Threads

Thread: a lightweight process which has its own stack, registers, signal mask, and scheduling priority. It shares global memory, file descriptors, etc.

- Threads are similar to child processes, but threads require less of a context switch and kernel overhead to manage it.

Threads

Thread: a lightweight process which has its own stack, registers, signal mask, and scheduling priority. It shares global memory, file descriptors, etc.

- Threads are similar to child processes, but threads require less of a context switch and kernel overhead to manage it.
- All a process's threads share the same data and code segments

Threads

Thread: a lightweight process which has its own stack, registers, signal mask, and scheduling priority. It shares global memory, file descriptors, etc.

- Threads are similar to child processes, but threads require less of a context switch and kernel overhead to manage it.
- All a process's threads share the same data and code segments
- A child process may use `exec` and have a completely different code and data (or merely copies thereof)

Threads

Thread: a lightweight process which has its own stack, registers, signal mask, and scheduling priority. It shares global memory, file descriptors, etc.

- Threads are similar to child processes, but threads require less of a context switch and kernel overhead to manage it.
- All a process's threads share the same data and code segments
- A child process may use `exec` and have a completely different code and data (or merely copies thereof)
- A thread which calls `exit()` will terminate the entire process.

Threads

Thread: a lightweight process which has its own stack, registers, signal mask, and scheduling priority. It shares global memory, file descriptors, etc.

- Threads are similar to child processes, but threads require less of a context switch and kernel overhead to manage it.
- All a process's threads share the same data and code segments
- A child process may use `exec` and have a completely different code and data (or merely copies thereof)
- A thread which calls `exit()` will terminate the entire process.
- A thread usually must synchronize access to global variables, often using *semaphores* or *mutexes*

Threads

Thread: a lightweight process which has its own stack, registers, signal mask, and scheduling priority. It shares global memory, file descriptors, etc.

- Threads are similar to child processes, but threads require less of a context switch and kernel overhead to manage it.
- All a process's threads share the same data and code segments
- A child process may use `exec` and have a completely different code and data (or merely copies thereof)
- A thread which calls `exit()` will terminate the entire process.
- A thread usually must synchronize access to global variables, often using *semaphores* or *mutexes*
- Threads are often used to facilitate use of multiple processors (cores)

Threads

Thread: a lightweight process which has its own stack, registers, signal mask, and scheduling priority. It shares global memory, file descriptors, etc.

- Threads are similar to child processes, but threads require less of a context switch and kernel overhead to manage it.
- All a process's threads share the same data and code segments
- A child process may use `exec` and have a completely different code and data (or merely copies thereof)
- A thread which calls `exit()` will terminate the entire process.
- A thread usually must synchronize access to global variables, often using *semaphores* or *mutexes*
- Threads are often used to facilitate use of multiple processors (cores)
- Threads may be associated with objects by the programmer (thereby facilitating object-oriented programming)

Models of Thread Use

Boss/Worker a single thread handles inputs, assigning tasks to worker threads

Example 1

```
while(1) {
..get input..
switch(request) {
        case 'a': pthread_create(...); break;
        case 'b': pthread_create(...); break;
        ...
        }
}
```

Models of Thread Use

Boss/Worker a single thread handles inputs, assigning tasks to worker threads

Example 4

```
while(1) {
..get input..
switch(request) {
        case 'a': pthread_create(...); break;
        case 'b': pthread_create(...); break;
        ...
        }
}
```

Peer Model each thread handles its own input

Example 5

```
server accepts client
pthread_create(...) // thread exclusively handles client
```

Models of Thread Use

Boss/Worker a single thread handles inputs, assigning tasks to worker threads

Example 7

```
while(1) {
..get input..
switch(request) {
        case 'a': pthread_create(...); break;
        case 'b': pthread_create(...); break;
        ...
        }
}
```

Peer Model each thread handles its own input

Example 8

server accepts client

pthread_create(...) // thread exclusively handles client

Pipeline Model threads handle parts of tasks and pass results along

Example 9

Dynamics controller for robot

Thread 1 drives joint 1 to desired value. Computes joint velocity, acceleration.

Thread 2 drives joint 2 to desired value, accounting for motion of joint 1.

Lightweight Process

- A **LWP** is what is actually scheduled to be run by the kernel.

Lightweight Process

- A **LWP** is what is actually scheduled to be run by the kernel.
- It can be thought of as a virtual CPU; thread libraries communicate with LWPs to schedule threads.

Lightweight Process

- A **LWP** is what is actually scheduled to be run by the kernel.
- It can be thought of as a virtual CPU; thread libraries communicate with LWPs to schedule threads.
- LWPs are sometimes called “kernel threads”

Lightweight Process

- A **LWP** is what is actually scheduled to be run by the kernel.
- It can be thought of as a virtual CPU; thread libraries communicate with LWPs to schedule threads.
- LWPs are sometimes called “kernel threads”
- Threads may be executed by any available LWP, but may also be bound to a particular one.

Lightweight Process

- A **LWP** is what is actually scheduled to be run by the kernel.
- It can be thought of as a virtual CPU; thread libraries communicate with LWPs to schedule threads.
- LWPs are sometimes called “kernel threads”
- Threads may be executed by any available LWP, but may also be bound to a particular one.
- In such a case, the kernel’s scheduling of a LWP to run implies that the bound thread will be run; the kernel is then doing the scheduling of the thread rather than the pthread library.

Lightweight Process

- A **LWP** is what is actually scheduled to be run by the kernel.
- It can be thought of as a virtual CPU; thread libraries communicate with LWPs to schedule threads.
- LWPs are sometimes called “kernel threads”
- Threads may be executed by any available LWP, but may also be bound to a particular one.
- In such a case, the kernel’s scheduling of a LWP to run implies that the bound thread will be run; the kernel is then doing the scheduling of the thread rather than the pthread library.
- Kernel threads need not be associated with a process, but do not execute in user mode. They require “context switching”.

