SYSC 5701

Operating System Methods for Real-Time Applications

Threads

Winter 2014

Recall Process Creation

- each process has code, data, and stack
  - other resources too? (I/O devices?)
- allocate a **thread of control**
  - managed by the kernel

Code, Data, Stack

- what if processes replicate behaviour?
  – execute the same code?
  – OK if memory manager will permit this(?)
- in a strict **process model:**

  processes do not share memory

  – each must have its own copy of **code** + **data** and **stack** space

Heavyweight Process

- a **heavyweight process** requires all of the above: code, data, stack
- heavyweight processes are strict:

  do not share resources
- Heavyweight processes: system must have memory manager hardware that can isolate and protect regions of memory based on s/w level id's
  – Cortex-A (yes, MMU) … Cortex-M (no, MPU)

Lightweight Processes

- can share code and data
- must have own stack and thread of control
- less overhead during lightweight kernel activity:
  – faster context switch and IPC ☺
  – application programmers must manage sharing of data ☹

    … (maybe specialized languages can help?)

Application Design Philosophy

- heavyweight processes: encapsulate **large-grained** parallel activities that are **loosely coupled**
  - i.e. interact infrequently, minimal data sharing
- lightweight processes encapsulate **finer-grained** parallel activities that are **tightly coupled**
  - i.e. interact frequently, lots of data sharing

Performance Goals

- heavyweight switching and IPC
  - → "more" expensive
  - – looser coupling = less heavyweight IPC and switching
- lightweight switching and IPC
  - → "less" expensive
  - – tighter coupling = more lightweight IPC and switching

Thread

- A **thread** is a lightweight process created in the context of a heavyweight process
- only the threads in the context of the same heavyweight process can access the code and data of that process

Thread Management

1. by kernel
2. outside of kernel
3. hybrid – developing trend!

1. Threads Managed by Kernel

kernel has two classes of processes:

- lightweight (thread) and heavyweight (process)
  - different services for each
- threads of a process are autonomous
- a thread may become blocked, but just that thread is blocked (not the entire process)

Kernel Managed Threads

- heavyweight process does not really execute as a single thread of control

  → a container for managing threads
- the process has a set of threads
  – the active elements of the process
- kernel manages both process and thread scheduling

2. Threads Outside of Kernel

- process has single thread of control
  - managed by kernel
- single control thread is shared among threads
  - managed by thread manager
    - unknown to kernel!
- this sort of thread called **user-thread** **(fiber?)**
- thread manager resides outside of kernel (appl$^{\underline{n}}$ code!)
- often a run-time library supplied by language/environment vendors

User Threads

- if strict process model – each process must have its own copy of thread manager (no code sharing!)
- if a thread makes an IPC call via the kernel and becomes blocked

