Implementing Task Schedulers (1)

Real-Time and Embedded Systems (M)

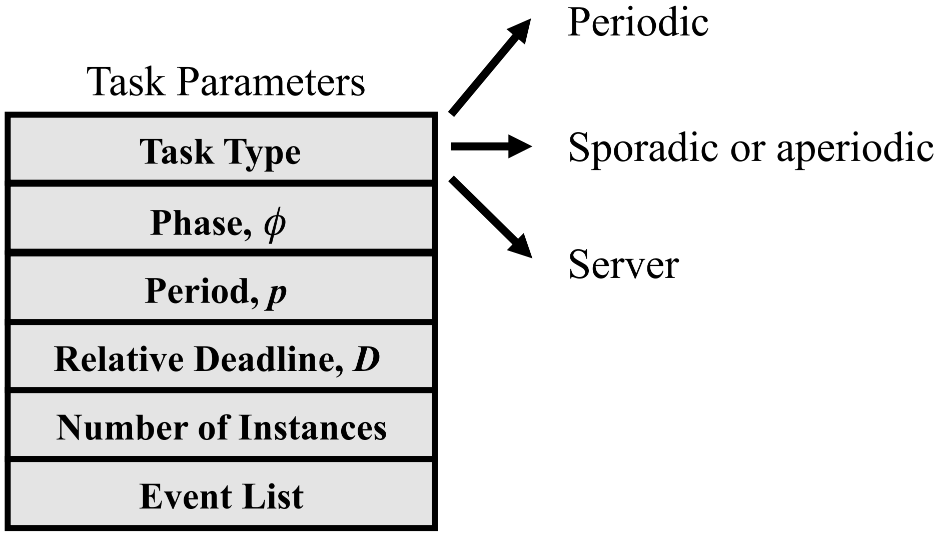
Lecture 10

UNIVERSITY
of
GLASGOW

Lecture Outline

- Implementing priority scheduling:
  - Tasks, threads and queues
  - Building a priority scheduler
  - Fixed priority scheduling (RM and DM)
  - Dynamic priority scheduling (EDF and LST)
  - Sporadic and aperiodic tasks
- Outline of priority scheduling standards:
  - POSIX 1003.1b (a.k.a. POSIX.4)
  - POSIX 1003.1c (a.k.a. pthreads)
  - Implementation details
- Use of priority scheduling standards:
  - Rate monotonic and deadline monotonic scheduling
  - User level servers to support aperiodic and sporadic tasks

Tasks and Threads

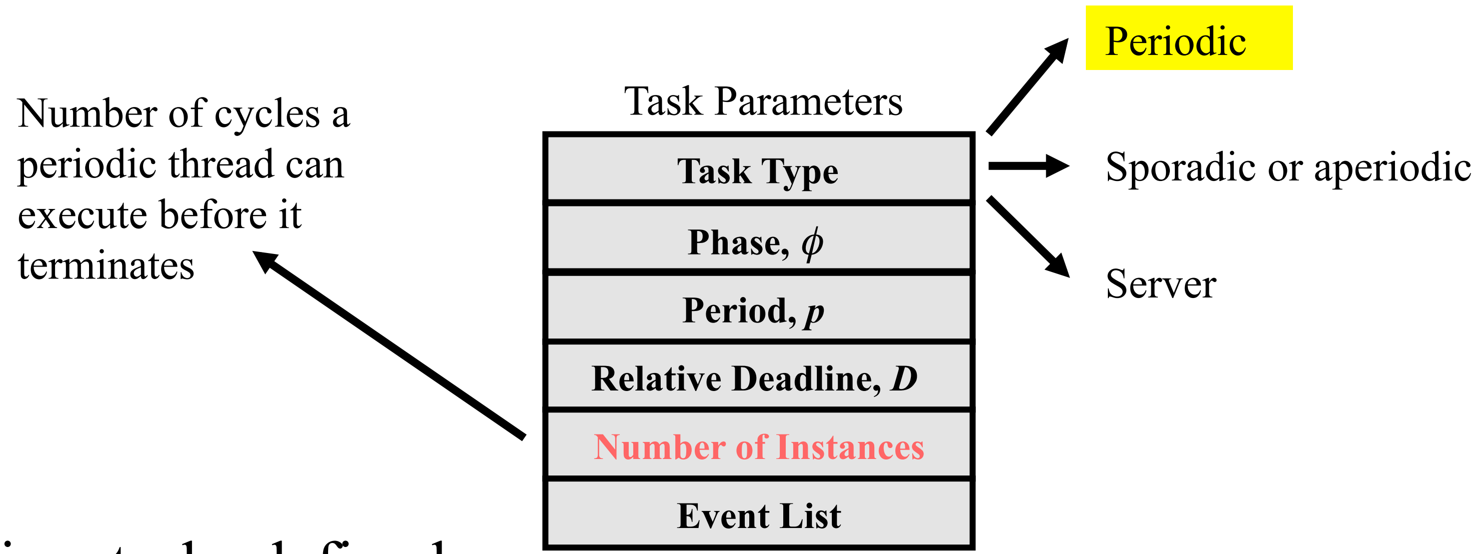


- A system comprises a set of *tasks* (or *jobs*)
- Tasks are typed, and timed with parameters (ϕ, p, e, D)

- A *thread* is the basic unit of work handled by the scheduler
  - Threads are the instantiation of tasks that have been admitted to the system
  - Acceptance test performed before admitting new tasks

