

Scheduling in Practice

Colin Perkins

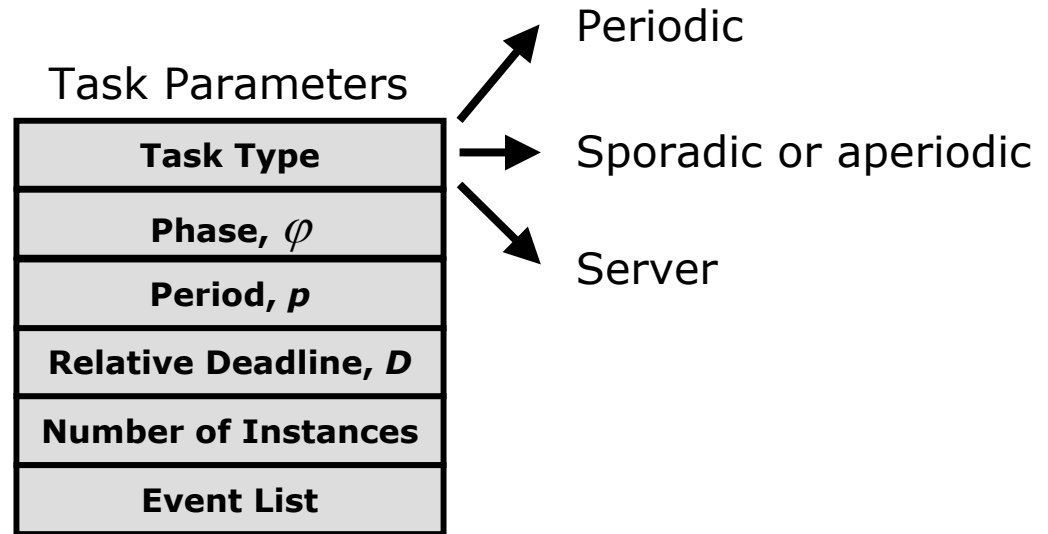
<http://csparks.org/teaching/2003-2004/rtes4/lecture12.pdf>

Reminder: Problem set 2 due at 5pm today!

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

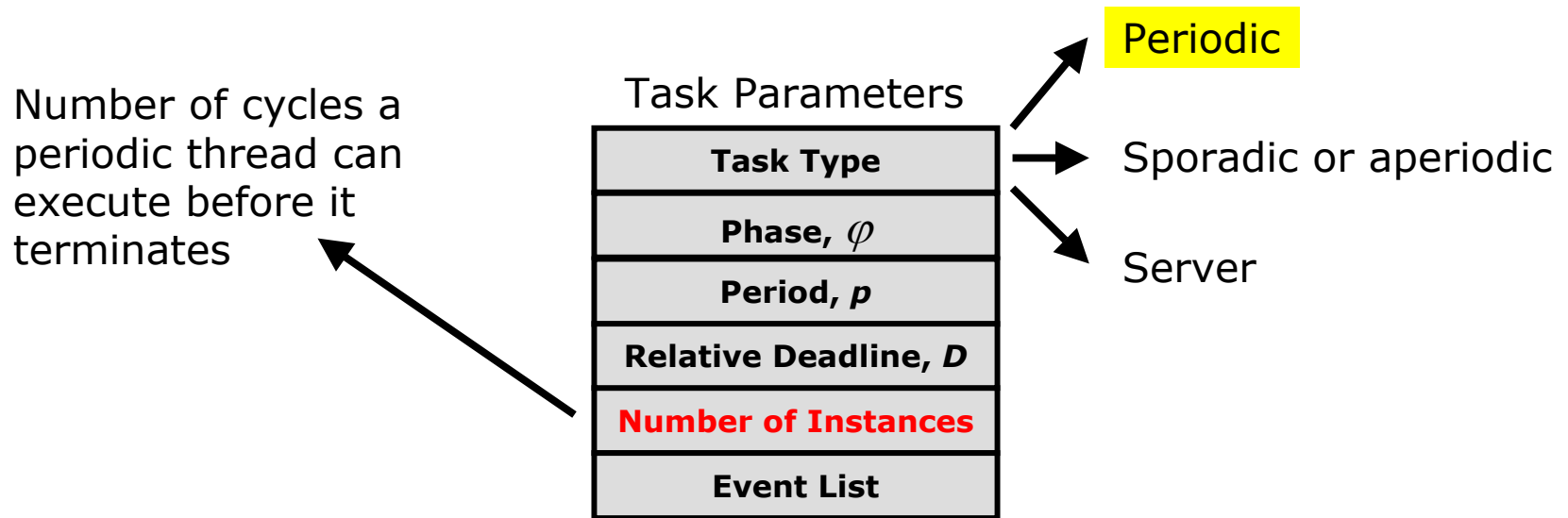
Tasks and Threads



- 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

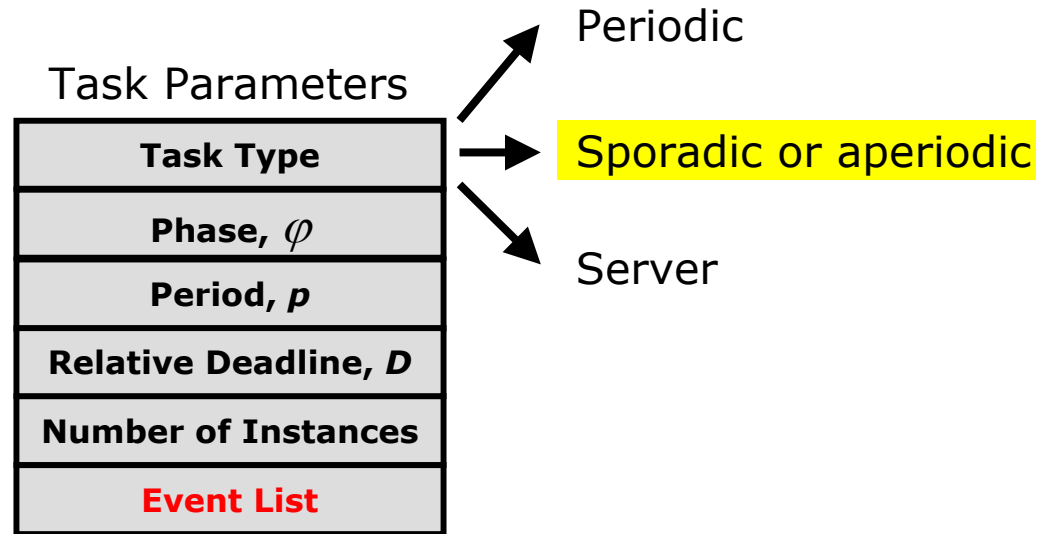
[All equally applicable to processes, rather than threads]