  → kernel blocks the process ! ☹

  (kernel has no knowledge of threads!) ☹

3. Hybrid Threads

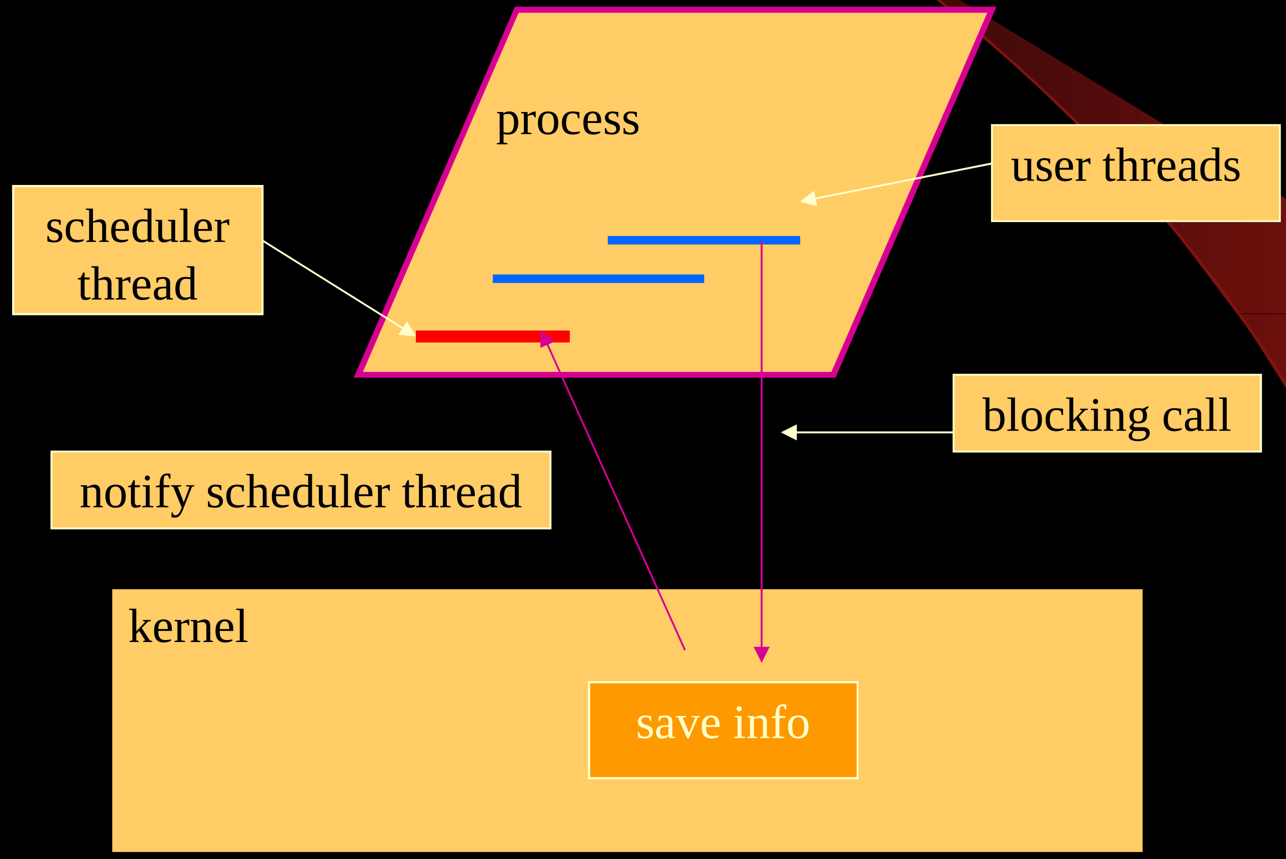
- user thread **concept** known to kernel, but user-thread manager is <u>outside</u> of kernel
- thread manager and kernel cooperate – **"scheduler" user thread** for the process is known to the kernel
  - special interactions supported between scheduler thread and kernel

Hybrid con't

when a user-thread invokes a kernel service and is blocked:

1. kernel "pseudo" blocks user-thread – records relevant blocking criteria, but instead of blocking process ...
2. kernel returns control to relevant **scheduler thread**
3. scheduler thread "blocks" the user-thread (**outside of kernel!**) and schedules a different user-thread
   - net result: thread is blocked, **process not blocked**
4. when criteria met to unblock original user-thread
   - kernel informs relevant scheduler thread
   - scheduler thread unblocks the thread and makes thread scheduling decisions