Lightweight Process

- A **LWP** is what is actually scheduled to be run by the kernel.
- It can be thought of as a virtual CPU; thread libraries communicate with LWPs to schedule threads.
- LWPs are sometimes called "kernel threads"
- Threads may be executed by any available LWP, but may also be bound to a particular one.
- In such a case, the kernel's scheduling of a LWP to run implies that the bound thread will be run; the kernel is then doing the scheduling of the thread rather than the pthread library.
- Kernel threads need not be associated with a process, but do not execute in user mode. They require "context switching".
- User-level thread scheduling can avoid many system calls and switches into kernel mode, thereby improving performance.

Lightweight Process

- A **LWP** is what is actually scheduled to be run by the kernel.
- It can be thought of as a virtual CPU; thread libraries communicate with LWPs to schedule threads.
- LWPs are sometimes called “kernel threads”
- Threads may be executed by any available LWP, but may also be bound to a particular one.
- In such a case, the kernel’s scheduling of a LWP to run implies that the bound thread will be run; the kernel is then doing the scheduling of the thread rather than the pthread library.
- Kernel threads need not be associated with a process, but do not execute in user mode. They require “context switching”.
- User-level thread scheduling can avoid many system calls and switches into kernel mode, thereby improving performance.
- LWPs have their own signal stacks, alarms, timers, and scheduling priority.

Thread Creation

```
#include <pthread.h>
void pthread_create(pthread_t *thread,const pthread_attr_t *attr,
  void *(*start_routine) (void *),void *arg);
```

- Use pthread_create() to create new threads. A thread will use a light weight process when running.

Thread Creation

```
#include <pthread.h>
void pthread_create(pthread_t *thread,const pthread_attr_t *attr,
  void *(*start_routine) (void *),void *arg);
```

- Use pthread_create() to create new threads. A thread will use a light weight process when running.
- Pass it a pointer to a pthread_t instance. This can be used with other pthread functions.

  ie. Declare a pthread_t and pass an address to it along:

  ```
  pthread_t threadname;
  pthread_create(&threadname,NULL,ThreadNameFunc,NULL);
  ```

Thread Creation

```
#include <pthread.h>
void pthread_create(pthread_t *thread,const pthread_attr_t *attr,
  void *(*start_routine) (void *),void *arg);
```

- Use pthread_create() to create new threads. A thread will use a light weight process when running.
- Pass it a pointer to a pthread_t instance. This can be used with other pthread functions.

  ie. Declare a pthread_t and pass an address to it along:

  ```
  pthread_t threadname;
  pthread_create(&threadname,NULL,ThreadNameFunc,NULL);
  ```
- One may specify various attributes to apply to the thread (scheduling, priority, etc). You may pass a null pointer to use default settings.

Thread Creation

```
#include <pthread.h>
void pthread_create(pthread_t *thread,const pthread_attr_t *attr,
  void *(*start_routine) (void *),void *arg);
```

- Use pthread_create() to create new threads. A thread will use a light weight process when running.
- Pass it a pointer to a pthread_t instance. This can be used with other pthread functions.

  ie. Declare a pthread_t and pass an address to it along:

  ```
  pthread_t threadname;
  pthread_create(&threadname,NULL,ThreadNameFunc,NULL);
  ```
- One may specify various attributes to apply to the thread (scheduling, priority, etc). You may pass a null pointer to use default settings.
- The new thread will start executing by calling the start_routine(void *arg) function.

Thread Creation

```
#include <pthread.h>
void pthread_create(pthread_t *thread,const pthread_attr_t *attr,
  void *(*start_routine) (void *),void *arg);
```

- Use pthread_create() to create new threads. A thread will use a light weight process when running.
- Pass it a pointer to a pthread_t instance. This can be used with other pthread functions.

  ie. Declare a pthread_t and pass an address to it along:

  ```
  pthread_t threadname;
  pthread_create(&threadname,NULL,ThreadNameFunc,NULL);
  ```
- One may specify various attributes to apply to the thread (scheduling, priority, etc). You may pass a null pointer to use default settings.
- The new thread will start executing by calling the start_routine(void *arg) function.
- One way to terminate the thread is to allow the start_routine() function to return.

Thread Termination

```
#include <pthread.h>
void pthread_exit(void *retval);
```

- Terminates the calling thread

Thread Termination

#include <pthread.h>
void pthread_exit(void *retval);

- Terminates the calling thread
- The pointer to retval is given to threads which have used pthread_join() (unless the thread has been detached)

Thread Termination

#include <pthread.h>
void pthread_exit(void *retval);

- Terminates the calling thread
- The pointer to retval is given to threads which have used pthread_join() (unless the thread has been detached)
- This call does not release any application-wide resources such as mutexes, file descriptors, etc.

Thread Termination

#include <pthread.h>
void pthread_exit(void *retval);

- Terminates the calling thread
- The pointer to retval is given to threads which have used pthread_join() (unless the thread has been detached)
- This call does not release any application-wide resources such as mutexes, file descriptors, etc.
- thrd2.c uses a function pthread_yield; this function causes the calling thread to

Thread Termination

#include <pthread.h>
void pthread_exit(void *retval);

- Terminates the calling thread
- The pointer to retval is given to threads which have used pthread_join() (unless the thread has been detached)
- This call does not release any application-wide resources such as mutexes, file descriptors, etc.
- thrd2.c uses a function pthread_yield; this function causes the calling thread to
  1. relinquish the CPU

Thread Termination

#include <pthread.h>
void pthread_exit(void *retval);

- Terminates the calling thread
- The pointer to retval is given to threads which have used pthread_join() (unless the thread has been detached)
- This call does not release any application-wide resources such as mutexes, file descriptors, etc.
- thrd2.c uses a function pthread_yield; this function causes the calling thread to
  1. relinquish the CPU
  2. place thread at end of run queue (given its priority)

Thread Termination

#include <pthread.h>
void pthread_exit(void *retval);

- Terminates the calling thread
- The pointer to retval is given to threads which have used pthread_join() (unless the thread has been detached)
- This call does not release any application-wide resources such as mutexes, file descriptors, etc.
- thrd2.c uses a function pthread_yield; this function causes the calling thread to
  1. relinquish the CPU
  2. place thread at end of run queue (given its priority)
  3. Another thread is scheduled to run

