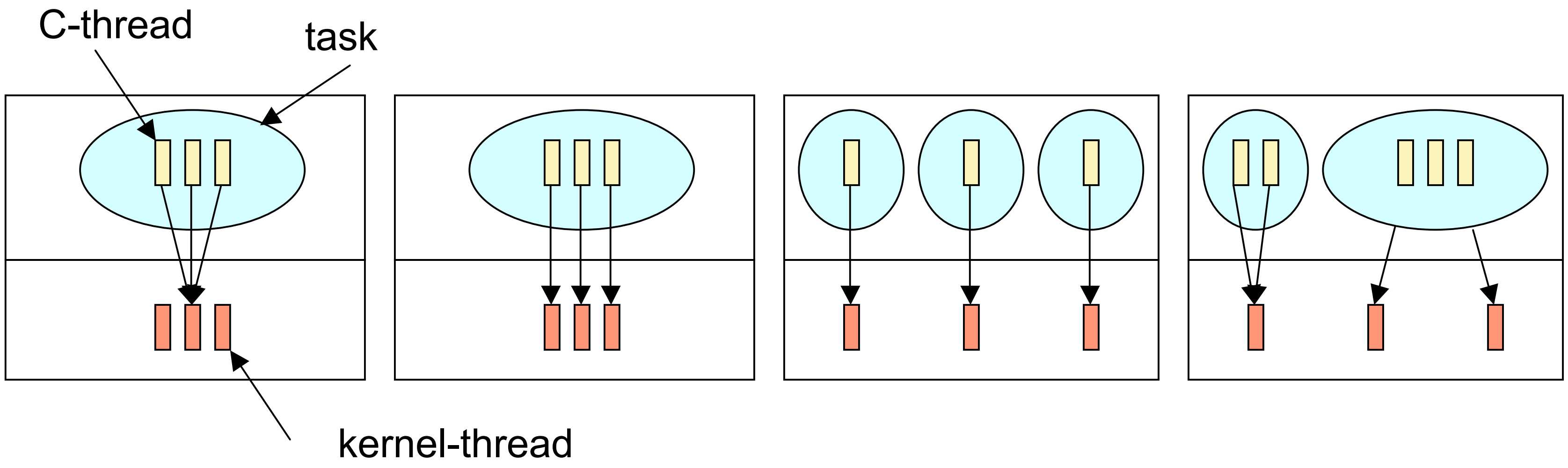
Unit 14: The Mach Operating System

14.2. Threads and Scheduling in Mach

Threads

- Active entities in Mach are threads
- Mach threads are managed by the kernel
- The C-Threads package provides a simpler interface to kernel threads
  - Several variants of mapping C-threads onto kernel threads



Mach C-Thread Functions

- Mach provides a set of low-level functions for manipulating threads of control.
- The C-thread run-time library provides an interface to the Mach facilities.
- The constructs provided in the C-thread functions are:
  - Forking and joining of threads
  - Protection of critical regions with mutual exclusion (mutex) variables
  - Condition variables for synchronization of threads
- C-thread functions should be used for multithreaded applications.
- Mach thread functions are designed to provide the low-level mechanisms.

C-thread Operations

Call	Description
Fork()	Creates a new thread running the same code as the parent thread
Exit()	Terminates the calling thread
Join()	Suspends the caller until a specified thread exits
Detach()	Announces that the thread will never be jointed (waited for)
Yield()	Gives up the CPU voluntarily
Self()	Returns the calling thread‘s identity to it

Using External Functions and Methods
(on the Mach-based NeXTSTEP OS)

- Many of the functions and methods provided by NeXTSTEP (Mach) weren't designed with multithreaded applications in mind.
  - they might not work correctly when called simultaneously.
- The following are thread-safe:
  - Distributed Objects (NeXTSTEP)
  - Mach functions (except for mach_error())
  - UNIX system calls (use cthread_errno() instead of errno)
  - NeXTSTEP exception handling (for example, NX_RAISE())
  - malloc() and its related functions,
  - thread safety can be disabled by calling malloc_singlethreaded()
- The Objective C runtime system is not thread-safe by default.
  - To make it thread-safe, use the function objc_setMultithreaded().