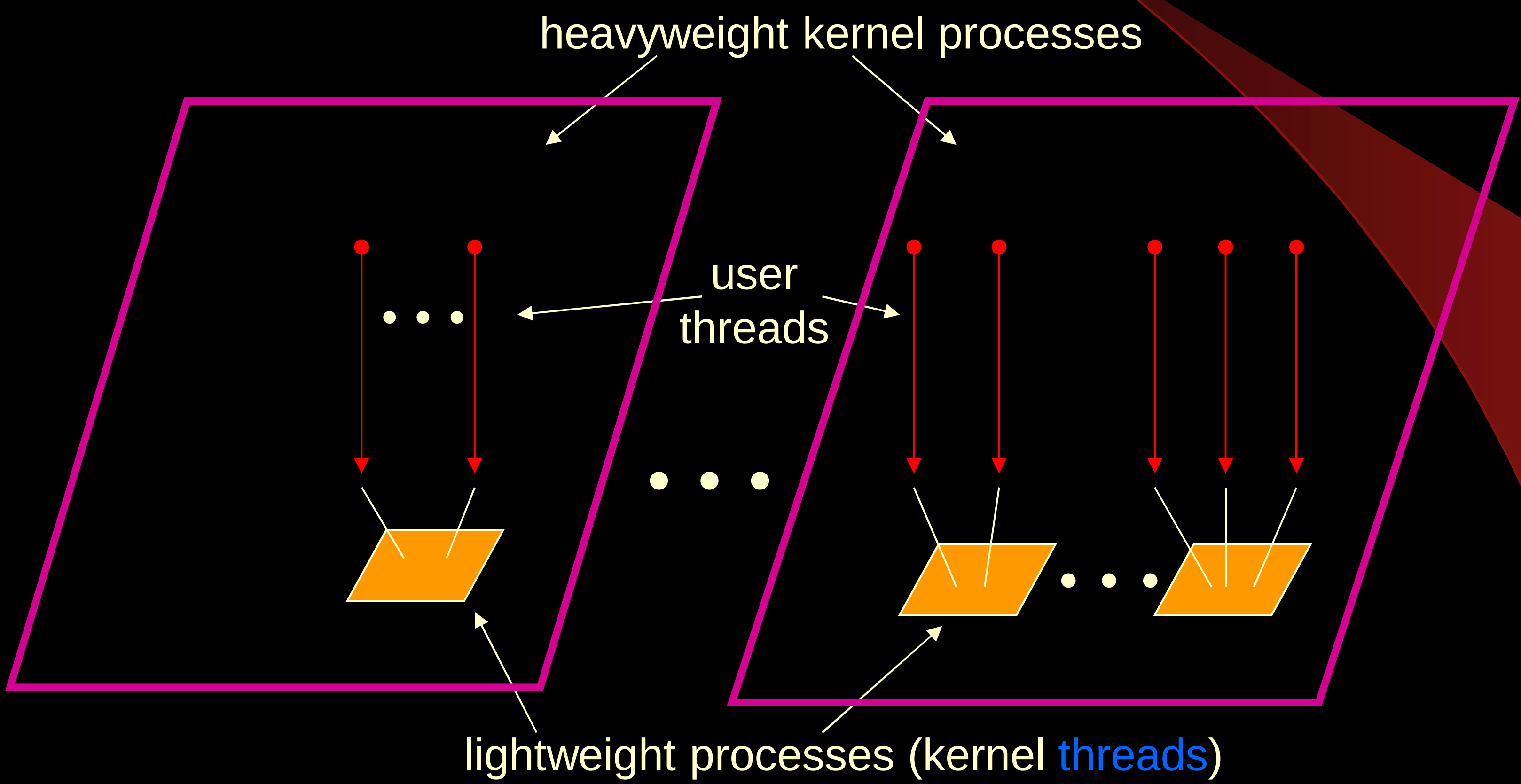
Hybrid Thread Blocking



User-Thread Scheduling Issues

- time-slice ☹
  - suitability to real-time app's ?
- preempt vs. non-preempt
  - voluntary relinquish?
- all the same-old issues:
  - priority? timed services? etc.