Thread Termination

#include <pthread.h>
void pthread_exit(void *retval);

- Terminates the calling thread
- The pointer to retval is given to threads which have used pthread_join() (unless the thread has been detached)
- This call does not release any application-wide resources such as mutexes, file descriptors, etc.
- thrd2.c uses a function pthread_yield; this function causes the calling thread to
  1. relinquish the CPU
  2. place thread at end of run queue (given its priority)
  3. Another thread is scheduled to run
  4. This function is *non-standard*

Thread Termination

#include <pthread.h>
void pthread_exit(void *retval);

- Terminates the calling thread
- The pointer to retval is given to threads which have used pthread_join() (unless the thread has been detached)
- This call does not release any application-wide resources such as mutexes, file descriptors, etc.
- thrd2.c uses a function pthread_yield; this function causes the calling thread to
  1. relinquish the CPU
  2. place thread at end of run queue (given its priority)
  3. Another thread is scheduled to run
  4. This function is *non-standard*

(See thrd1.c and thrd2.c)

Thread Termination, con't.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

- Suspends the calling thread until the specified thread has terminated

Thread Termination, con't.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

- Suspends the calling thread until the specified thread has terminated
- A pointer to the retval, if not null, becomes available

Thread Termination, con't.

#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);

- Suspends the calling thread until the specified thread has terminated
- A pointer to the retval, if not null, becomes available
- This function is a lot like waitpid().

Thread Termination, con’t.

#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);

- Suspends the calling thread until the specified thread has terminated
- A pointer to the retval, if not null, becomes available
- This function is a lot like waitpid().
- If you don’t pthread_join() non-detached threads, you end up with **zombie threads**

Thread Termination, con't.

#include <pthread.h>
int pthread_detach(pthread_t thread);

- All storage and resources owned by the thread will be released immediately when the thread terminates.

Thread Termination, con't.

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

- All storage and resources owned by the thread will be released immediately when the thread terminates.
- Use this when your thread won't be "join'd"

Thread Termination, con’t.

#include <pthread.h>
int pthread_detach(pthread_t thread);

- All storage and resources owned by the thread will be released immediately when the thread terminates.
- Use this when your thread won’t be “join’d”
- Attempting to detach an already detached thread is unspecified behavior

Thread Termination, con't.

#include <pthread.h>
int pthread_detach(pthread_t thread);

- All storage and resources owned by the thread will be released immediately when the thread terminates.
- Use this when your thread won't be "join'd"
- Attempting to detach an already detached thread is unspecified behavior
- A detached thread cannot be joined via pthread_join()

Thread Termination, con't.

#include <pthread.h>
int pthread_detach(pthread_t thread);

- All storage and resources owned by the thread will be released immediately when the thread terminates.
- Use this when your thread won't be "join'd"
- Attempting to detach an already detached thread is unspecified behavior
- A detached thread cannot be joined via `pthread_join()`
- When the process exits, detached threads will be terminated and all process and thread resources will be released

Threads and Signals

Design goals of the Posix Threads Committee:

- A thread should be able to send and receive signals without any changes to how a process normally handles signals

Threads and Signals

Design goals of the Posix Threads Committee:

- A thread should be able to send and receive signals without any changes to how a process normally handles signals
- When a signal is delivered to a multi-threaded process, one of the threads will be selected to handle the required action.
  That is, one of the process' threads will be interrupted

Threads and Signals

Design goals of the Posix Threads Committee:

- A thread should be able to send and receive signals without any changes to how a process normally handles signals
- When a signal is delivered to a multi-threaded process, one of the threads will be selected to handle the required action.
  That is, one of the process' threads will be interrupted

Notes on thread signal handling:

Threads and Signals

Design goals of the Posix Threads Committee:

- A thread should be able to send and receive signals without any changes to how a process normally handles signals
- When a signal is delivered to a multi-threaded process, one of the threads will be selected to handle the required action.
  That is, one of the process' threads will be interrupted

Notes on thread signal handling:

- See the "Signal-Safe Function" document on the website

Threads and Signals

Design goals of the Posix Threads Committee:

- A thread should be able to send and receive signals without any changes to how a process normally handles signals
- When a signal is delivered to a multi-threaded process, one of the threads will be selected to handle the required action.
  That is, one of the process' threads will be interrupted

Notes on thread signal handling:

- See the "Signal-Safe Function" document on the website
- Note that no `pthread_···` functions are on this list!
  No mutexes. No condition variables. Etc.

Threads and Signals

Design goals of the Posix Threads Committee:

- A thread should be able to send and receive signals without any changes to how a process normally handles signals
- When a signal is delivered to a multi-threaded process, one of the threads will be selected to handle the required action.
  That is, one of the process' threads will be interrupted

Notes on thread signal handling:

- See the "Signal-Safe Function" document on the website
- Note that no `pthread_`$\cdots$ functions are on this list!
  No mutexes. No condition variables. Etc.
- A thread may use the sem_post() call, however, to communicate with other threads. *link*

Threads and Signals

- To designate a specific thread to handle signals, mask out those signals from all other threads. (see pthread_sigmask())

Threads and Signals

- To designate a specific thread to handle signals, mask out those signals from all other threads. (see pthread_sigmask())
- You may have that designated thread use sigwait() to wait for specified signals. *(see later)* *link*

Threads and Signals

- To designate a specific thread to handle signals, mask out those signals from all other threads. (see pthread_sigmask())
- You may have that designated thread use sigwait() to wait for specified signals. *(see later)* *link*
- Recall sigprocmask() – that was for a process’s mask; for threads, pthread_sigmask() is available.

Threads and Signals

- To designate a specific thread to handle signals, mask out those signals from all other threads. (see pthread_sigmask())
- You may have that designated thread use sigwait() to wait for specified signals. *(see later)* *link*
- Recall sigprocmask() – that was for a process's mask; for threads, pthread_sigmask() is available.
- The signal sets may be built in the usual fashion using *sigsetops*: sigemptyset(), sigaddset(), sigdelset(), sigfillset(), and sigismember().