Tasks and Threads



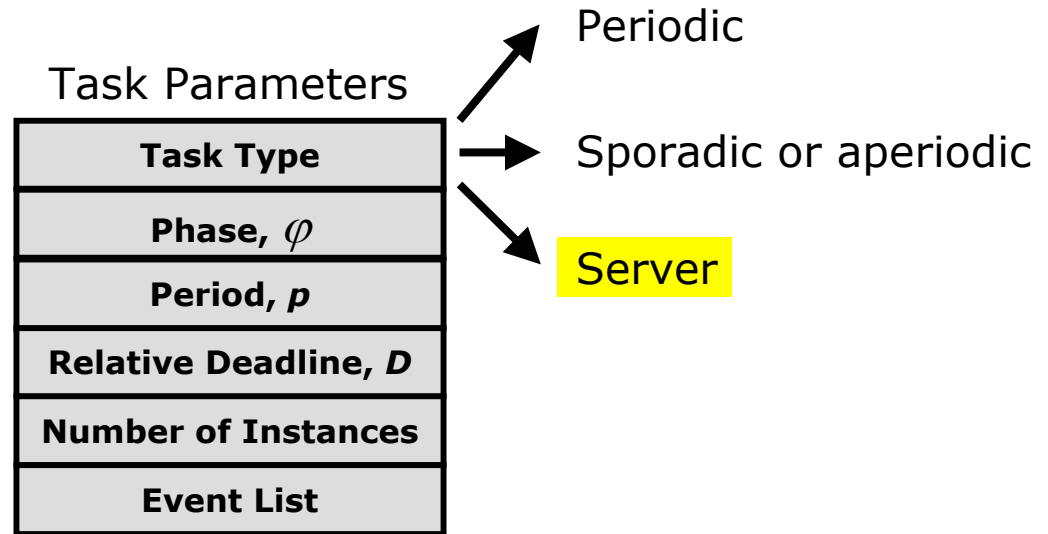
- Real time tasks commonly defined to execute periodically
- Two implementation strategies:
 - A thread is instantiated by the system each period, and runs a single instance of the task
 - A **periodic thread**, not widely supported
 - Clean abstraction: a function that runs periodically
 - high overhead due to repeated thread instantiation
 - Operating system handles timing
 - A thread is instantiated once, and repeatedly performs the task then sleeps until the beginning of the next period
 - Lower overhead, but relies on the programmer to handle timing

Tasks and Threads



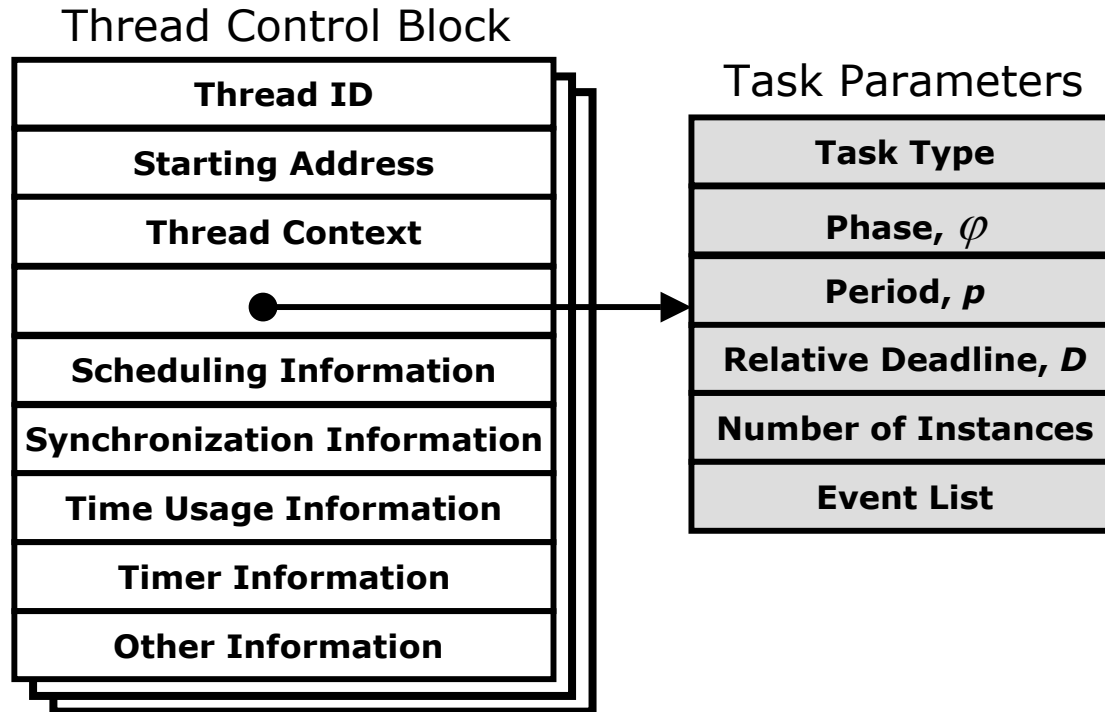
- Event list to trigger sporadic and aperiodic tasks
 - May be external (hardware) interrupts
 - May be signalled by another task
- Each instance of a sporadic or aperiodic task may be instantiated by the system as a **sporadic** or **aperiodic thread**
 - Not well supported
 - Requires scheduler assistance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