Extended Process Model with Threads



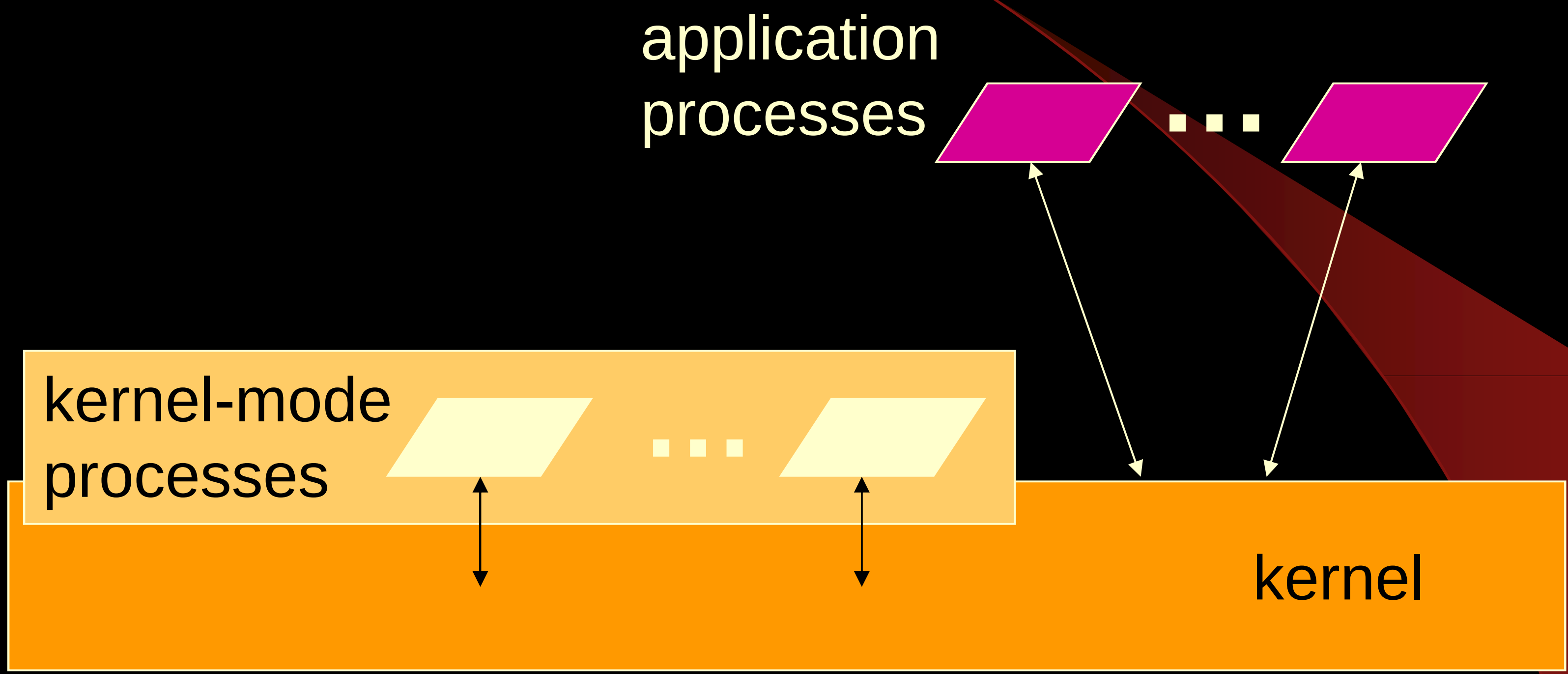
Kernel-Mode Threads/Processes

- common in large, general-purpose o/s
  - MMU hardware!
- **not as common in real-time applications**
- modern o/s' often manage I/O subsystems
  - e.g. disk I/O subsystem
  - require "supervisor" permission to access restricted I/O devices
  - must execute in kernel's supervisor context

O/S Layer Above Kernel

- may include processes that are not visible outside of the o/s but exist inside the o/s
- these processes are created (and run) in the kernel's context to permit access to restricted h/w
- called **kernel-mode processes**

Kernel-Mode Process Layer



Similar (Kernel) Thread Trend

- when process invokes kernel service – kernel blocks process and creates a kernel-mode thread associated with the request
- new thread has unique stack (in kernel's space) + shares access to kernel code and data
- must have enough stack space to permit a kernel-mode thread for each process ☹
- kernel-mode thread executes with supervisor privileges on behalf of the requesting process

Kernel-Mode Threads

- kernel manages kernel-mode threads
- kernel threads can be preempted
- more concurrency
- still need to protect access to kernel's shared data structures!

- has overheads! ☹
- hasn't hit real-time kernels in a big way (yet)

  but h/w is driving in this direction!

Threads Example: POSIX Threads (pthreads)

- Part of Portable Operating System Standard (POSIX)
  - http://en.wikipedia.org/wiki/POSIX
- IEEE Std 1003.1-2013
- http://www.opengroup.org/onlinepubs/9699919799/nframe.html
- Maintained by The Open Group http://www.opengroup.org
- Based on UNIX thread model … widely supported
  - E.g.: QNX, VxWorks (PSE52 … later)
  - Many more (Windows add-ins too!)
- API spec for: C (primary) [Ada, Fortran (optional)]

Just scratching the surface in these slides!!!!