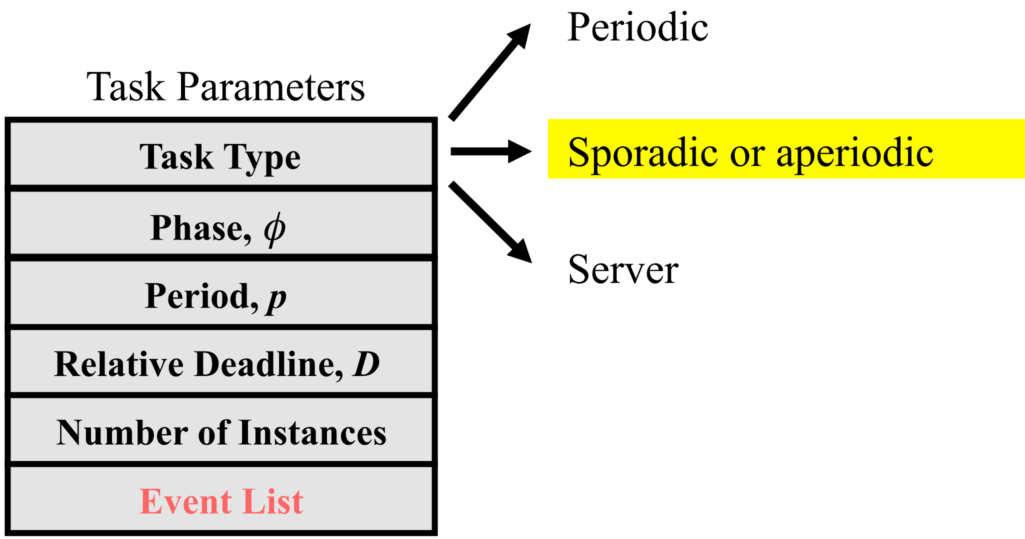
[All equally applicable to processes, rather than threads]

Periodic Threads



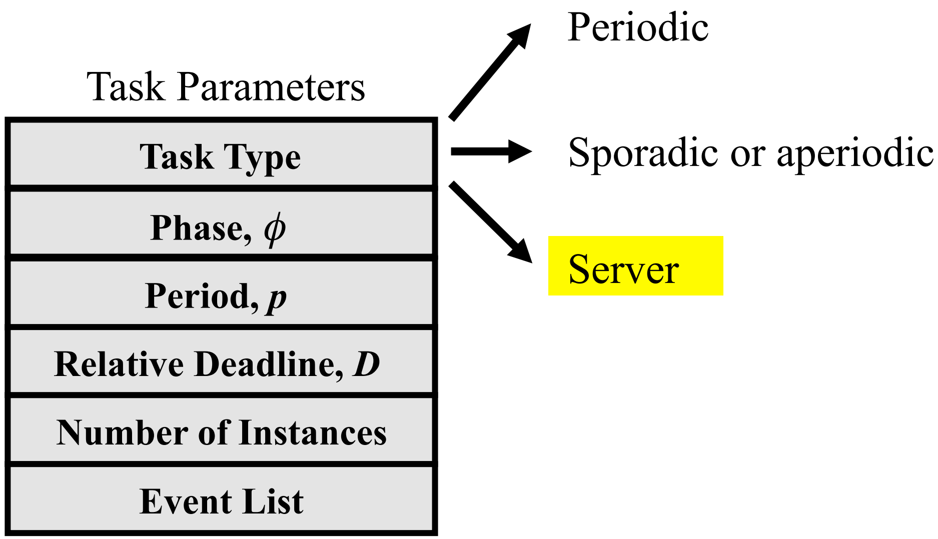
- Real time tasks defined to execute periodically
- Two implementation strategies:
  - Thread instantiated by system each period, runs a single instance of the task
    - A *periodic thread* ⇒ supported by some RTOS
    - Clean abstraction: a function that runs periodically; system handles timing
    - High overhead due to repeated thread instantiation
  - Thread instantiated once, repeatedly performs task, sleeps until next period
    - Lower overhead, but relies on the programmer to handle timing

Sporadic and Aperiodic Threads



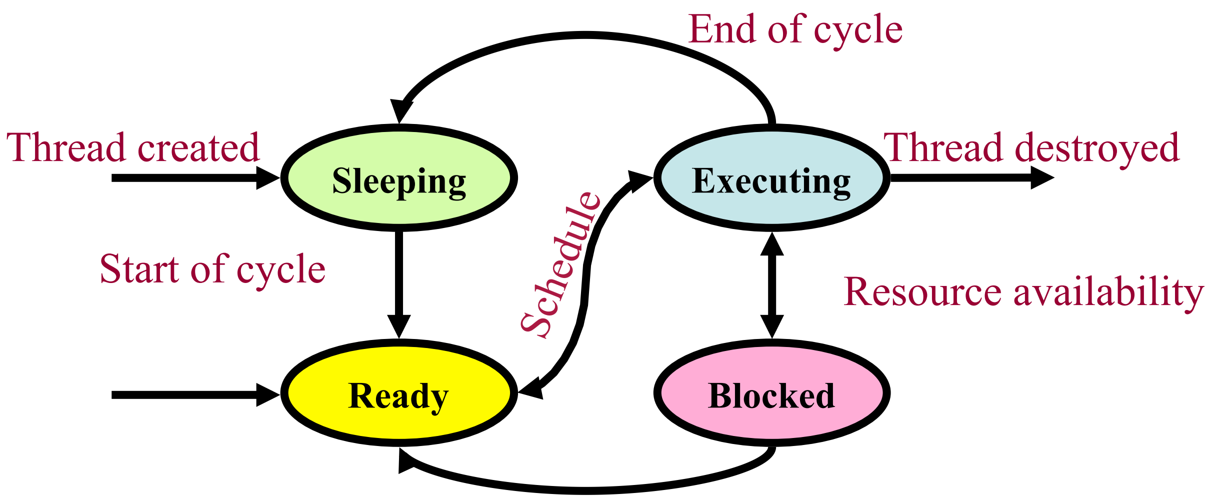
- Event list to trigger sporadic and aperiodic tasks
  - May be external (hardware) interrupts
  - May be signalled by another task
- Each instance of a sporadic or aperiodic task may be instantiated by the system as a *sporadic* or *aperiodic thread*
  - Not well supported for user-level tasks, often used in the kernel
  - Requires scheduler assistance
- Alternatively, may be implemented using a server task

Server Threads



- A server thread is a periodic thread that implements either:
  - a background server (simple, widely implemented)
  - a bandwidth preserving server (useful, but hard to implement)
- Used to implement sporadic and aperiodic threads, if not directly supported by the scheduler