Threads: pthread_sigmask()

```
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.

Threads: pthread_sigmask()

#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.
- Examine and/or change a thread's signal mask

Threads: pthread_sigmask()

#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.
- Examine and/or change a thread's signal mask
- how:

Threads: pthread_sigmask()

#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.
- Examine and/or change a thread's signal mask
- how:

  SIG_BLOCK resulting set is the union of the current mask with the set

Threads: pthread_sigmask()

#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.
- Examine and/or change a thread's signal mask
- how:

  SIG_BLOCK resulting set is the union of the current mask with the set

  SIG_SETMASK resulting set is the given set

Threads: pthread_sigmask()

#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.
- Examine and/or change a thread's signal mask
- how:

  SIG_BLOCK resulting set is the union of the current mask with the set

  SIG_SETMASK resulting set is the given set

  SIG_UNBLOCK resulting set is the intersection of the current set and the set

Threads: pthread_sigmask()

#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.
- Examine and/or change a thread's signal mask
- how:

  SIG_BLOCK resulting set is the union of the current mask with the set

  SIG_SETMASK resulting set is the given set

  SIG_UNBLOCK resulting set is the intersection of the current set and the set
- If oldset is not null, then the previous mask will be stored there

Threads: pthread_sigmask()

#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.
- Examine and/or change a thread's signal mask
- how:

  SIG_BLOCK resulting set is the union of the current mask with the set

  SIG_SETMASK resulting set is the given set

  SIG_UNBLOCK resulting set is the intersection of the current set and the set
- If oldset is not null, then the previous mask will be stored there
- If set is null, then how is ignored and the thread's mask will remain unchanged.

Threads: pthread_sigmask()

#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

- pthread_sigmask() is just like sigprocmask(), with the obvious proviso that sigprocmask() affected a process, whereas pthread_sigmask() affects only the calling thread.
- Examine and/or change a thread's signal mask
- how:

  SIG_BLOCK resulting set is the union of the current mask with the set

  SIG_SETMASK resulting set is the given set

  SIG_UNBLOCK resulting set is the intersection of the current set and the set
- If oldset is not null, then the previous mask will be stored there
- If set is null, then how is ignored and the thread's mask will remain unchanged.
- Use sigsetops to build sets: sigemptyset(), sigaddset(), sigdelset(), sigfillset(), and sigismember()

Threads: sigwait()

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

- When no signal in set is pending, then the thread (or process) will be suspended until one or more of those signals becomes pending.

Threads: sigwait()

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

- When no signal in set is pending, then the thread (or process) will be suspended until one or more of those signals becomes pending.
- Will accept a signal from the set, atomically clear it from the system's set of pending signals, and return which signal in sig.

Threads: sigwait()

#include <signal.h>
int sigwait(const sigset_t *set, int *sig);

- When no signal in set is pending, then the thread (or process) will be suspended until one or more of those signals becomes pending.
- Will accept a signal from the set, atomically clear it from the system's set of pending signals, and return which signal in sig.
- sigwait() resembles sigwaitinfo(), except that:

Threads: sigwait()

#include <signal.h>
int sigwait(const sigset_t *set, int *sig);

- When no signal in set is pending, then the thread (or process) will be suspended until one or more of those signals becomes pending.
- Will accept a signal from the set, atomically clear it from the system's set of pending signals, and return which signal in sig.
- sigwait() resembles sigwaitinfo(), except that:
  - sigwait() only returns the signal number rather than a siginfo_t structure

Threads: sigwait()

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

- When no signal in set is pending, then the thread (or process) will be suspended until one or more of those signals becomes pending.
- Will accept a signal from the set, atomically clear it from the system's set of pending signals, and return which signal in sig.
- sigwait() resembles sigwaitinfo(), except that:
  - sigwait() only returns the signal number rather than a siginfo_t structure
  - The return values are different

Threads: sigwait()

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

- When no signal in set is pending, then the thread (or process) will be suspended until one or more of those signals becomes pending.
- Will accept a signal from the set, atomically clear it from the system's set of pending signals, and return which signal in sig.
- sigwait() resembles sigwaitinfo(), except that:
  - sigwait() only returns the signal number rather than a siginfo_t structure
  - The return values are different
- On success: sigwait() returns 0.
  On error: sigwait() returns a positive error number

Threads: sem_post()

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

- Unlocks the semaphore pointed to by `sem`.

Threads: sem_post()

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

- Unlocks the semaphore pointed to by `sem`.
- This semaphore is *not* the same as the IPC semaphores presented earlier this semester.

Threads: sem_post()

#include <semaphore.h>
int sem_post(sem_t *sem);

- Unlocks the semaphore pointed to by sem.
- This semaphore is *not* the same as the IPC semaphores presented earlier this semester.
- May be used to synchronize threads or processes (Posix semaphore)

Threads: sem_post()

#include <semaphore.h>
int sem_post(sem_t *sem);

- Unlocks the semaphore pointed to by sem.
- This semaphore is *not* the same as the IPC semaphores presented earlier this semester.
- May be used to synchronize threads or processes (Posix semaphore)
- This function is asynchronous signal safe; it may be safely called within a signal handler

Threads: sem_post()

#include <semaphore.h>
int sem_post(sem_t *sem);

- Unlocks the semaphore pointed to by `sem`.
- This semaphore is *not* the same as the IPC semaphores presented earlier this semester.
- May be used to synchronize threads or processes (Posix semaphore)
- This function is asynchronous signal safe; it may be safely called within a signal handler
- More on Posix semaphores next week

Threads: pthread_kill()

#include <signal.h>
int pthread_kill(pthread_t thread, int sig);

(compile and link with -pthread)

- Sends the specified sig to another thread in the same process

Threads: pthread_kill()

#include <signal.h>

int pthread_kill(pthread_t thread, int sig);

(compile and link with -pthread)

- Sends the specified sig to another thread in the same process
- The signal is asynchronously directed to the thread

Threads: pthread_kill()

#include <signal.h>
int pthread_kill(pthread_t thread, int sig);
(compile and link with -pthread)

- Sends the specified sig to another thread in the same process
- The signal is asynchronously directed to the thread
- Some signal dispositions will affect the entire process (stop, continue, terminate)

Threads: Qty LWPs in a Process