Sync & Comm (Normal)

- semaphores for synchronization
- shared memory
- messaging (message queues)
- mutex (lock, unlock)
  - access control (e.g. priority inheritance)
- condition variables (Hoare-style monitors)
- timed services

Thread Management

- “main” thread – created when containing process created
- process terminates when “main” thread terminates

```
int pthread_create (
  pthread_t *thread,                  // id
  pthread_attr_t *attr,             // attributes
  void *(*start_routine)(void *), // funcn
  void *arg                           // args
);
```


Carleton UNIVERSITY

Joining a Thread

- sometimes want thread to return some application-specific value at termination
- allow a thread to "join" a 2nd thread to receive the 2nd thread's termination data
- need special commⁿ mechanism → make sure thread's return value is not lost if 2nd thread terminates before joiner

Carleton UNIVERSITY

Join

`int pthread_join( 2nd_thread, **return_value)`

- "attach" thread to 2nd_thread
- blocks (waits) until 2nd_thread terminates
- returns with "return value" from 2nd_thread
- if 2nd_thread terminates first, 2nd_thread is put in "limbo" state, but data persists to ensure the return_value is not lost

Thread Termination

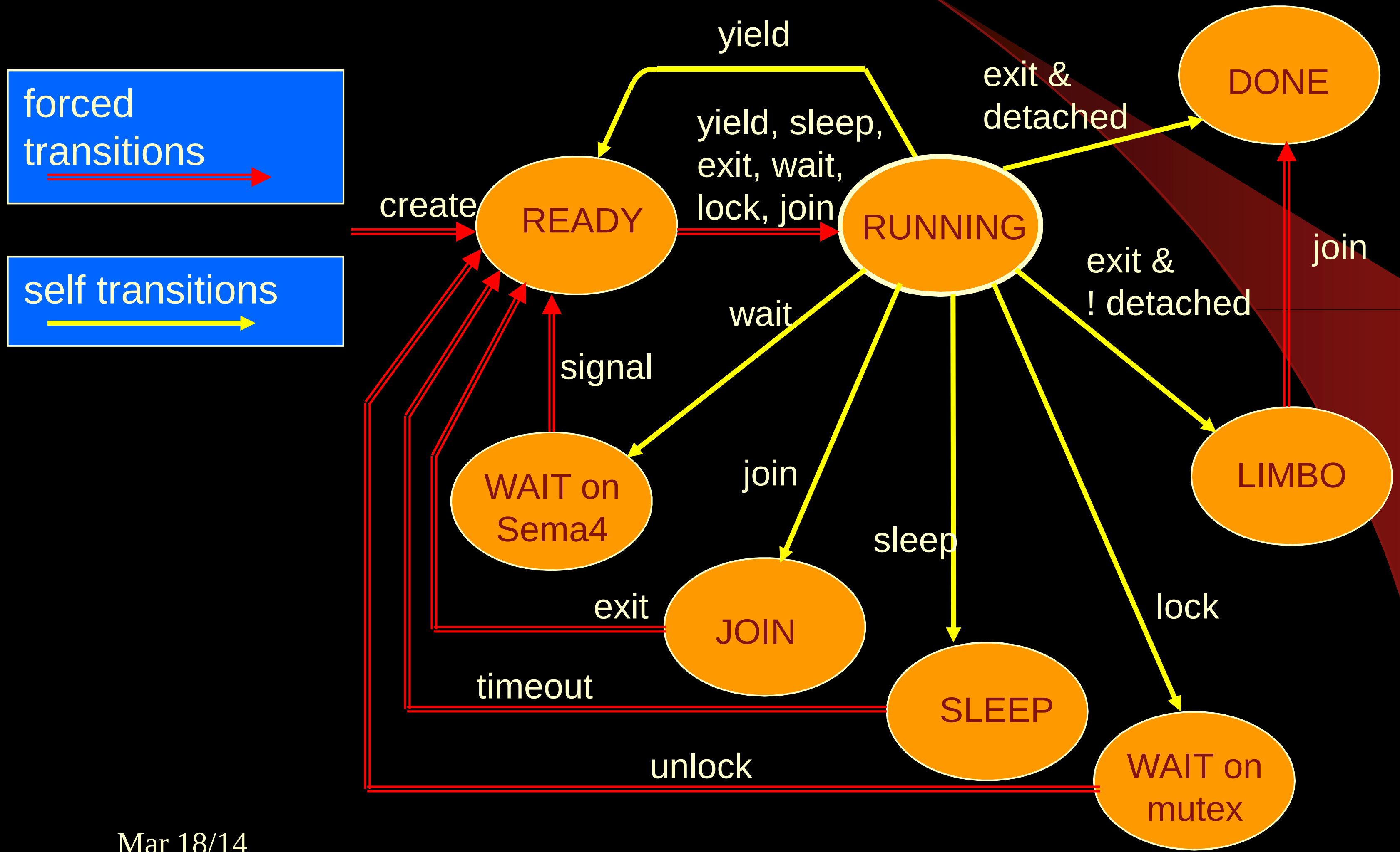
- implicit: return from start_routine function
  - returns function's return_value to joiner
- explicit: `void pthread_exit( *return_value )`