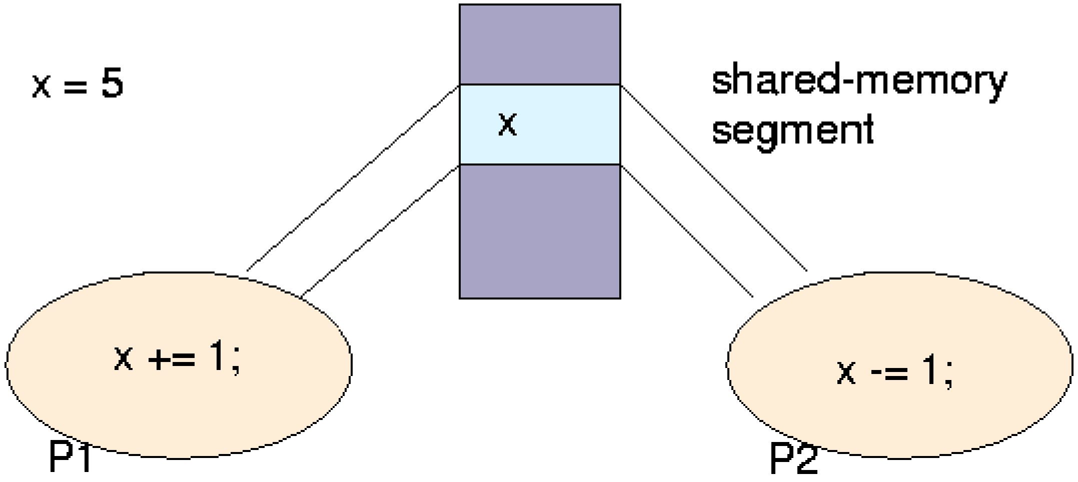
Threads and System Calls (contd.)

- The following are not thread-safe:
  - The Application Kit (messages to kit objects should be sent only from the main thread)
  - DPS (Display Postscript) client routines
  - The Window Server (drawing should be done only from the main thread)
  - Standard I/O functions, such as printf()
  - Most of the functions in the libc library
- usleep() should never be used in multithreaded programs.
  - alternatively use thread_switch():
  - thread_switch(THREAD_NULL, SWITCH_OPTION_WAIT, msecs);

Threads and Shared Data

- Global and static variables are shared among all threads:
  - If one thread modifies such a variable, all other threads will observe the new value.
  - A variable reachable from a pointer is shared.
  - This includes arguments passed by reference in cthread_fork().
- Declare all shared variables as volatile, or the optimizer might remove references to them!
- When pointers are shared, some care is required to avoid problems with dangling references.
  - lifetime of the object pointed to must allow other threads to dereference the pointer.
  - no bound on the relative execution speed of threads
  - share pointers to global or heap-allocated objects only.
- Libraries might make unprotected use of shared data.
  - use a mutex that's locked before every library call

The Synchronization Problem



x = 5

x

shared-memory segment

x += 1;

P1

x -= 1;

P2

printf("%d", x);

P1 read	P1 read	P2 read
P2 read	P2 read	P1 read
P2 write	P1 write	P1 write
P1 write	P2 write	P2 write
6	5	4

Synchronization of Variables

- Mutual exclusion and synchronization functions constrain interleaving of the execution threads.

```
typedef struct mutex {...} *mutex_t;
typedef struct condition {...} *condition_t;
```

- Mutually exclusive access to mutable data is necessary to prevent corruption of data.

```
mutex_lock(m);
count += 1;
mutex_unlock(m);
```

- Any other thread will block when it tries to lock the mutex in the meantime.
- If more than one thread tries to lock the mutex at the same time, only one succeeds.

