt1,ăstruct timespec*ăt2); // the ↩
    task for a given time in t1
int pthread_create(pthread_t *thread,ăconst pthread_attr_t *attr,↩
    ăvoidă*(*start_routine)ă(voidă*),voidă*arg); // creates a task
int pthread_mutex_init(pthread_mutex_t *restrict mutex,ăconst ↩
    pthread_mutexattr_t *restrict attr); // initializes a mutex
intăpthread_mutex_lock(pthread_mutex_tă*mutex); // locks a mutex
intăpthread_mutex_unlock(pthread_mutex_tă*mutex); // unlocks a mutex
```

3 Implement a system cont.

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 chară*name,ăint oflag,ămode_t mode,ăăstruct ↩
    mq_attr *attr);
// creates and opens a queue
//O_RDONLYăopenătheăqueueătoăreceiveămessagesăonly,↩
    ăO_WRONLYăopenătheăqueueăto //sendămessagesăonly,↩
    ăandăO_RDWRăopenătheăqueueătoăbothăsendăandăreceiveămessages.
int mq_send(mqd_t mqdes,ăconst chară*msg_ptr,ăsize_t msg_len,ăunsigned↩
     int msg_prio);
//sends aămessage toătheăgivenăqueue
ssize_t mq_receive(mqd_t mqdes,ăchară*msg_ptr,ăsize_t msg_len,↩
    ăunsignedăint *msg_prio);
//readsăaămessageăfromătheăgivenăqueue
```

4 Petri net

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

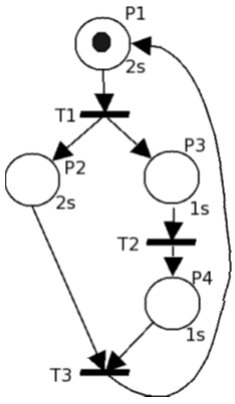

Figure 2: Petri model

- a) Draw the reachability graph of the Petri net:

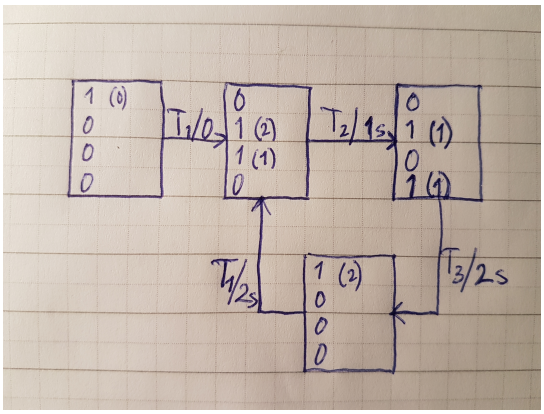

Figure 3: Reachability graph (T3 should be 1 sec, not 2 sec!)

-
- b) Assuming the model executes with maximum speed what is the:
  - Period of the Petri net?
    - $1s + 2s + 2s = 4seconds$
  - The firing frequency of the transitions?
    - $4firings = 1/4Hz = 0.25Hz$

5 Scheduling and Resource Access Protocols

- Given a periodic preemptive task set, how the following scheduling algorithm would schedule the task set?
  - Earliest Deadline First (EDF)
    - Consider 3 periodic processes scheduled on a preemptive uniprocessor. The execution times and periods are as shown in the following table:

Process Timing Data

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	4	10

Figure 4: 3 periodic processes scheduled on a preemptive processor

* In this example, the units of time may be considered to be schedulable time slices. The deadlines are that each periodic process must complete within its period:

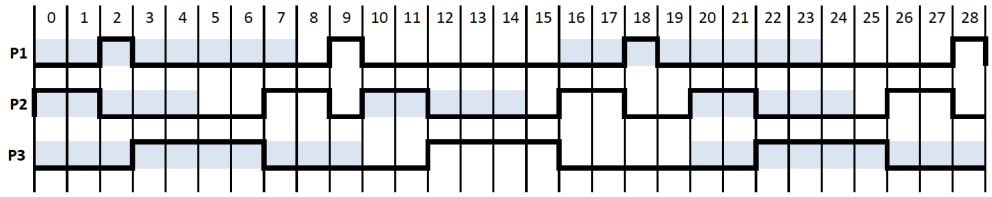

Figure 5: Timing Diagram showing part of one possible schedule for the example

* In the timing diagram, the columns represent time slices with time increasing to the right, and the processes all start their periods at time slice 0. The timing diagram's alternating blue and white shading indicates each process's periods, with deadlines at the color changes.
* The first process scheduled by EDF is P2, because its period is shortest, and therefore it has the earliest deadline. Likewise, when P2 completes, P1 is scheduled, followed by P3.
* At time slice 5, both P2 and P3 have the same deadline, needing to complete before time slice 10, so EDF may schedule either one.
* The **utilization** will be: $\left(\frac{1}{8} + \frac{2}{5} + \frac{4}{10}\right) = \left(\frac{37}{40}\right) = 0.925 = \mathbf{92.5}\%$
* Since the least common multiple of the periods is 40, the scheduling pattern can repeat every 40 time slices. But, only 37 of those 40 time slices are used by P1, P2, or P3. Since the utilization, 92.5%, is not greater than 100%, the system is schedulable with EDF.

– Rate Monotonic Scheduling (RMS)

* A simple version of rate-monotonic analysis assumes that threads have the following properties:
  · No resource sharing (processes do not share resources, e.g. a hardware resource, a queue, or any kind of semaphore blocking or non-blocking (busy-waits))
  · Deterministic deadlines are exactly equal to periods
  · Static priorities (the task with the highest static priority that is runnable immediately preempts all other tasks)
  · Static priorities assigned according to the rate monotonic conventions (tasks with shorter periods/deadlines are given higher priorities)

· Context switch times and other thread operations are free and have no impact on the model

* Given the example:

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	2	10

Figure 6: RMS example table

* The utilization will be: $\frac{1}{8} + \frac{2}{5} + \frac{2}{10} = 0.725$
* The sufficient condition for 3 processes, under which we can conclude that the system is schedulable is: $U = 3(2^{\frac{1}{3}} - 1) = 0.77976\ldots$
* Since $0.725 < 0.77976\ldots$ the system is surely schedulable.
* But remember, this condition is not a necessary one. So we cannot say that a system with higher utilization is not schedulable with this scheduling algorithm.

- What is a Resource Access Protocol? How the following resource access protocols work?
  - Priority Inheritance Protocal (PIP)
    * When a task preempties another task, it inherits the priority from the task it preemptied for the time the task is under execution. After release of the resource the original priority is restored to what it was before the preemption.
    * Any time a lower priority task blocks a higher priority task it inherits the priority of the higher priority task. Though PIP is a protocol which gives good peformance, deadlocks and multiple blockings (chained blocking) are possible occurances.
  - Highest Locker Priority Protocol (HLP) or Immediate Priority Ceiling Protocol (IPC)
    * A ceiling is assigned to each resource (semaphore) which is the highest priority of all tasks that may use it.
    * Whenever a task starts using the resource (locks the semaphore) its priority is immediately boosted to the ceiling of the resource.
    * This protocol disables any possibility for deadlocks or multiple blocking since a task is at most only blocked once.