Thread States and Transitions



Sleeping ⇒ Periodic thread queued between cycles

Ready ⇒ Queued at some priority, waiting to run

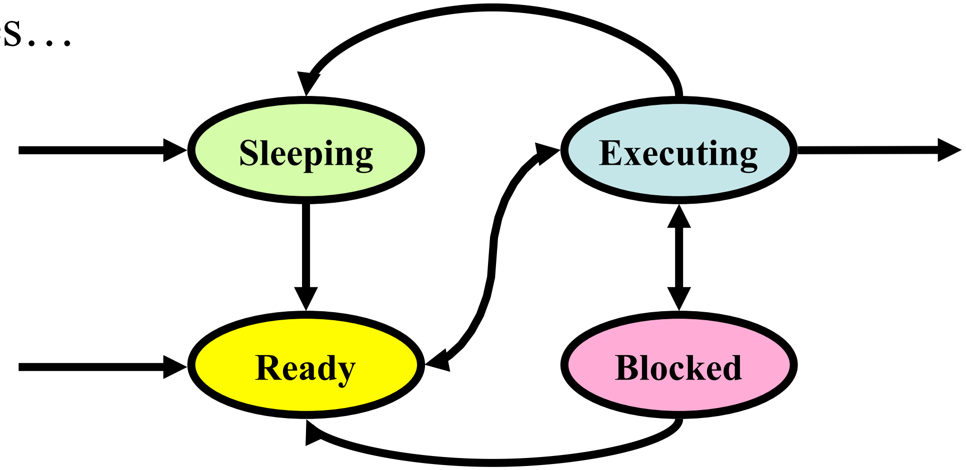
Executing ⇒ Running on a processor

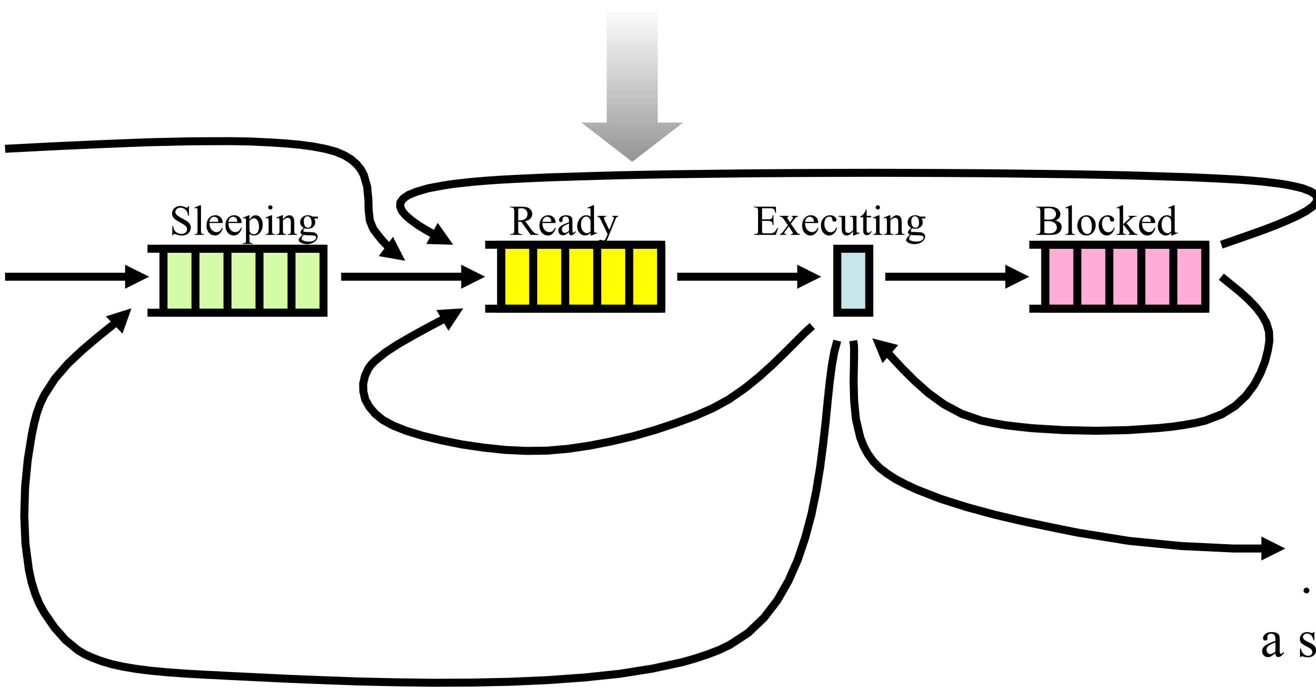
Blocked ⇒ Queued waiting for a resource

Transitions happen according to scheduling policy, resource access, external events