Tasks and Threads



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

Threads

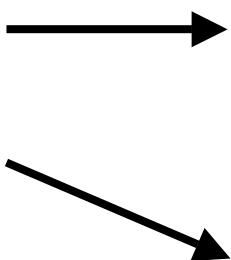


- When a thread is instantiated, the **thread control block** is created referencing the task, and maintaining the thread ID, starting address, register context and other state

Threads

Thread Control Block

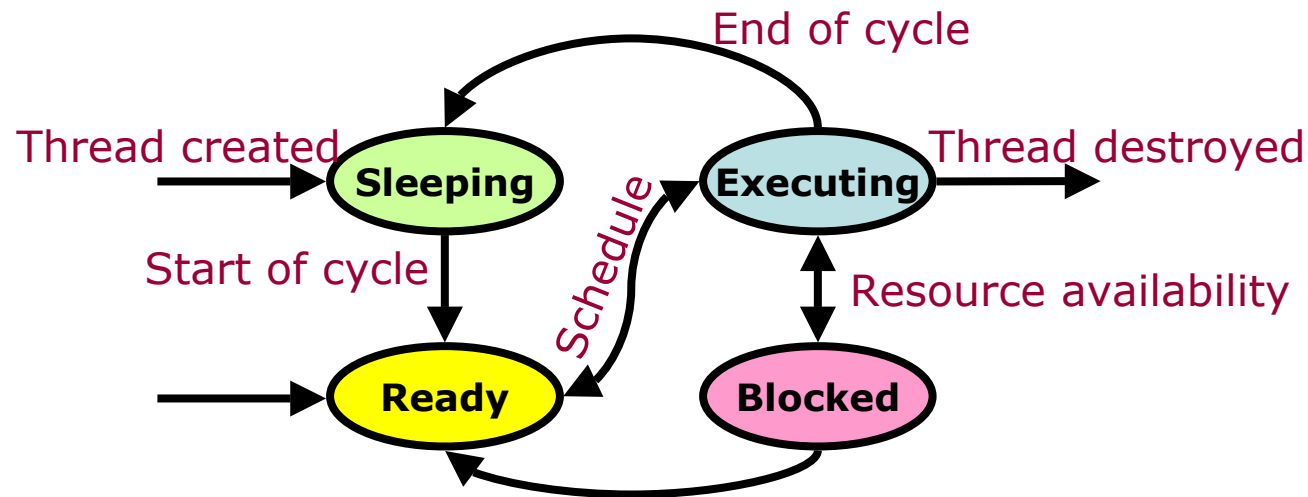
Thread ID
Starting Address
Thread Context
Scheduling Information
Synchronization Information
Time Usage Information
Timer Information
Other Information

- 
- Current priority
 - Normal priority
 - Absolute deadline
 - Remaining time quantum

- Used to implement server threads
- Budget
- Replenishment/Consumption rules

Of interest today is the scheduling and time usage data...

Thread States and Transitions



Sleeping \Rightarrow Periodic thread queued between cycles

Ready \Rightarrow Queued at some priority, waiting to run

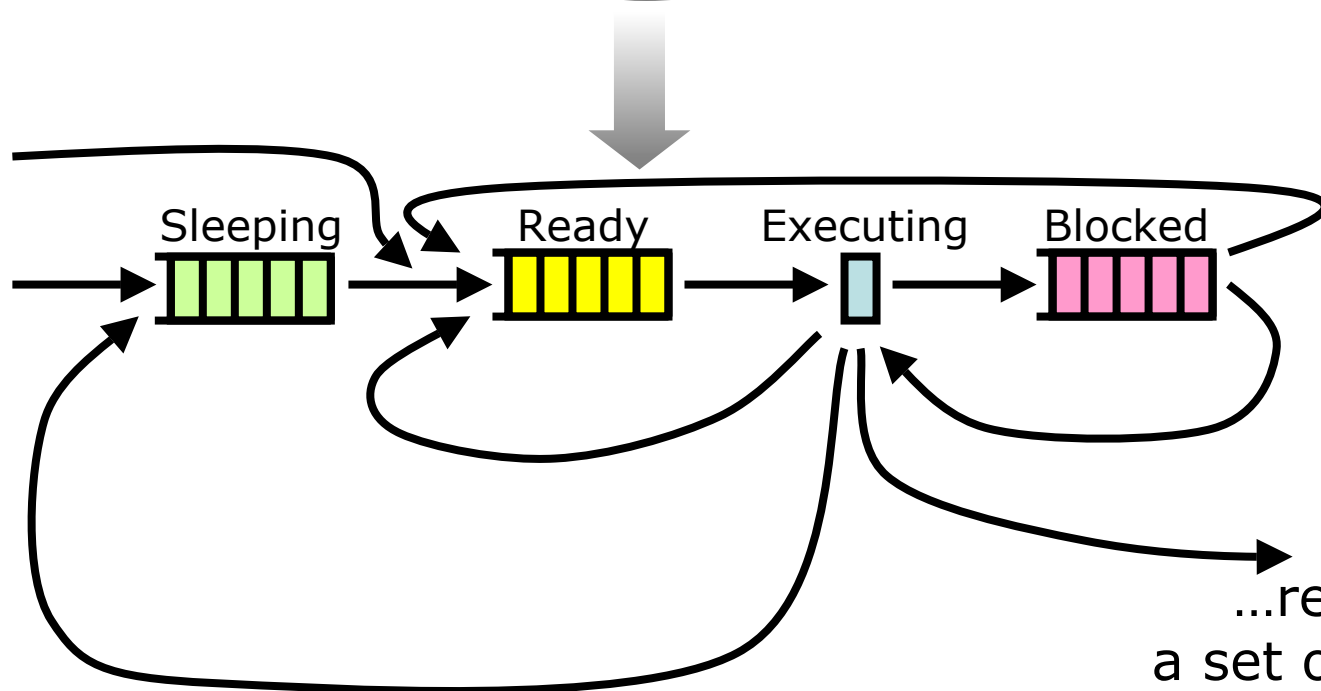
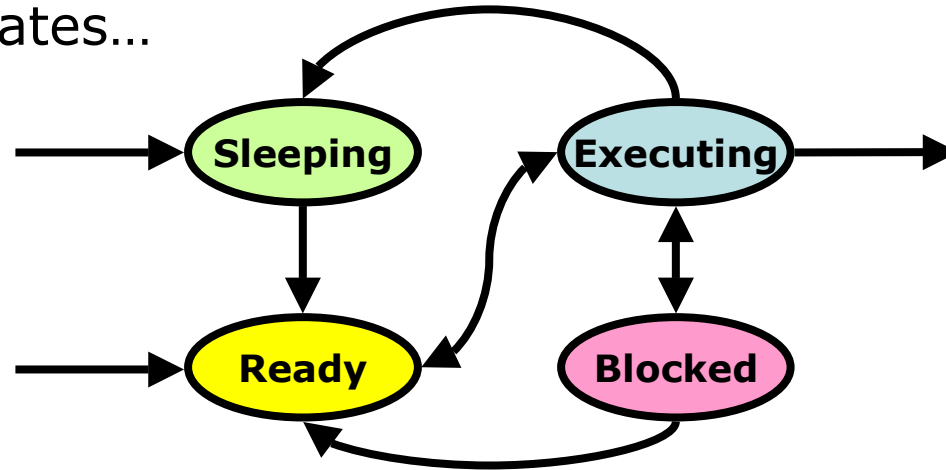
Executing \Rightarrow Running on a processor