```
#include <pthread.h>
int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);
```

- pthread_setconcurrency() sets the quantity of LWPs to new_level. This is the qty of threads that may be simultaneously active. If new_level is zero, then the default concurrency may be restored.

Threads: Qty LWPs in a Process

```
#include <pthread.h>
int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);
```

- pthread_setconcurrency() sets the quantity of LWPs to new_level. This is the qty of threads that may be simultaneously active. If new_level is zero, then the default concurrency may be restored.
- By default, pthreads insures that a sufficient number of threads will be active. This default qty may not make the most effective use of the hardware's concurrency.

Threads: Qty LWPs in a Process

```
#include <pthread.h>
int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);
```

- pthread_setconcurrency() sets the quantity of LWPs to new_level. This is the qty of threads that may be simultaneously active. If new_level is zero, then the default concurrency may be restored.
- By default, pthreads insures that a sufficient number of threads will be active. This default qty may not make the most effective use of the hardware's concurrency.
- pthread_setconcurrency() actually is only a hint, not a requirement.

Threads: Qty LWPs in a Process

```
#include <pthread.h>
int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);
```

- pthread_setconcurrency() sets the quantity of LWPs to new_level. This is the qty of threads that may be simultaneously active. If new_level is zero, then the default concurrency may be restored.
- By default, pthreads insures that a sufficient number of threads will be active. This default qty may not make the most effective use of the hardware's concurrency.
- pthread_setconcurrency() actually is only a hint, not a requirement.
- If the Linux implementation does not support multiplexing of user threads atop kernel-scheduled entities (LWPs), then these two functions will have no effect when called.

Threads: Qty LWPs in a Process

#include <pthread.h>
int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);

- pthread_setconcurrency() sets the quantity of LWPs to new_level. This is the qty of threads that may be simultaneously active. If new_level is zero, then the default concurrency may be restored.
- By default, pthreads insures that a sufficient number of threads will be active. This default qty may not make the most effective use of the hardware's concurrency.
- pthread_setconcurrency() actually is only a hint, not a requirement.
- If the Linux implementation does not support multiplexing of user threads atop kernel-scheduled entities (LWPs), then these two functions will have no effect when called.
- pthread_getconcurrency() returns the value previously set by pthread_setconcurrency(). If pthread_setconcurrency() was never called, then this function will return 0 indicating that pthreads is maintaining the concurrency level.

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location
  - scheduling policy

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location
  - scheduling policy
  - scheduling priority

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location
  - scheduling policy
  - scheduling priority
- Procedure:

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location
  - scheduling policy
  - scheduling priority
- Procedure:
  1. Define/allocate an attribute object (type: pthread_attr_t)

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location
  - scheduling policy
  - scheduling priority
- Procedure:
  1. Define/allocate an attribute object (type: pthread_attr_t)
  2. Use pthread_attr_init() to initialize the attr object.

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location
  - scheduling policy
  - scheduling priority
- Procedure:
  1. Define/allocate an attribute object (type: pthread_attr_t)
  2. Use pthread_attr_init() to initialize the attr object.
  3. Use various calls to pthread_attr_...() functions to control attributes

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location
  - scheduling policy
  - scheduling priority
- Procedure:
  1. Define/allocate an attribute object (type: pthread_attr_t)
  2. Use pthread_attr_init() to initialize the attr object.
  3. Use various calls to pthread_attr_...() functions to control attributes
  4. Apply attribute via pthread_create(). Attributes may only be specified when a thread is created and cannot be changed.

Threads: Attributes

- Define new or initial attributes (recall pthread_create()'s attr parameter).
- Threads can set or query the following:
  - whether the thread is initially detached or join-able
  - stack control: size, location
  - scheduling policy
  - scheduling priority
- Procedure:
  1. Define/allocate an attribute object (type: pthread_attr_t)
  2. Use pthread_attr_init() to initialize the attr object.
  3. Use various calls to pthread_attr_...() functions to control attributes
  4. Apply attribute via pthread_create(). Attributes may only be specified when a thread is created and cannot be changed.
  5. Destroy attribute (release resources); destroying attributes does not affect threads that have been initialized with them

Attributes: Initialization

```
#include <pthread.h>
pthread_attr_t attrib;
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- pthread_attr_init(): Initializes an attribute with default values.

Attributes: Initialization

```
#include <pthread.h>
pthread_attr_t attrib;
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- pthread_attr_init(): Initializes an attribute with default values.
- pthread_attr_init(): Undefined behavior if used on an already initialized attr object.

Attributes: Initialization

```
#include <pthread.h>
pthread_attr_t attrib;
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- pthread_attr_init(): Initializes an attribute with default values.
- pthread_attr_init(): Undefined behavior if used on an already initialized attr object.
- pthread_attr_init(): After this call, individual attributes may be set (see following slide)

Attributes: Initialization

```
#include <pthread.h>
pthread_attr_t attrib;
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- pthread_attr_init(): Initializes an attribute with default values.
- pthread_attr_init(): Undefined behavior if used on an already initialized attr object.
- pthread_attr_init(): After this call, individual attributes may be set (see following slide)
- pthread_attr_destroy(): Use this call to release resources held by the attribute

Attributes: Initialization

#include <pthread.h>
pthread_attr_t attrib;
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

- pthread_attr_init(): Initializes an attribute with default values.
- pthread_attr_init(): Undefined behavior if used on an already initialized attr object.
- pthread_attr_init(): After this call, individual attributes may be set (see following slide)
- pthread_attr_destroy(): Use this call to release resources held by the attribute
- pthread_attr_destroy(): Destroying an attribute has no effect on threads already created with the attribute

Attributes: Setting up a Thread's Stack

#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);

- If _POSIX_THREAD_ATTR_STACKSIZE is defined, then one may change the initial size of a thread's stack

Attributes: Setting up a Thread's Stack

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
```