Mapping States onto Queues

Abstract states…

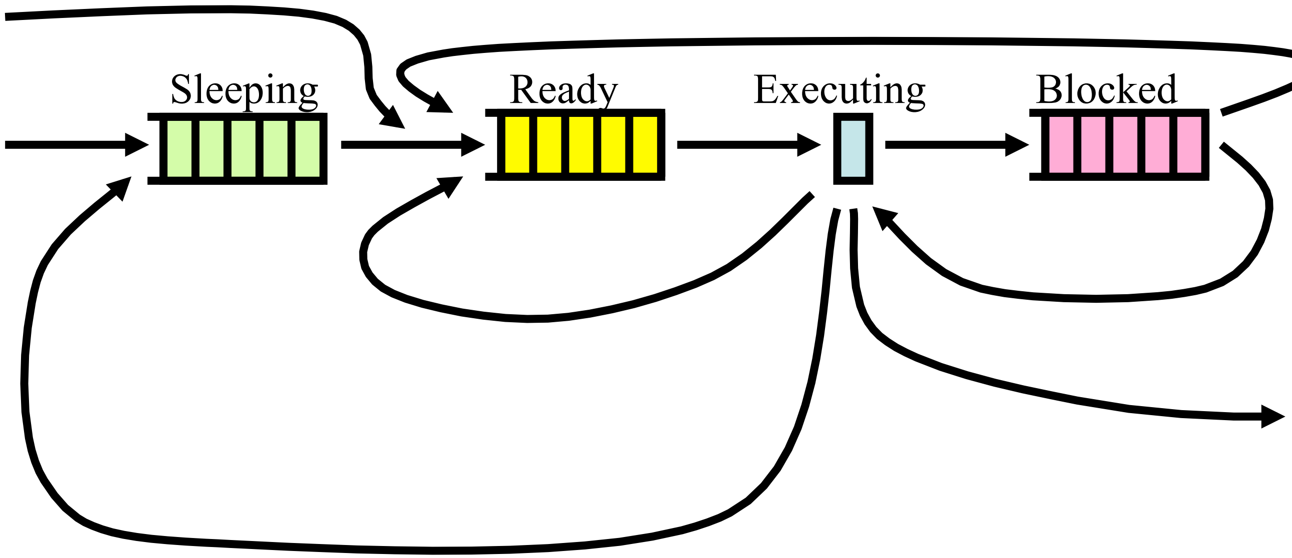




…realised as
a set of queues

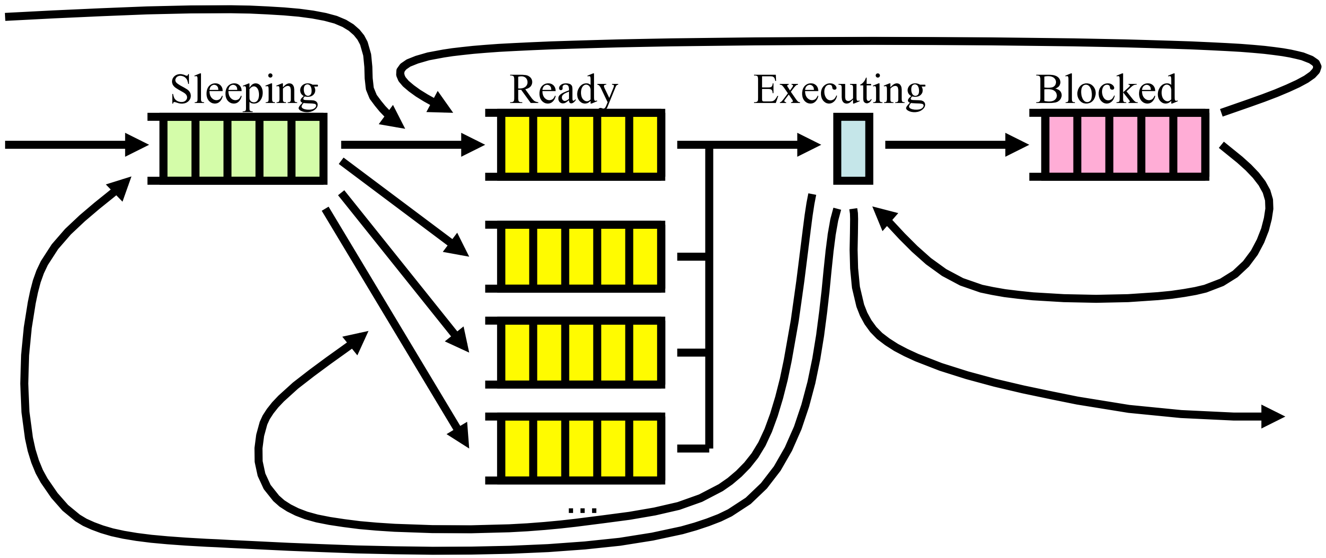
Queuing in a Priority Scheduler

- Scheduling algorithms implemented by varying the number of queues, queue selection policy and service discipline
  - How to decide which queue holds a newly released thread?
  - How are the queues ordered?
  - From which queue is the next job to execute taken?
- Different solutions for:
  - Fixed priority scheduling
  - Dynamic priority/deadline scheduling
  - Sporadic and server tasks



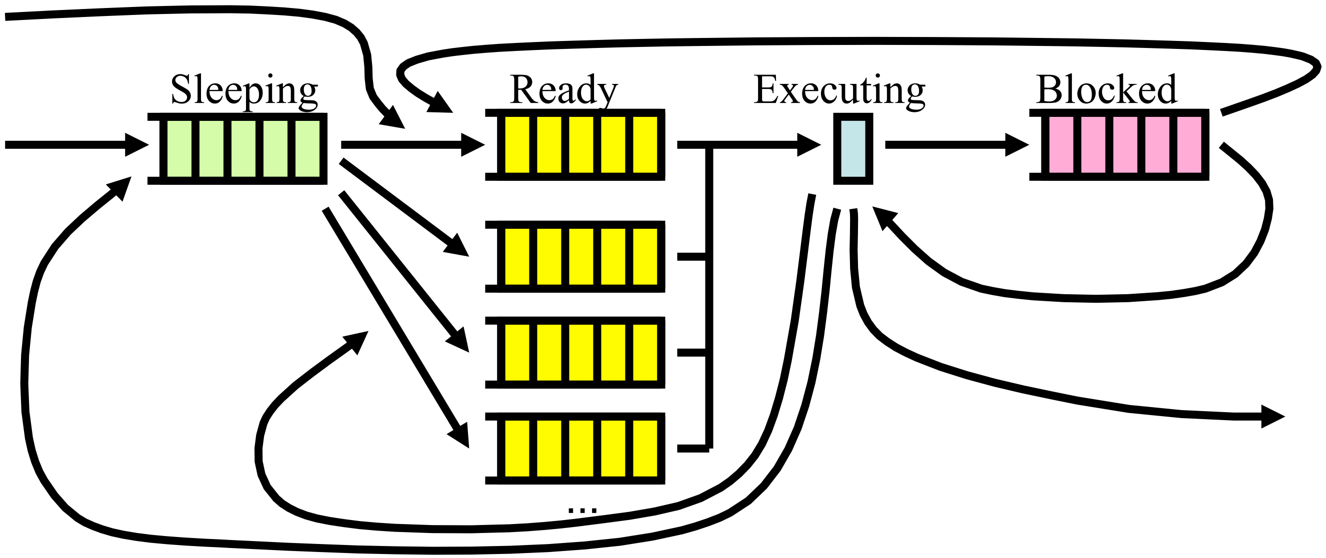
Fixed Priority Scheduling

- Provide a number of ready queues
- Each queue represents a priority level
  - Tasks inserted into queues according to priority
  - Queues serviced in FIFO or round-robin order
    - RR tasks have a budget that depletes with each clock interrupt, then yield and go to back of queue; FIFO tasks run until sleep, block or yield
- Always run task at the head of the highest priority queue that has ready tasks
- Can be used to implement rate monotonic, deadline monotonic scheduling