Blocked \Rightarrow Queued waiting for a resource

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

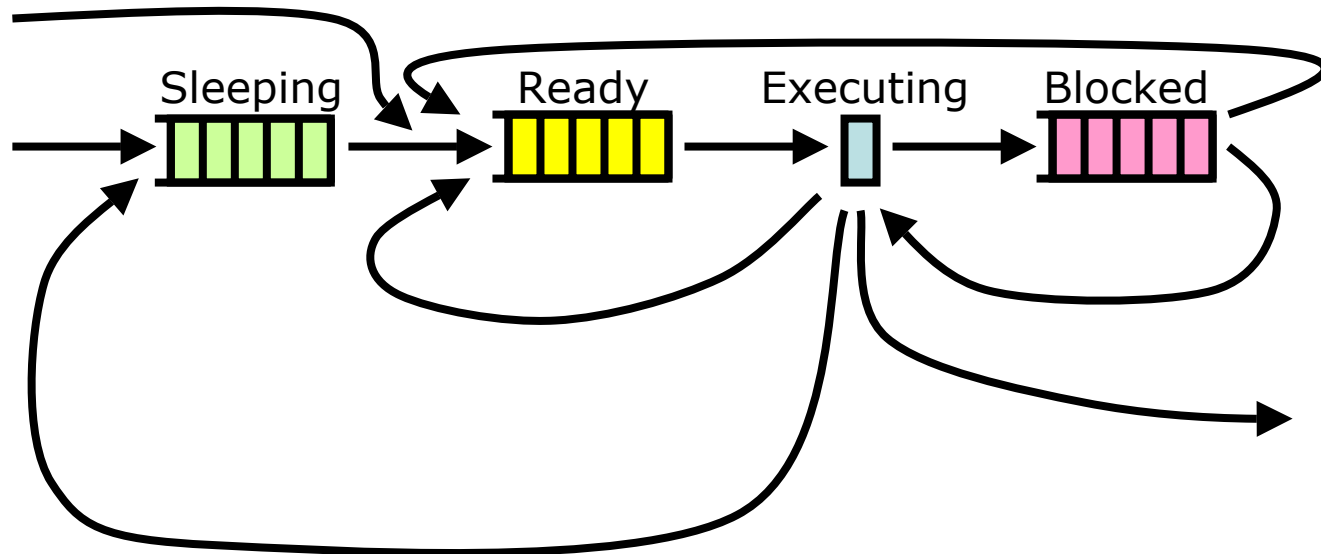
Mapping States onto Queues

Abstract states...