- If _POSIX_THREAD_ATTR_STACKSIZE is defined, then one may change the initial size of a thread's stack
- pthread_attr_setstacksize(): sets the minimum stack size attribute

  (in bytes)

Attributes: Setting up a Thread's Stack

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
```

- If _POSIX_THREAD_ATTR_STACKSIZE is defined, then one may change the initial size of a thread's stack
- pthread_attr_setstacksize(): sets the minimum stack size attribute (in bytes)
- pthread_attr_getstacksize(): gets the attribute stack size, setting stacksize with that value

Attributes: Setting up a Thread's Stack

#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);

- If _POSIX_THREAD_ATTR_STACKSIZE is defined, then one may change the initial size of a thread's stack
- pthread_attr_setstacksize(): sets the minimum stack size attribute
  (in bytes)
- pthread_attr_getstacksize(): gets the attribute stack size, setting stacksize with that value
- If _POSIX_THREAD_ATTR_STACKADDR is defined, then one may change where a stack is located

Attributes: Setting up a Thread's Stack

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
```

- If _POSIX_THREAD_ATTR_STACKSIZE is defined, then one may change the initial size of a thread's stack
- pthread_attr_setstacksize(): sets the minimum stack size attribute (in bytes)
- pthread_attr_getstacksize(): gets the attribute stack size, setting stacksize with that value
- If _POSIX_THREAD_ATTR_STACKADDR is defined, then one may change where a stack is located
- pthread_attr_setstackaddr(): sets the stack's base address

Attributes: Setting up a Thread's Stack

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
```

- If _POSIX_THREAD_ATTR_STACKSIZE is defined, then one may change the initial size of a thread's stack
- pthread_attr_setstacksize(): sets the minimum stack size attribute (in bytes)
- pthread_attr_getstacksize(): gets the attribute stack size, setting stacksize with that value
- If _POSIX_THREAD_ATTR_STACKADDR is defined, then one may change where a stack is located
- pthread_attr_setstackaddr(): sets the stack's base address
- pthread_attr_getstackaddr(): gets the stack's base address

Attributes: Setting up a Thread's Stack

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
```

- If _POSIX_THREAD_ATTR_STACKSIZE is defined, then one may change the initial size of a thread's stack
- pthread_attr_setstacksize(): sets the minimum stack size attribute (in bytes)
- pthread_attr_getstacksize(): gets the attribute stack size, setting stacksize with that value
- If _POSIX_THREAD_ATTR_STACKADDR is defined, then one may change where a stack is located
- pthread_attr_setstackaddr(): sets the stack's base address
- pthread_attr_getstackaddr(): gets the stack's base address
- pthread_attr_setstackaddr(): should point to a buffer of at least PTHREAD_STACK_MIN bytes that has been allocated by the caller.
  The pages of the allocated buffer should be both readable and writable.

(See thrdstacksize.c and thrdstack.c)

Attributes: Detached State

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
```

- The detach state attribute determines if a thread is joinable or detached

Attributes: Detached State

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
```

- The detach state attribute determines if a thread is joinable or detached
- You may use `pthread_detach()` to detach a currently running thread

Attributes: Detached State

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
```

- The detach state attribute determines if a thread is joinable or detached
- You may use `pthread_detach()` to detach a currently running thread
- The `pthread_attr_setdetachstate()` function sets the detach state attribute

Attributes: Detached State

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
```

- The detach state attribute determines if a thread is joinable or detached
- You may use `pthread_detach()` to detach a currently running thread
- The `pthread_attr_setdetachstate()` function sets the detach state attribute
- The `pthread_attr_getdetachstate()` function gets the detach state attribute into `*detachstate`

Attributes: Detached State , con't.

- The detachstate is either

Attributes: Detached State , con't.

- The detachstate is either

  PTHREAD_CREATE_JOINABLE (default state) Another thread can wait for a *joinable* thread's completion using pthread_join(). A waiting thread is suspended until the targeted thread terminates. A thread may have terminated before being "joined"; consequently, the system retains information on joinable threads (thread id, returned value).

Attributes: Detached State , con’t.

- The detachstate is either

  PTHREAD_CREATE_JOINABLE (default state) Another thread can wait for a *joinable* thread’s completion using pthread_join(). A waiting thread is suspended until the targeted thread terminates. A thread may have terminated before being “joined”; consequently, the system retains information on joinable threads (thread id, returned value).

  PTHREAD_CREATE_DETACHED Cannot be joined, not expected to return anything useful. When a detached thread terminates, pthreads releases any resources the thread held.

Attributes: Detached State , con't.

- The detachstate is either

  PTHREAD_CREATE_JOINABLE (default state) Another thread can wait for a *joinable* thread's completion using pthread_join(). A waiting thread is suspended until the targeted thread terminates. A thread may have terminated before being "joined"; consequently, the system retains information on joinable threads (thread id, returned value).

  PTHREAD_CREATE_DETACHED Cannot be joined, not expected to return anything useful. When a detached thread terminates, pthreads releases any resources the thread held.

- Caveat: don't join a thread that has already been joined

Attributes: Detached State , con't.

- The detachstate is either

  PTHREAD_CREATE_JOINABLE (default state) Another thread can wait for a *joinable* thread's completion using pthread_join(). A waiting thread is suspended until the targeted thread terminates. A thread may have terminated before being "joined"; consequently, the system retains information on joinable threads (thread id, returned value).

  PTHREAD_CREATE_DETACHED Cannot be joined, not expected to return anything useful. When a detached thread terminates, pthreads releases any resources the thread held.

- Caveat: don't join a thread that has already been joined
- Caveat: don't join a detached thread

Attributes: Detached State , con't.