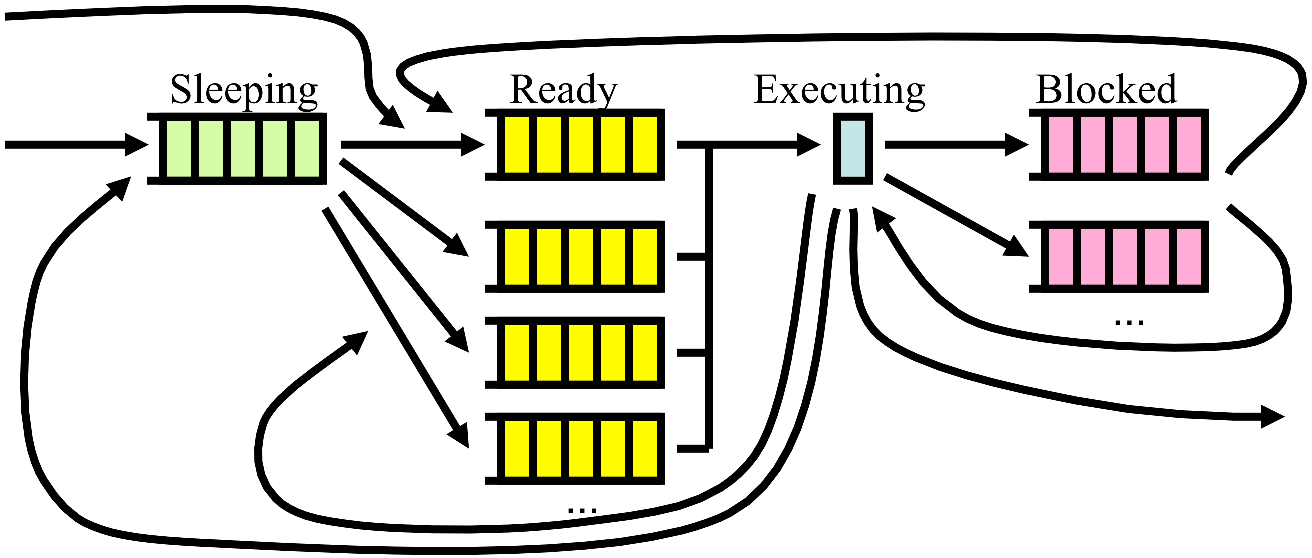
Fixed Priority Scheduling: Rate Monotonic

- Assign fixed priorities to tasks based on their period, *p*
  - short period ⇒ higher priority
- Implementation:
  - Task resides in sleep queue until released at phase, ϕ
  - When released, task inserted into a FIFO ready queue
  - One ready queue for each distinct priority
  - Always run task at the head of the highest priority queue that has ready tasks



Blocking on Multiple Events

- Typically there are several reasons why tasks may block
  - Disk I/O
  - Network
  - Inter-process communication
  - etc.
- Usually want multiple blocked queues, for different reasons
  - Reduces overheads searching a long queue to wakeup thread
- This is a typical priority scheduler provided by most RTOS



Dynamic Priority Scheduling

- Thread priority can change during execution
- Implies that threads move between ready queues
  - Search through the ready queues to find the thread changing it's priority
  - Remove from the ready queue
  - Calculate new priority
  - Insert at end of new ready queue
- Expensive operation:
  - O(*N*) where *N* is the number of tasks
  - Suitable for system reconfiguration or priority inheritance when the rate of change of priorities is slow
  - Naïve implementation of EDF or LST scheduling inefficient, since require frequent priority changes
    - Too computationally expensive
    - Alternative implementation strategies possible…

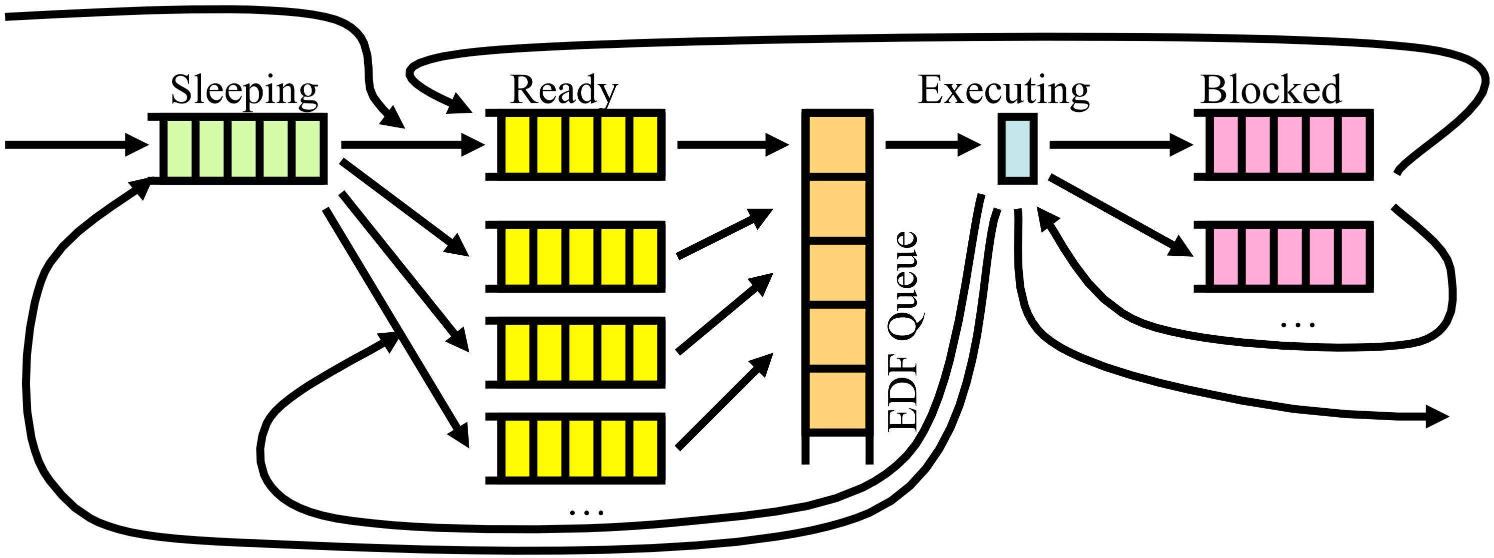
Earliest Deadline First Scheduling

- To directly support EDF scheduling:
  - When each thread is created, it's relative deadline is specified
  - When a thread is released, it's absolute deadline is calculated from it's relative deadline and release time
- Could maintain a single ready queue:
  - Conceptually simple, threads ordered by absolute deadline
  - Inefficient if many active threads, since scheduling decision involves walking the queue of *N* tasks