- can also explicitly terminate another thread:
  `int pthread_kill( other_thread, sig);`

Detached Thread

- when thread not required to supply a return_value → declare thread as detached
- when detached thread terminates, all allocated resources are reclaimed

(Partial) pthread State Machine



Lock/Unlock Mutex

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

- Similar to binary semaphore
- Can set a priority ceiling
- Can set policy (priority inheritance, priority ceiling emulation, none)

Monitors

- use condition variable as the blocking "queue"

No aliasing

```
int pthread_cond_init(
     pthread_cond_t *restrict cond,
     const pthread_condattr_t *restrict attr);
```

Monitor: Wait on Condition Variable

```
int pthread_cond_wait(
        pthread_cond_t *cond,
        pthread_mutex_t *mutex);
```

- caller waits on condition variable, and mutex is unlocked (single operation!)
- convention: when return from wait, caller “owns” the mutex
  - i.e. releaser must leave monitor!

Monitor: Signal Condition Variable

```
int pthread_cond_signal( thread_cond_t *cond );
```

- Unblocks a thread from the condition variable
  - No blocked thread? → no effect
- Scheduling policy decides which thread
  - E.G. priority-driven
- Released thread "owns" mutex
  - Recall **`pthread_cond_wait`**

Timed Wait on Sema4

```
int sem_timedwait(  sem_t *restrict sem,
  const struct timespec *restrict abstime);
```

- Absolute time
- Return: 0 = success (locked sema4 within specified time)
    - – 1 = error, including eTimedOut
- Similar call for wait on condition variable

Signals

- Allow asynchrounous events to be communicated and then processed
- Record individual signal in sigevent data structure
  - Some are predefined system event types
  - Application can define specific event types, too
- All event types for system = union of all sigevents

N.B.: Just touching surface … very complex!!

Raising Signals

- Can be caused by runtime events
  - Asynchronous I/O, Timeouts, Faults
- May be sent to:
  - Process (thread container)
    - Process by any “willing” thread
  - Specific thread
- May be queued
- Is “pending” until received