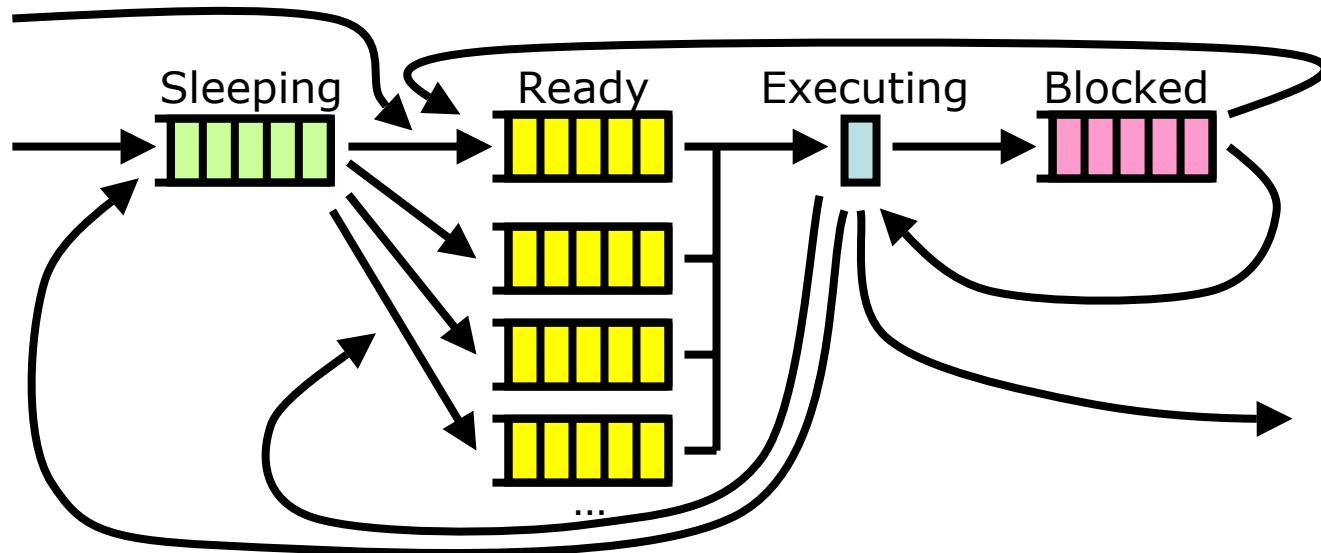
Queuing in a Priority Scheduler

- 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



Fixed Priority Scheduling

- Theoretical scheduling complexity is $O(\Omega)$
 - Where Ω is the number of priorities/ready queues
 - Commonly 256 queues provided
 - Since you have to look at each queue to determine what to run
- Requires $n = \Omega/k + \log_2 k - 1$ comparisons
 - 256 priority levels ($\Omega = 256$)
 - 32 bit processor ($k = 32$)
 - $n = 256/32 + 5 - 1$
 $= 12$
 - Implement as a bit mask of runnable priorities:

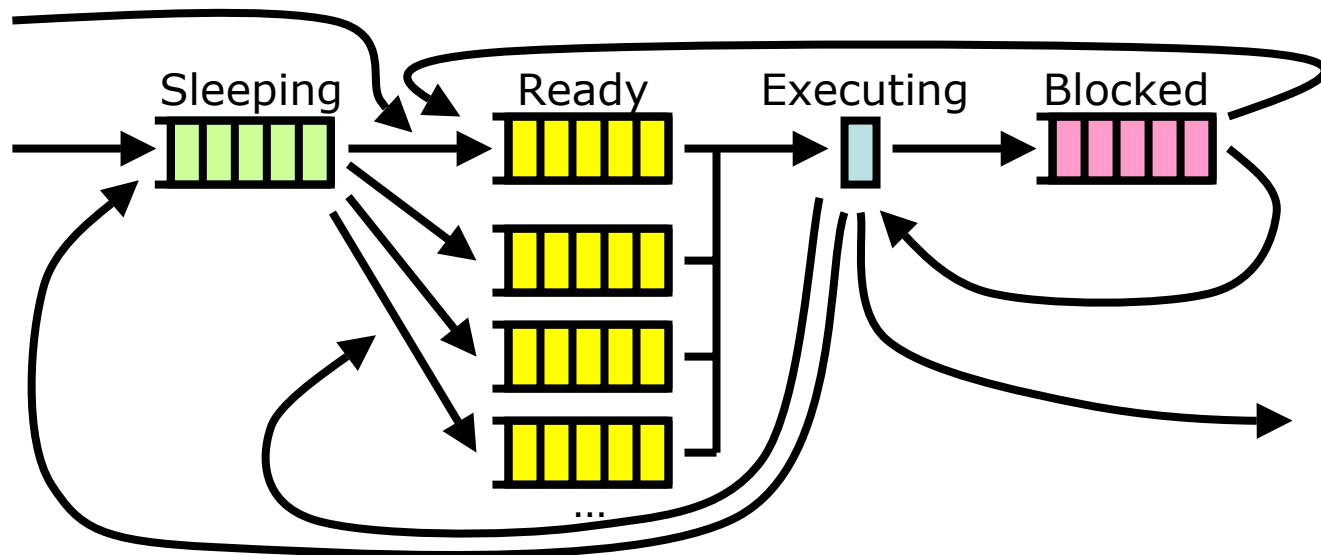


Low

High

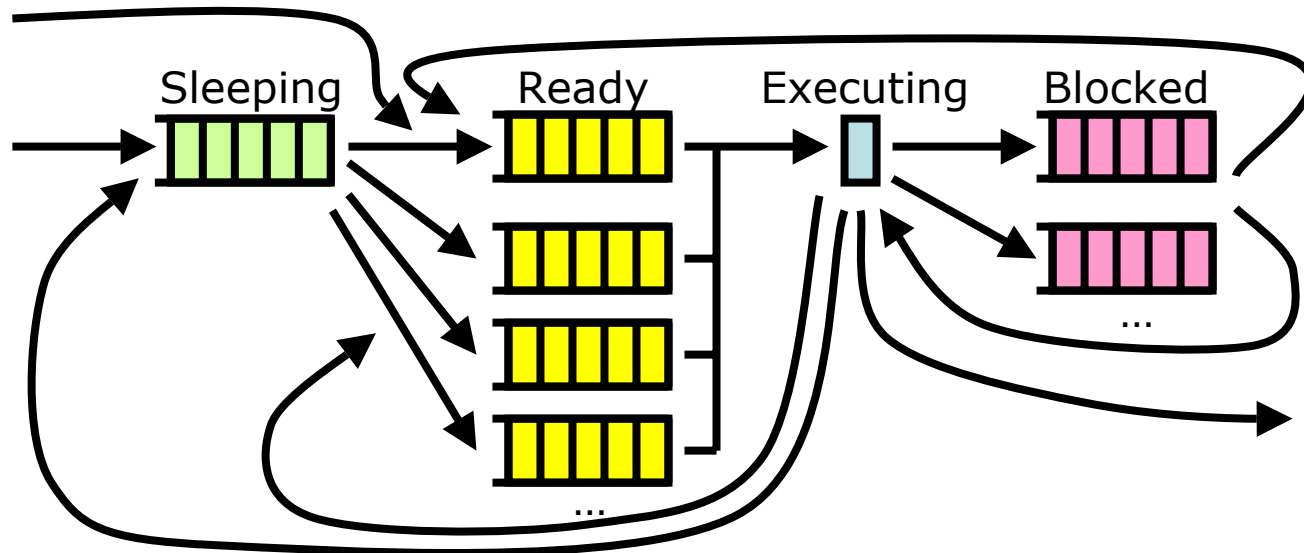
Example: Rate Monotonic

- Assign fixed priorities to tasks based on their period, p
 - short period \Rightarrow 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