Earliest Deadline First Scheduling

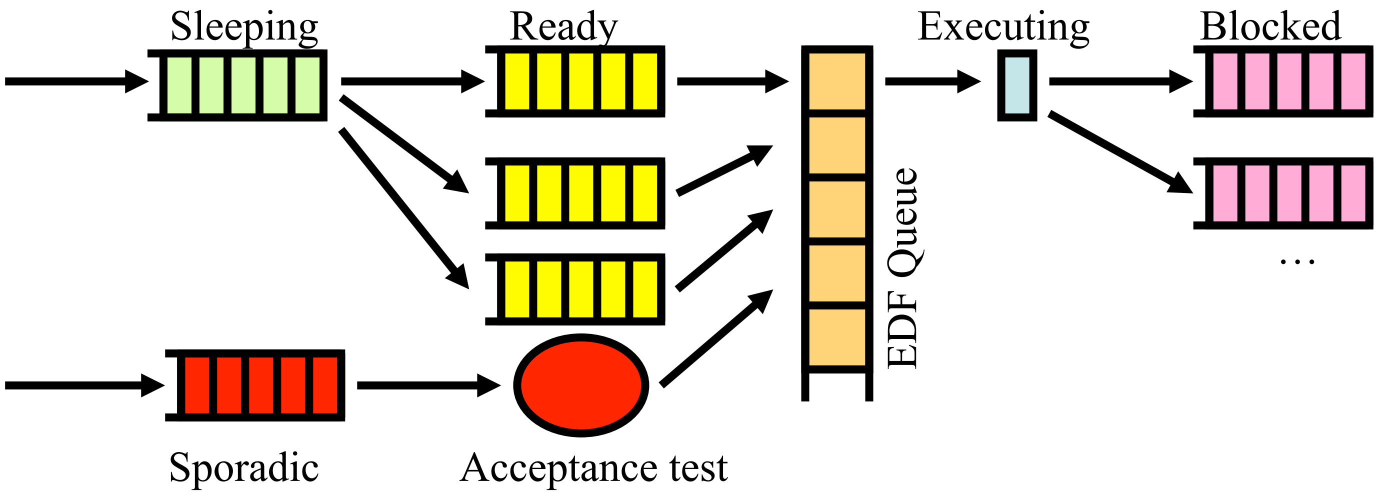
- Maintain a ready queue for each *relative* deadline
  - Tasks enter these queues in order of release
  - $\Omega' < N$ queues

- Maintain a queue, sorted by *absolute* deadline, pointing to tasks at the head of each ready queue
  - Updated each time a task completes
  - Updated when a task added to an empty ready queue
  - Always execute the task at the head of this queue
  - More efficient, since only perform a linear scan through active tasks



Scheduling Sporadic Tasks

- Recall: sporadic tasks have hard deadlines but unpredictable arrival times
- Straight-forward to schedule using EDF:
  - Add to separate queue of ready sporadic tasks on release
  - Perform acceptance test
  - If accepted, insert into the EDF queue according to deadline
- Difficult if using fixed priority scheduling:
  - Need a bandwidth preserving server



Scheduling Aperiodic Tasks

- Trivial to implement in as a background server, using a single lowest priority queue
  - All the problems described in lecture 7:
    - Excessive delay of aperiodic jobs
    - Potential for priority inversion if the aperiodic jobs use resources
    - [Linux has exactly this issue with idle-jobs]
  - Better to use a bandwidth preserving server

Bandwidth Preserving Servers

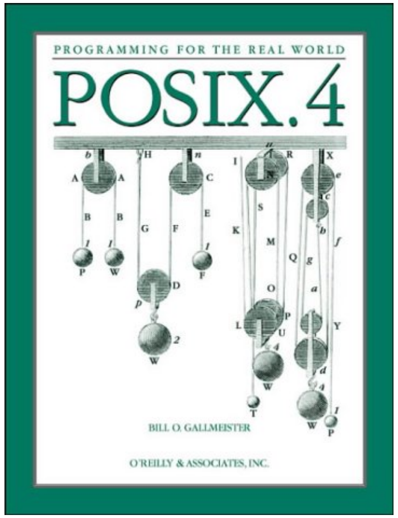
- Server scheduled as a periodic task, with some priority
- When ready and selected to execute, given scheduling quantum equal to the current budget
  - Runs until pre-empted or blocked; or
  - Runs until the quantum expires, sleeps until replenished

Unlike RR scheduling which yields when a quantum expires

- At each scheduling event in the system
  - Update budget consumption considering:
    - time for which the BP server has executed
    - time for which other tasks have executed
    - algorithm depends on BP server type
  - Replenish budget if necessary
  - Keep remaining budget in the thread control block
  - Fairly complex calculations, e.g. for sporadic server
- Not widely supported… typically have to use background server

Standards for Real-Time Scheduling

- There are two widely implemented standards for real-time scheduling
  - POSIX 1003.1b (a.k.a. POSIX.4)
  - POSIX 1003.1c (a.k.a. pthreads)
- Support a sub-set of scheduler features we have discussed
  - A least-common denominator interface, design to this and the system will be easily portable



- Most RTOS also implement a non-portable "native" interface, with more features, higher performance

POSIX 1003.1b Real-Time Scheduling API

Key features:

- Get/set scheduling policy
- Get/set parameters
- Yield the processor

```
#include <unistd.h>
#ifdef _POSIX_PRIORITY_SCHEDULING
#include <sched.h>

struct sched_param {
   int sched_priority;
   …
}

int sched_setscheduler(pid_t pid, int policy,
                                          struct sched_param *sp);
int sched_getscheduler(pid_t pid);
int sched_getparam(pid_t pid, struct sched_param *sp);
int sched_setparam(pid_t pid, struct sched_param *sp);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_yield(void);
#endif
```

POSIX 1003.1b Real-Time Scheduling API

- POSIX 1003.1b provides three scheduling policies::
  - `SCHED_FIFO`: Fixed priority, pre-emptive, FIFO scheduler
  - `SCHED_RR`: Fixed priority, pre-emptive, round robin scheduler
    - Use `sched_rr_get_interval(pid_t pid, struct timespec *t)` to find the scheduling time quantum
  - `SCHED_OTHER`: Unspecified (often the default time-sharing scheduler)

- Implementations can support alternative schedulers
- Scheduling parameters are defined in `struct sched_param`
  - Currently just priority; other parameters can be added in future
  - Not all parameters applicable to all schedulers
    - E.g. `SCHED_OTHER` doesn't use priority

- A process can `sched_yield()` or otherwise block at any time

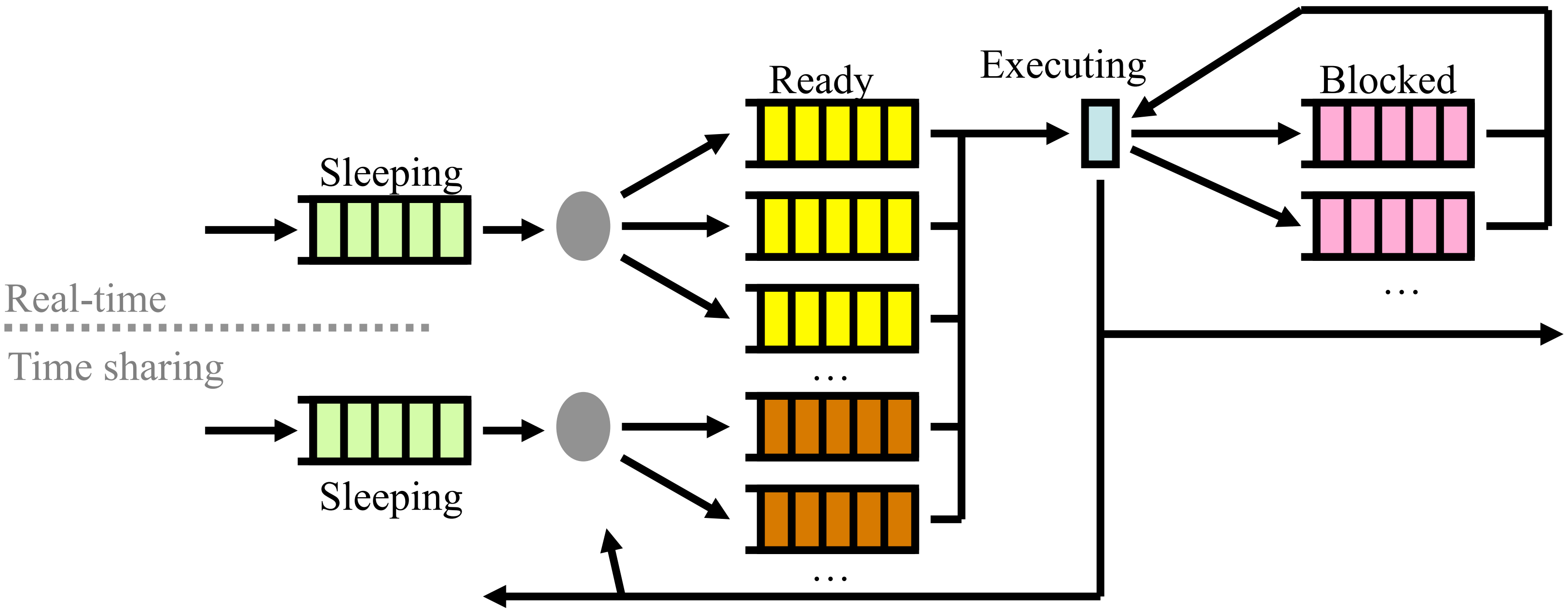
POSIX APIs: Priority

- POSIX 1003.1b provides (largely) fixed priority scheduling
  - Priority can be changed using **`sched_set_param()`**, but this is high overhead and is intended for reconfiguration rather than for dynamic scheduling
  - No direct support for dynamic priority algorithms (e.g. EDF)

- Limited set of priorities:
  - Use **`sched_get_priority_min()`**, **`sched_get_priority_max()`** to determine the range
  - Guarantees at least 32 priority levels

Mapping onto Priority Queues

- Tasks using **`SCHED_FIFO`** and **`SCHED_RR`** map onto a set of priority queues as described previously
  - Relatively small change to existing time-sharing scheduler
- Additional queues support **`SCHED_OTHER`** if providing a time sharing service
  - Time sharing tasks only progress if no active real-time task
  - Beware: a rogue real-time task can lock out time sharing tasks



POSIX 1003.1c Real-Time Scheduling API

```
#include <unistd.h>
#ifdef _POSIX_THREADS
#include <pthread.h>
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING
```

Check for presence of pthreads

```
int pthread_attr_init(pthread_attr_t *attr);
```

Same scheduling policies and parameters as POSIX 1003.1b

```
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *p);
int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *p);


int pthread_create(pthread_t      *thread,
                   pthread_attr_t *attr,
                   void *(*thread_func)(void*),
                   void   *thread_arg);
int pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
```

Returns thread ID

Pointer to function that runs as the thread, and it's argument

Detecting POSIX Support

- If you need to write portable code, e.g. to run on Unix/Linux systems, you can check the presence of POSIX 1003.1b via pre-processor defines:

```
#include <stdio.h>
#include <unistd.h>
#ifdef _POSIX_PRIORITY_SCHEDULING
   printf("POSIX 1003.1b\n");
#endif
#ifdef _POSIX_THREADS
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING
   printf("POSIC 1003.1c\n");
#endif
#endif
```

- Access to POSIX real-time extensions is usually privileged on general purpose systems (e.g. suid root on Unix)
  - Remember to drop privileges!