Receiving Signals

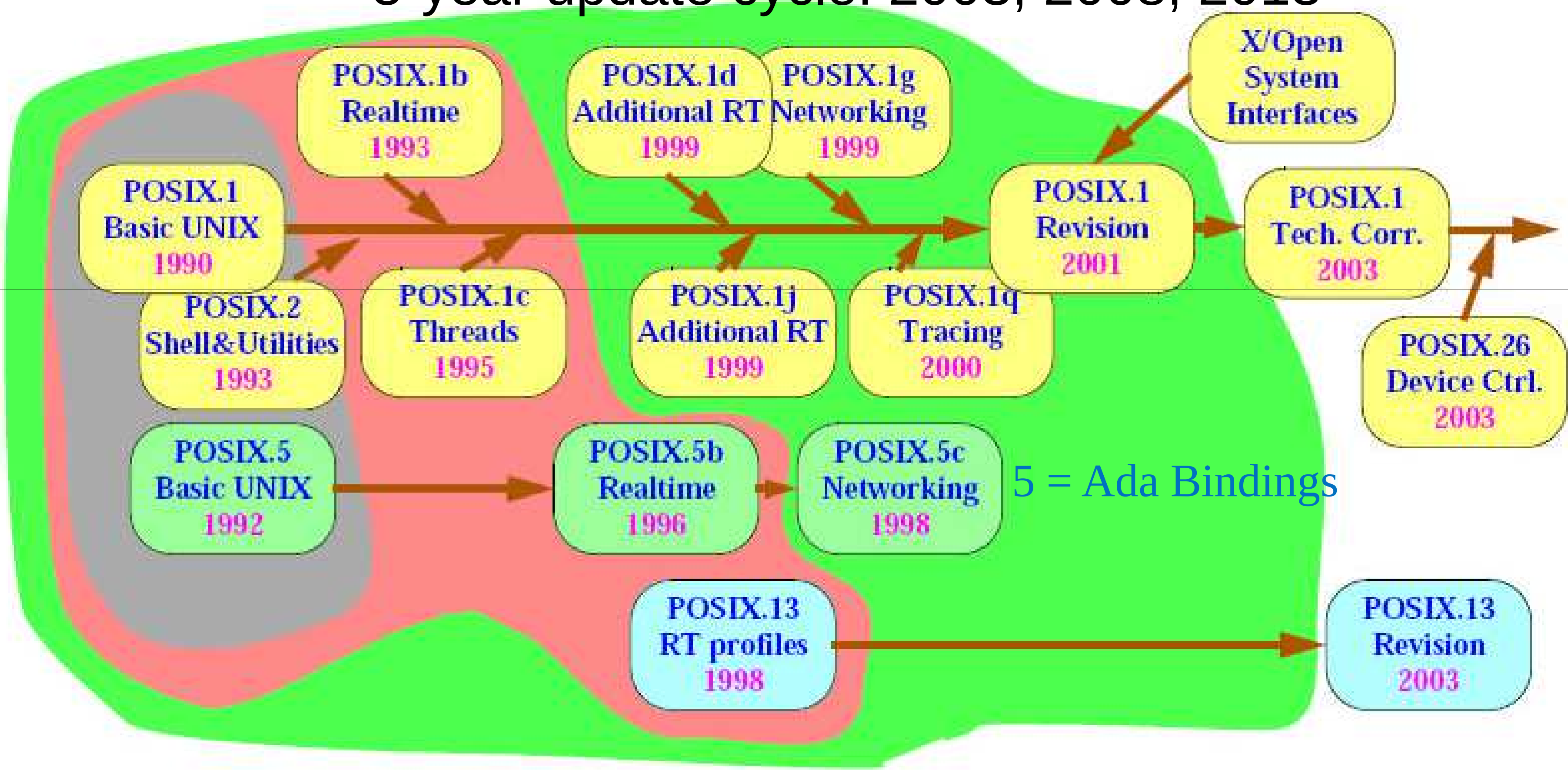
```
int sigwait( const sigset_t *restrict set,
     int *restrict sig);
```

- Calling thread waits for any signal in set
  - “waits” = blocked
  - “willing” to process any signal in set
- Signal number stored (returned) in sig
- After call, caller then takes appropriate action to process signal
- Provides asynchronous pre-emption

POSIX.1, RT & RT Profiles

https://www.opengroup.org/platform/single_unix_specification/uploads/40/5991/POSIX-briefing-2006-2.PDF

5-year update cycle: 2003, 2008, 2013



Minimal profile (PSE51)

http://www-users.cs.york.ac.uk/~burns/papers/c-posix.pdf

- Single Process
- Threads
- Memory Management
- Semaphores
- Mutexes with Priority Inheritance
- Condition Variables
- Signals
- Clocks and Timers
- I/O devices
- Fixed priority sporadic server
- NOT in profile:
  - file service (beyond I/O)
  - message queues
  - networking

POSIX RT Profiles

https://www.opengroup.org/platform/single_unix_specification/uploads/40/5991/POSIX-briefing-2006-2.PDF