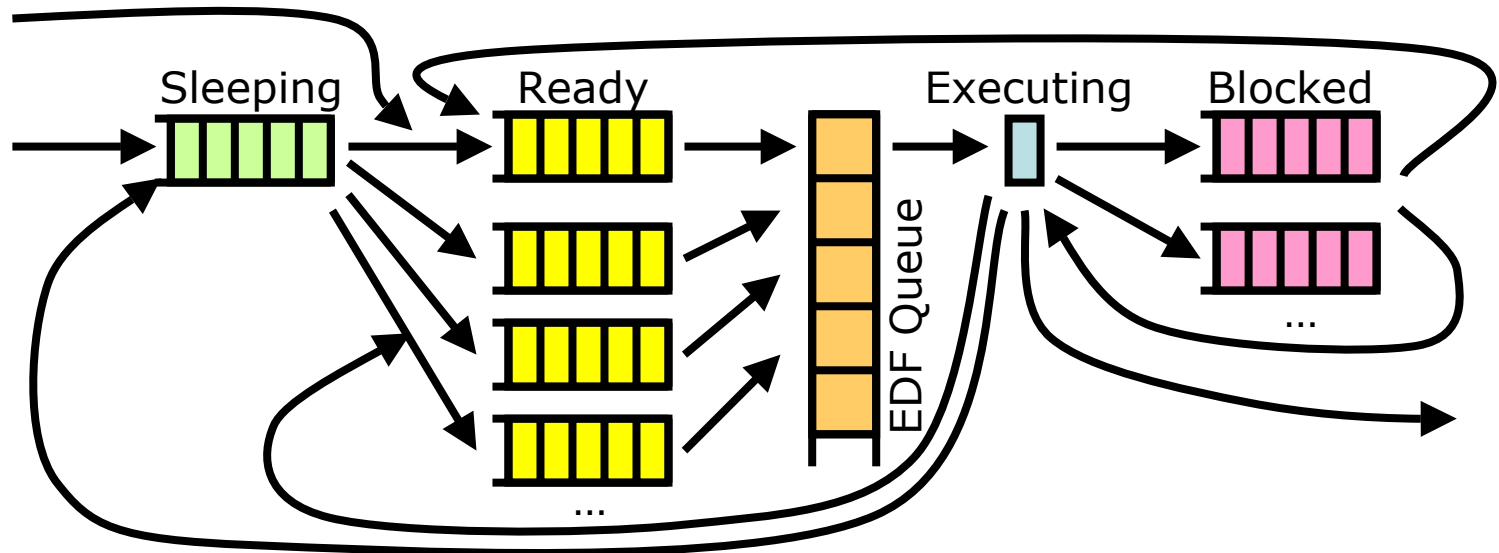
- Thread priority can change during execution
- Implies that threads move between ready queues
 - Search through the ready queues to find the thread changing it's priority
 - Remove from the ready queue
 - Calculate new priority
 - Insert at end of new ready queue
- Expensive operation:
 - $O(N)$ where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

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
 - Scheduling decision involves walking the queue
 - $O(N)$ where N is the number of tasks

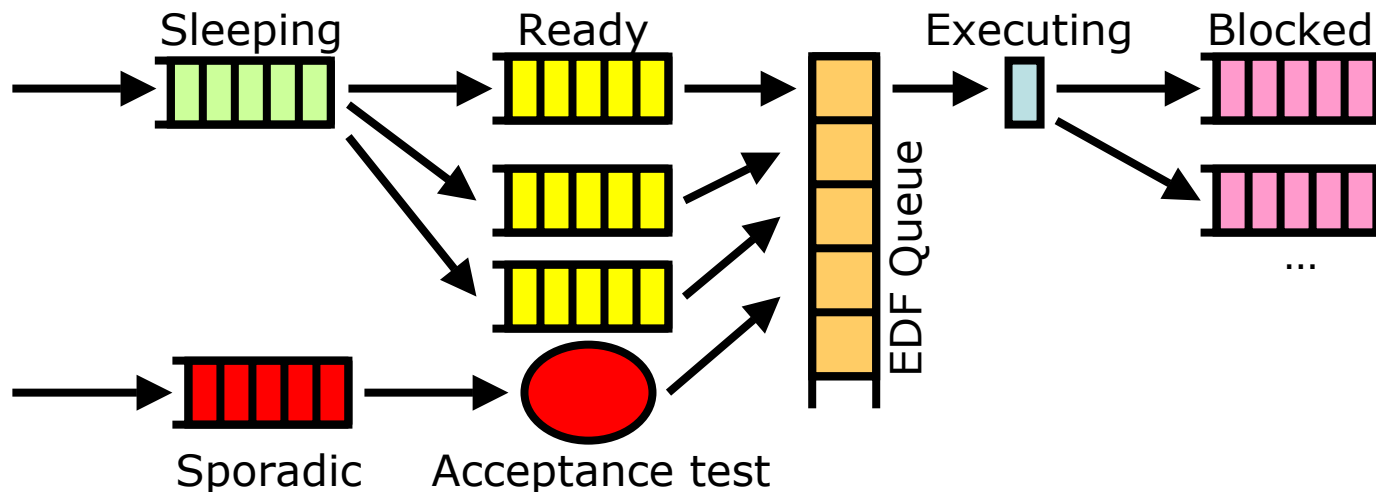
Earliest Deadline First Scheduling

- Maintain a ready queue for each relative deadline
 - Tasks enter these queues in order of release
 - Ω' 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
- Scheduling decision is $O(\Omega')$ where $\Omega' < N$