Synchronization (contd.)

- Condition variables allow one thread to wait until another thread signals an event.
- Every condition variable should be protected by a mutex.

```
mutex_lock(mutex_t m);
. . .
while ( /* condition isn't true */ )
    condition_wait(condition_t c, mutex_t m);
. . .
mutex_unlock(mutex_t m);
```

Synchronization (contd.)

- condition_wait() temporarily unlocks the mutex
  - gives other threads a chance to get in and modify the shared data.
  - Eventually, one of them signals the condition before it unlocks the mutex:

```
mutex_lock(mutex_t m);
. . .       /* modify shared data */
condition_signal(condition_t c);
mutex_unlock(mutex_t m);
```

- Then, the original thread will regain its lock and can access the shared data again.

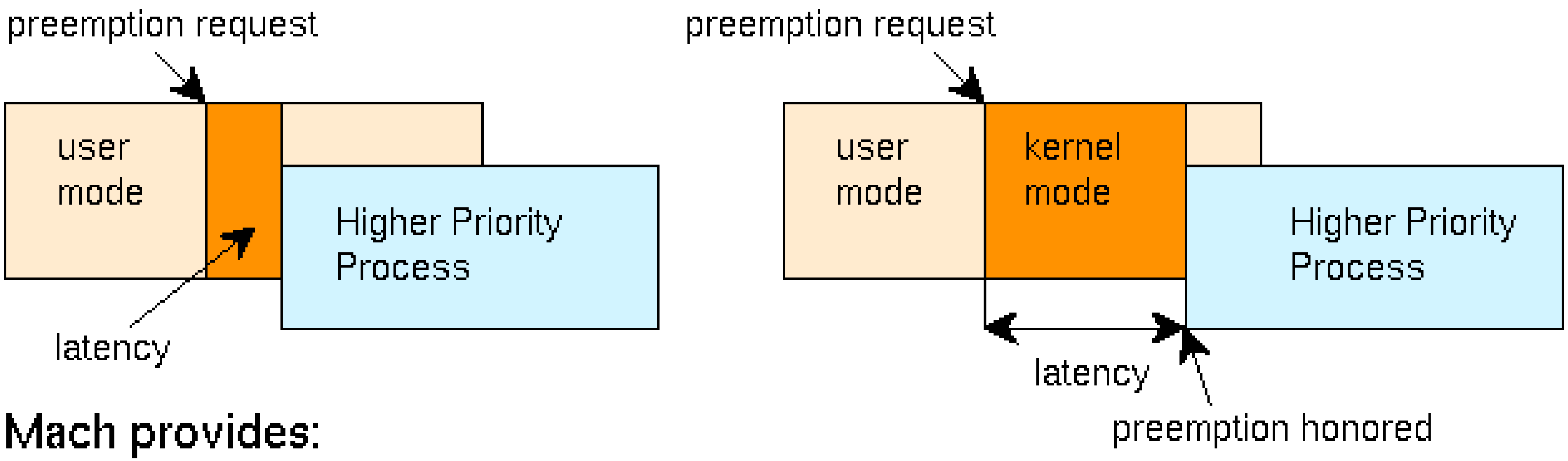
Synchronization pitfalls

- Attempting to lock a mutex that one already holds is a common error.
  - The offending thread will block waiting for itself.
- What kind of granularity to use in protecting shared data with mutexes?
  - one mutex protecting all shared memory
  - one mutex for every byte of shared memory.
- Finer granularity normally increases the possible parallelism.
- It also increases the overhead lost to locking and unlocking mutexes.

Mach Scheduling

- Each thread has a scheduling priority and policy.
  - Priority is a number between 0 and 31
  - indicates how likely the thread is to run.
- The higher the priority, the more likely a thread is to run.
- Timesharing policy is default
  - whenever the running thread blocks or after a certain amount of time -
  - the highest-priority runnable thread is executed.
- A thread's priority gets lower as it runs (it ages)
  - not even a high-priority thread can keep a low-priority thread from eventually running.