Using POSIX Scheduling: Rate Monotonic

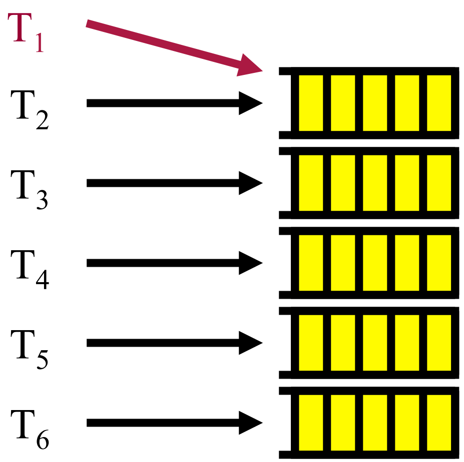
- Rate monotonic and deadline monotonic schedules can naturally be implemented using POSIX primitives
    1. Assign priorities to tasks in the usual way for RM/DM
    2. Query the range of allowed system priorities

       `sched_get_priority_min()`

       `sched_get_priority_max()`
    3. Map task set onto system priorities
        - Care needs to be taken if there are large numbers of tasks, since some systems only support a few priority levels
    4. Start tasks using assigned priorities and **`SCHED_FIFO`**

Using POSIX Scheduling: Rate Monotonic

- When building a rate monotonic system, ensure there are as many ready queues as priority levels
- May be limited by the operating system is present, and need priority levels than there are queues provided



Implication: non-distinct priorities

Some tasks will be delayed relative to the "correct" schedule

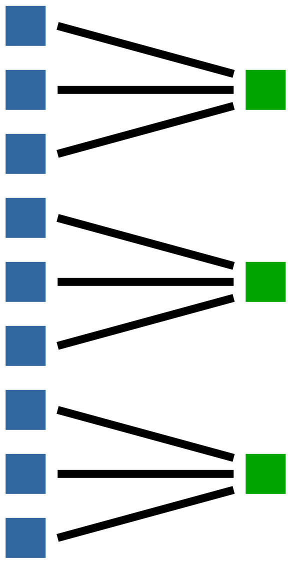
A set of tasks $T_E(i)$ is mapped to the same priority queue as task T_i

This may delay T_i up to $\sum_{T_k \in T_E(i)} e_k$

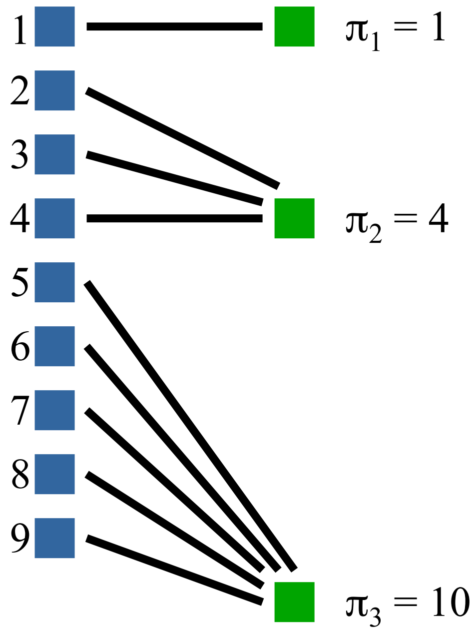
Schedulable utilization of system will be reduced

Using POSIX Scheduling: Rate Monotonic

- How to map a set of tasks needing Ω_n priorities onto a set of Ω_s priority levels, where $\Omega_s < \Omega_n$?



Uniform mapping
$Q = | \Omega_n / \Omega_s |$
tasks map onto each system priority level



Constant Ratio mapping
$k = (\pi_{i-1}+1)/\pi_i$
tasks where k is a constant map to each system priority with weight, π_i

Constant ratio mapping better preserves execution times of high priority jobs

Using POSIX Scheduling: EDF

- EDF scheduling is not supported by POSIX
- Conceptually would be simple to add:
  - A new scheduling policy
  - A new parameter to specify the relative deadline of each task
- But, requires the kernel to implement deadline scheduling
  - POSIX grew out of the Unix community
  - Unlike priority scheduling, difficult to retro-fit deadline scheduling onto a Unix kernel…

Periodic Tasks

- Much of the previous discussion has assumed periodic tasks scheduled by the operating systems
- However, direct support for periodic tasks is rare
  - RT-Mach
  - Not one of the standard real-time POSIX extensions
- Implement instead using a looping task:

```
…set repeating wake up timer
while (1) {
 …suspend until timer expires
 …do something
}
```

- Beware drift, due to inaccurate timers

Scheduling Aperiodic and Sporadic Tasks

- Difficult to implement aperiodic and sporadic tasks using POSIX interface since:
  - No support for EDF scheduling
  - No support for bandwidth preserving server
- Can use background server thread at the lowest priority:
  - One thread with a queue of functions to execute
    - Work added to the queue by other threads
  - One thread per event, blocked on the event
  - Take care about priority inversion when accessing resources
- Bandwidth preserving server cannot easily be simulated:
  - Need to measure execution time of the server, but:
    - Inaccurate
    - Often lacking resolution
    - Implies: may underestimate BP server run-time, and overuse resources
  - No way of knowing which other tasks have run, needed for the sporadic server algorithm

Summary of POSIX Scheduling

- Good support for fixed priority scheduling
  - Rate and deadline monotonic
  - Background server can be used for aperiodic tasks
- No support for earliest deadline scheduling, sporadic tasks
  - Some specialised RTOS support these
  - Earliest deadline scheduling more widely used to schedule network packets

Summary

- Implementing priority scheduling:
  - Tasks, threads and queues
  - Building a priority scheduler
  - Fixed priority scheduling (RM and DM)
  - Dynamic priority scheduling (EDF and LST)
  - Sporadic and aperiodic tasks
- Outline of priority scheduling standards:
  - POSIX 1003.1b (a.k.a. POSIX.4)
  - POSIX 1003.1c (a.k.a. pthreads)
  - Implementation details
- Use of priority scheduling standards:
  - Rate monotonic and deadline monotonic scheduling
  - User level servers to support aperiodic and sporadic tasks