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

- Recall: aperiodic tasks have unpredictable arrival times, but no deadline
- Trivial to implement in as a background server, using a single lowest priority queue
 - All the problems described in lecture 8:
 - Excessive delay of aperiodic jobs
 - Potential for priority inversion if the aperiodic jobs use resources
 - Linux community has exactly this issue with idle-jobs
- As discussed in lecture 8, better to use a bandwidth preserving server

Implementing Bandwidth Preserving Servers

- Recall definition of BP server:
 - a periodic server, defined by budget consumption and replenishment rules
 - consume when executing
 - consume when idle in some cases
 - executes when it has budget and work to do
 - sleeps when budget expires or idle
 - moves onto ready queue when
 - budget replenished, if work to do
 - work becomes available
- Several different types of BP server
 - Different replenishment and consumption rules

Implementing Bandwidth Preserving Servers

- BP server is scheduled as a periodic task, with some priority
- When ready and selected for execution, given a scheduling quantum equal to the current budget
 - Runs until pre-empted or blocked; or
 - Runs until the quantum expires, **then sleeps until replenished**
- 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
 - “Time usage information” seen earlier
 - Fairly complex calculations, e.g. for sporadic server
 - Possible, since the scheduler knows the run times of all tasks
 - Difficult to be accurate, if the BP server runs for partial intervals
 - May need to use a high resolution timer, instead of the usual scheduling clock
- Not widely supported...

**Not like RR scheduling
which yields when
quantum expires**

Standards for Priority Scheduling

- POSIX 1003.1b (a.k.a. POSIX.4)
- POSIX 1003.1c (a.k.a. pthreads)
- Implementation details



The POSIX 1003.1b Real-Time Scheduling API

```
#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
```

Key features:

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

Policy is one of **SCHED_FIFO**, **SCHED_RR** and **SCHED_OTHER**
Works with tasks (i.e. processes on Unix, not threads)

POSIX APIs: Scheduling

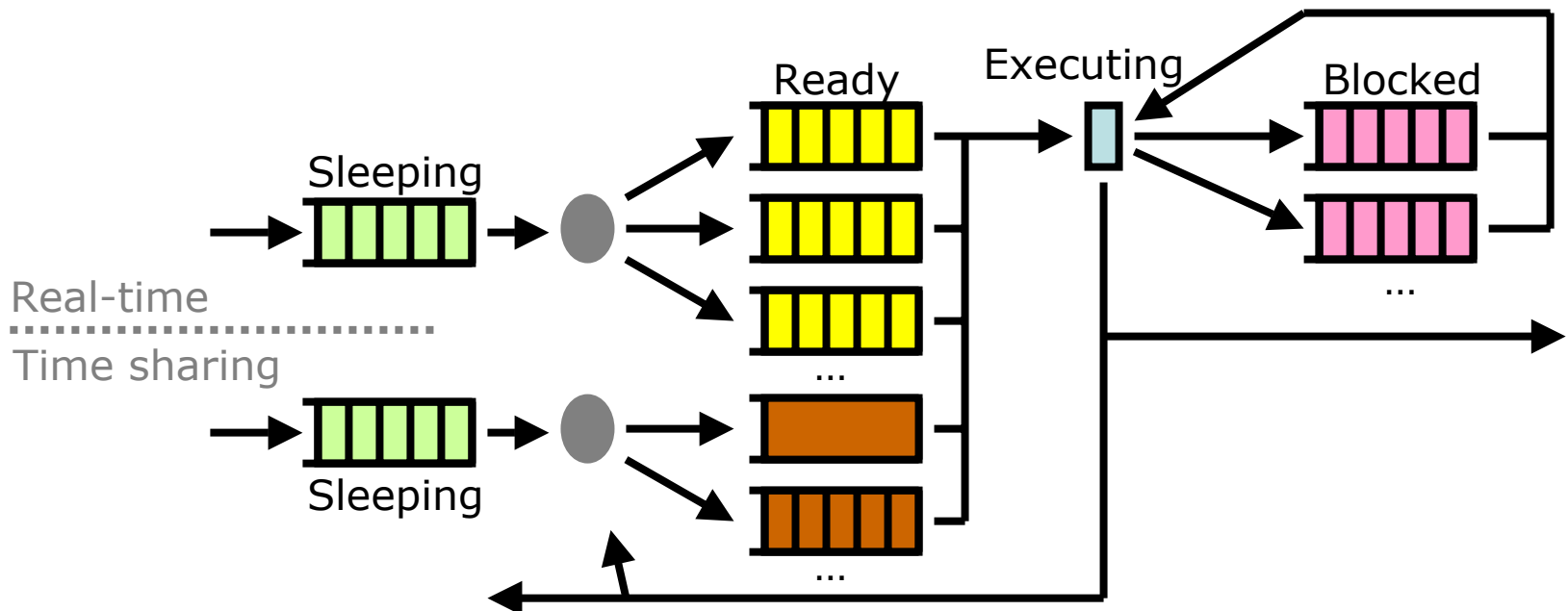
- POSIX 1003.1b provides three scheduling policies, selected using `sched_setscheduler()`:
 - `SCHED_FIFO`
 - Fixed priority, pre-emptive, FIFO scheduler
 - `SCHED_RR`
 - Fixed priority, pre-emptive, round robin scheduler
 - Use `sched_rr_get_interval(pid_t pid, struct timespec *t)` to find the scheduling 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
- Limited set of priorities:
 - Use `sched_get_priority_min()`, `sched_get_priority_max()` to determine the range
 - Guarantees at least 32 priority levels
 - Compare to Windows NT which supports only 16 priorities

POSIX APIs: 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



The 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);

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);
```

} Returns thread ID

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

Same scheduling policies and parameters as POSIX 1003.1b

The POSIX 1003.1c Real-Time Scheduling API

Scheduling parameters can be modified after a thread is created:

```
int pthread_getschedparam(pthread_t t, int *policy, struct sched_param *p);  
int pthread_setschedparam(pthread_t t, int policy, struct sched_param *p);
```

Threads can yield:

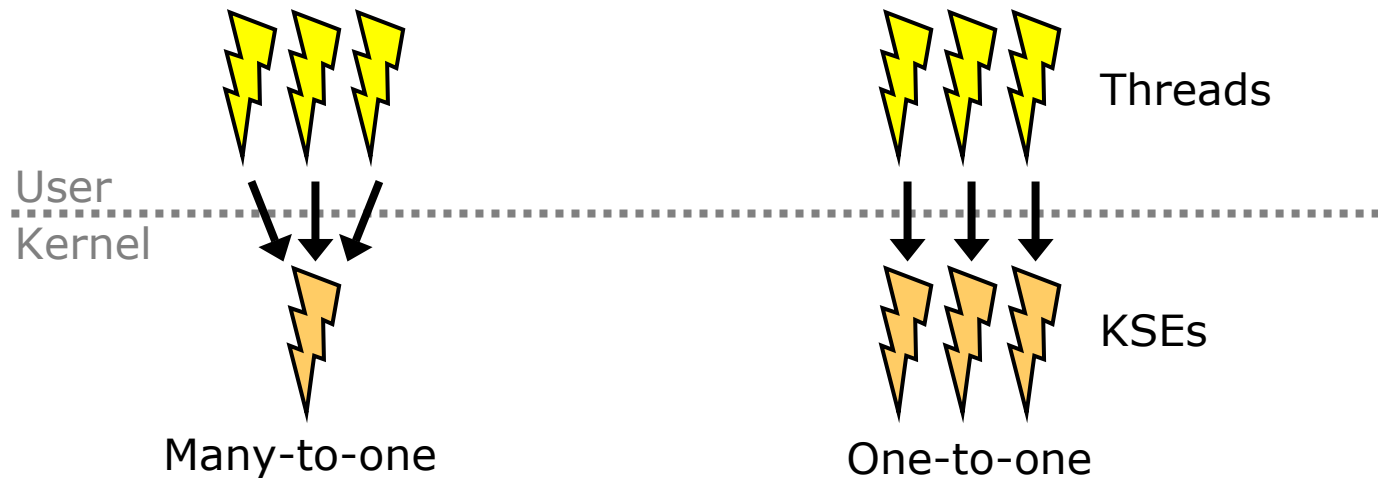
```
int sched_yield(void);
```

Threads exit in the usual manner:

```
int pthread_exit(void *retval);  
int pthread_join(pthread_t thread, void **retval);
```

Implementation of Threads

- Many operating systems make a distinction between **threads** which are exposed to the user, and **kernel scheduled entities (KSE)**
- The mapping may be one thread for each KSE or many threads for each KSE:



- If many threads map to each KSE, all threads in the group may share the same scheduling policy, or may block at once

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!

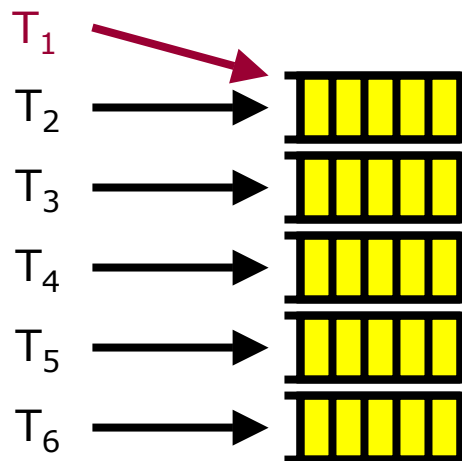
Using POSIX Scheduling: Rate Monotonic

- 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

Effects of Limited Priority Levels

- When building a custom system, can ensure that there are as many ready queues as priority levels
- If an operating system is present, through, the set of tasks may need priority levels than there are queues provided

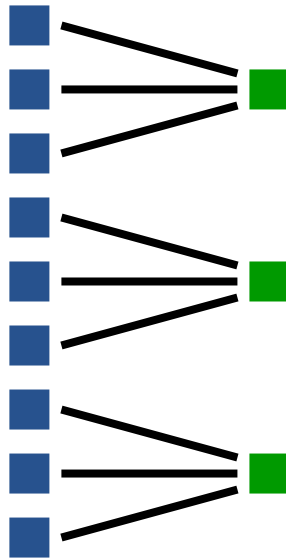


Implication: non-distinct priorities

- 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 $\sum_{T_k \in T_E(i)} e_k$
- The **schedulable utilization** of the system will be reduced

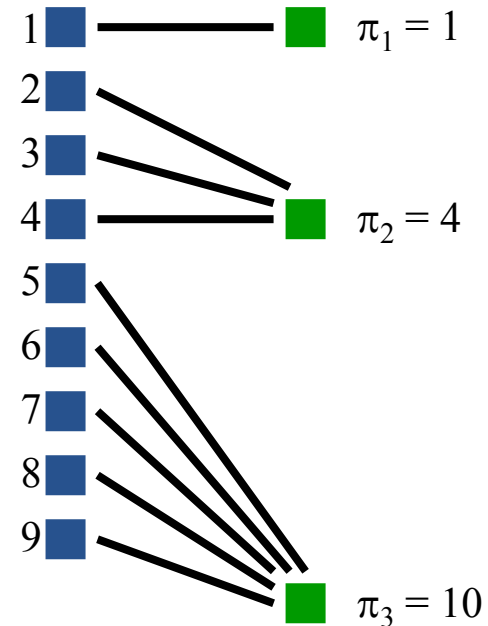
Effects of Limited Priority Levels

- 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

$Q = \lceil \Omega_n / \Omega_s \rceil$
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

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  
}
```
- [Will discuss timers more tomorrow]

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...

Using POSIX Scheduling: Aperiodic and Sporadic Tasks

- Difficult to implement aperiodic and sporadic tasks 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 something like `ITIMER_VIRTUAL` 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

By now, you should know:

- How real-time scheduler are typically implemented
- POSIX 1003.1b and 1003.1c APIs
- How to use POSIX APIs for fixed priority scheduling
- Limitations of the POSIX APIs