- The detachstate is either

  PTHREAD_CREATE_JOINABLE (default state) Another thread can wait for a *joinable* thread's completion using pthread_join(). A waiting thread is suspended until the targeted thread terminates. A thread may have terminated before being "joined"; consequently, the system retains information on joinable threads (thread id, returned value).

  PTHREAD_CREATE_DETACHED Cannot be joined, not expected to return anything useful. When a detached thread terminates, pthreads releases any resources the thread held.

- Caveat: don't join a thread that has already been joined
- Caveat: don't join a detached thread
- Caveat: once detached, one cannot "re-attach" it

see thrd_detached.c

Scheduling Scope

scheduling scope: which threads contend for processing

system scope Threads with system scope contend with all other threads with system scope, irrespective of which process to which they belong.
These threads are schedule by the kernel; context switches usually require at least one call into the kernel. This context switch is often more expensive than process scope context switches.

Scheduling Scope

scheduling scope: which threads contend for processing

system scope Threads with system scope contend with all other threads with system scope, irrespective of which process to which they belong.
These threads are schedule by the kernel; context switches usually require at least one call into the kernel. This context switch is often more expensive than process scope context switches.

process scope Threads with process scope contend only with other threads in the same process. These are lightweight process context switches, and may use the same or some small number of kernel entities (LWPs).
However, lower priority threads from other processes may run instead of your process' high priority thread.

Allocation Domains

scheduling allocation domain: a group of one or more processes which act as separate (LWP) scheduling targets.

- Having multiple processors doesn't mean scheduling domains are greater than one.

Allocation Domains

scheduling allocation domain: a group of one or more processes which act as separate (LWP) scheduling targets.

- Having multiple processors doesn't mean scheduling domains are greater than one.
- Conforming implementations may map all or any subset of CPUs to one or multiple scheduling allocation domains

Allocation Domains

scheduling allocation domain: a group of one or more processes which act as separate (LWP) scheduling targets.

- Having multiple processors doesn't mean scheduling domains are greater than one.
- Conforming implementations may map all or any subset of CPUs to one or multiple scheduling allocation domains
- ASDs may be done on a per-thread, per-process, or per-system basis.

Allocation Domains

scheduling allocation domain: a group of one or more processes which act as separate (LWP) scheduling targets.

- Having multiple processors doesn’t mean scheduling domains are greater than one.
- Conforming implementations may map all or any subset of CPUs to one or multiple scheduling allocation domains
- ASDs may be done on a per-thread, per-process, or per-system basis.
- ASD is independent of scheduling contention scope, which defines the set of threads with which a thread contends for processor resources

Allocation Domains

scheduling allocation domain: a group of one or more processes which act as separate (LWP) scheduling targets.

- Having multiple processors doesn't mean scheduling domains are greater than one.
- Conforming implementations may map all or any subset of CPUs to one or multiple scheduling allocation domains
- ASDs may be done on a per-thread, per-process, or per-system basis.
- ASD is independent of scheduling contention scope, which defines the set of threads with which a thread contends for processor resources
- An ASD defines the set of processors for which a thread may contend

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS
pset_destroy	destroy a PS

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS
pset_destroy	destroy a PS
pset_assign	reassign a processor from one PS to another

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS
pset_destroy	destroy a PS
pset_assign	reassign a processor from one PS to another
pset_bind	rebind a process/thread from one PS to another

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS
pset_destroy	destroy a PS
pset_assign	reassign a processor from one PS to another
pset_bind	rebind a process/thread from one PS to another
pset_setattr	change PS attribute values

Allocation Domains, con't.

- For example, HP-UX defines "processor sets".
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS
pset_destroy	destroy a PS
pset_assign	reassign a processor from one PS to another
pset_bind	rebind a process/thread from one PS to another
pset_setattr	change PS attribute values
pset_ctl	query PS configuration

Allocation Domains, con’t.

- For example, HP-UX defines “processor sets”.
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS
pset_destroy	destroy a PS
pset_assign	reassign a processor from one PS to another
pset_bind	rebind a process/thread from one PS to another
pset_setattr	change PS attribute values
pset_ctl	query PS configuration
pthread_pset_bind_np	change pset binding of a pthread

Allocation Domains, con’t.

- For example, HP-UX defines “processor sets”.
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS
pset_destroy	destroy a PS
pset_assign	reassign a processor from one PS to another
pset_bind	rebind a process/thread from one PS to another
pset_setattr	change PS attribute values
pset_ctl	query PS configuration
pthread_pset_bind_np	change pset binding of a pthread
mpctl	apply processor affinity to applications