Preemptive vs. Non-preemptive Kernel



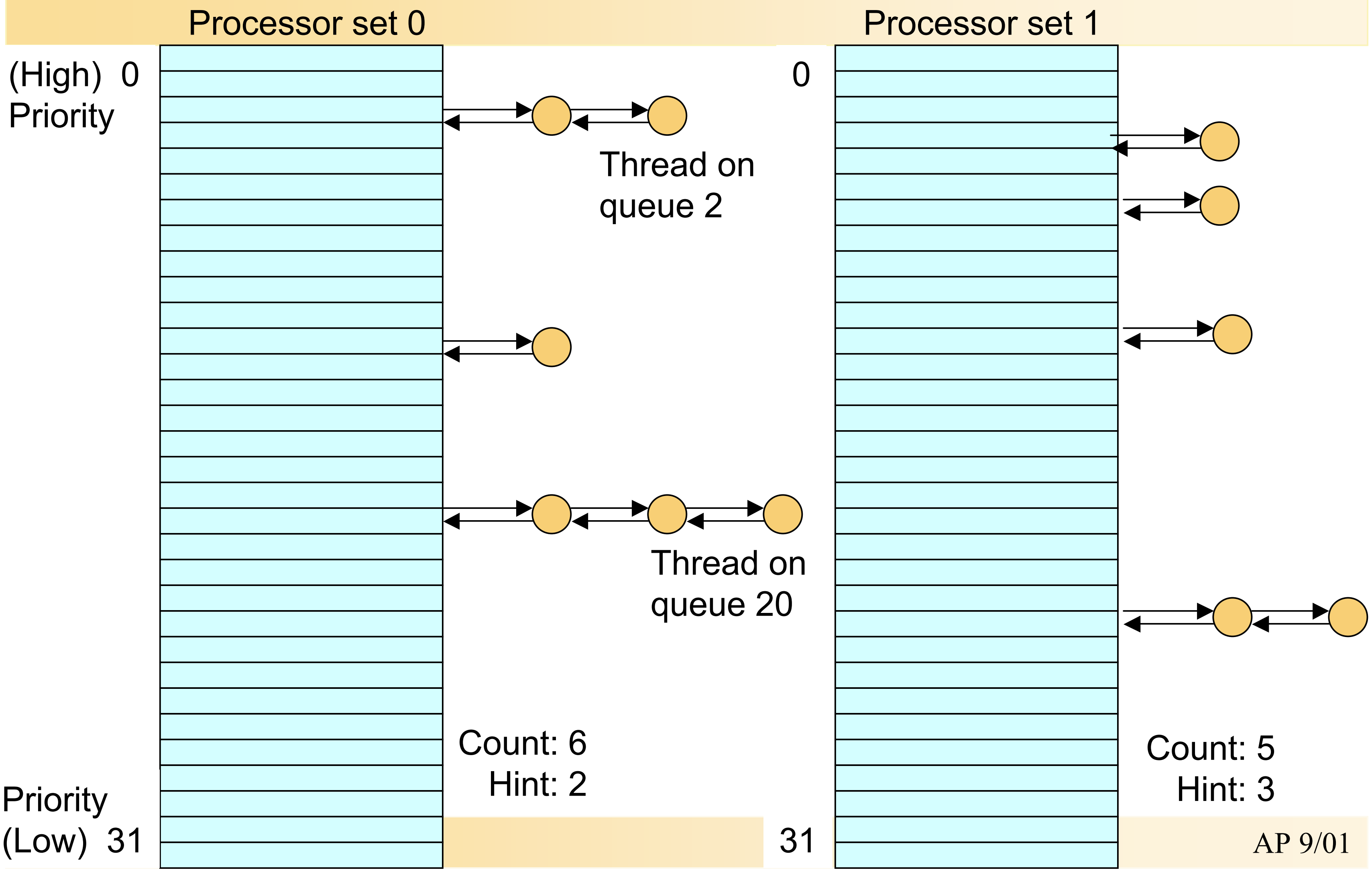
Mach provides:

- preemptive kernel: OSF/1 [RT], RT-Mach
- fixed-priority scheduling
- adjustable quantum
- memory locking - `vm_wire()`

Mach Scheduling (contd.)

- Heavily influenced by its goal of running on multiprocessors
  - CPUs in a multiprocessor can be assigned to processor sets
  - Each CPU belongs to exactly one processor set
  - Threads can also be assigned to processor sets and may be scheduled on any CPU belonging to a processor set
- Scheduling algorithm assigns threads to CPUs
  - Fairness and efficiency are optimization criteria
  - Priority-driven, decreasing priority preemptive scheduling with processor usage aging
  - Global run queues for each processor set
  - Local run queues for each processor (to allow attaching a thread to a particular processor during system calls)

Global Run Queues



Processor set 0

(High) 0
Priority

Thread on
queue 2

Thread on
queue 20

Count: 6
Hint: 2

Priority
(Low) 31

Processor set 1

0

31

Count: 5
Hint: 3

Priorities

- Each thread has three types of priorities associated with it:
  - base priority,
  - current priority,
  - maximum priority.
- Base priority is the one the thread starts with; it can be set using cthread_priority().
- Current priority is the one at which the thread is executing;
  - may be lower than the base priority due to aging or a call to thread_switch().
- Maximum priority is the highest priority at which the thread can execute.
  - a thread inherits its base priority from its parent task,
  - its maximum priority is set to a system-defined maximum.

Priorities (contd.)

- Priorities can be set at three levels:
  - the thread,
  - the task,
  - the processor set (on multiprocessors).
- At the thread level:
  - cthread_priority() , thread_priority() - set base priority, lower maximum priority.
- Raising or lowering just the maximum priority:
  - cthread_max_priority(), thread_max_priority().

Priorities (contd.)

- To raise a thread's maximum priority:
  - the privileged port of the thread's processor set must be obtained,
  - only the superuser can do this.
- At the task level:
  - task_priority() sets the task's base priority.
  - inherited by all threads that it forks;
  - optionally all existing threads in the task can get the new base priority.

Policies

- The NeXT Mach operating system has three scheduling policies:
  - Timesharing
  - Interactive
  - Fixed priority
- Every thread starts with the timesharing policy, no matter what policy the creator of the thread has.
- Policies other than timesharing can be set using thread_policy().
- The interactive policy is a variant of timesharing;
  - designed to be optimized for interactive applications.
  - a non-NEXTSTEP application should be set to interactive policy.
  - Currently, the interactive policy is exactly the same as timesharing).
  - performance might be enhanced by making interactive policy threads have higher priorities than the other threads in the task.

Fixed Priority Scheduling

- No descreasing priorities, no aging
- Fixed priority can be a dangerous policy if you aren't familiar with all of its consequences.
  - fixed-priority policy is disabled by default.
  - must be enabled using processor_set_policy_enable().
- Threads that have the fixed-priority policy have their current priority always equal to their base priority
  (unless their priority is depressed by thread_switch()).
- A thread with the fixed-priority policy runs until one of the following happens:
  - A higher-priority process becomes available to run.
  - A per-thread, user-specified amount of time (the quantum) passes.
  - The thread blocks, waiting for some event or system resource.

Fixed Priority Scheduling Problems

- Fixed-priority threads can prevent lower-priority threads from running.
- The opposite can happen, too;
  - a low-priority, fixed-priority thread can be kept from running by higher-priority threads.
- The first problem can be solved by a call to thread_switch()
  - to temporarily depress priority
  - hand off the processor to another thread.
- The fixed-priority policy is often used for real-time problems.