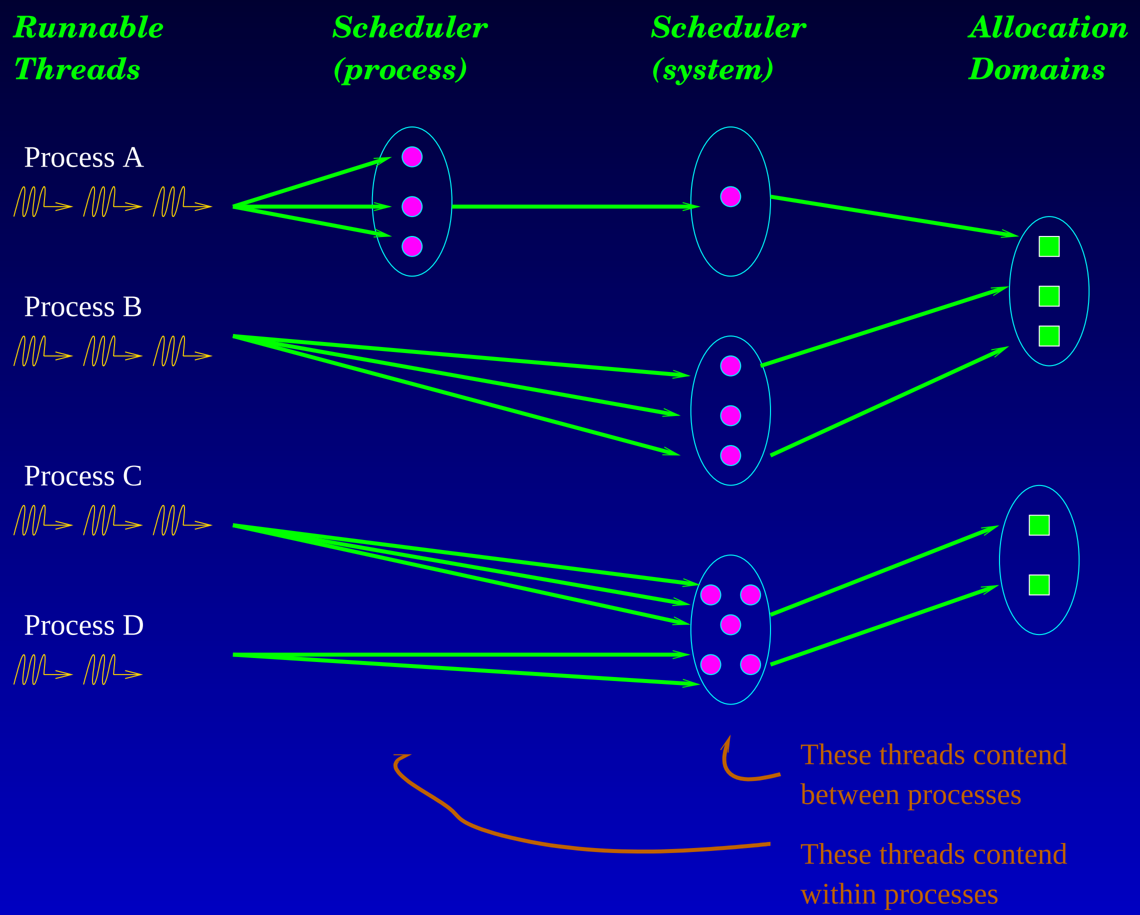
Allocation Domains, con’t.

- For example, HP-UX defines “processor sets”.
  - By default, the entire system is the scheduling allocation domain for an application.
  - One may group processors together for the exclusive use of specific applications
  - Threads in the same process may be bound to different processor sets

API	*Description*
pset_create	create a new PS
pset_destroy	destroy a PS
pset_assign	reassign a processor from one PS to another
pset_bind	rebind a process/thread from one PS to another
pset_setattr	change PS attribute values
pset_ctl	query PS configuration
pthread_pset_bind_np	change pset binding of a pthread
mpctl	apply processor affinity to applications within the app’s PS

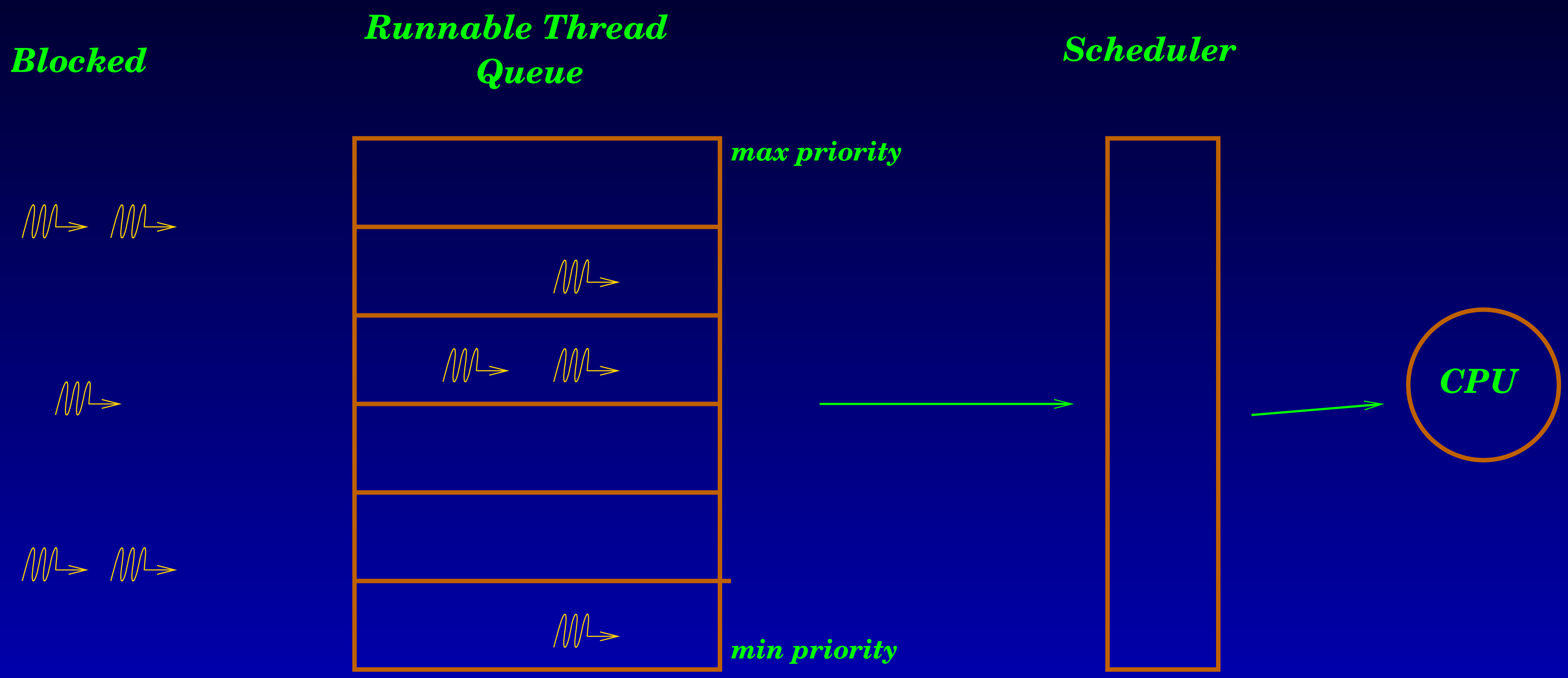
PS=processor set

Allocation Domains, con't.



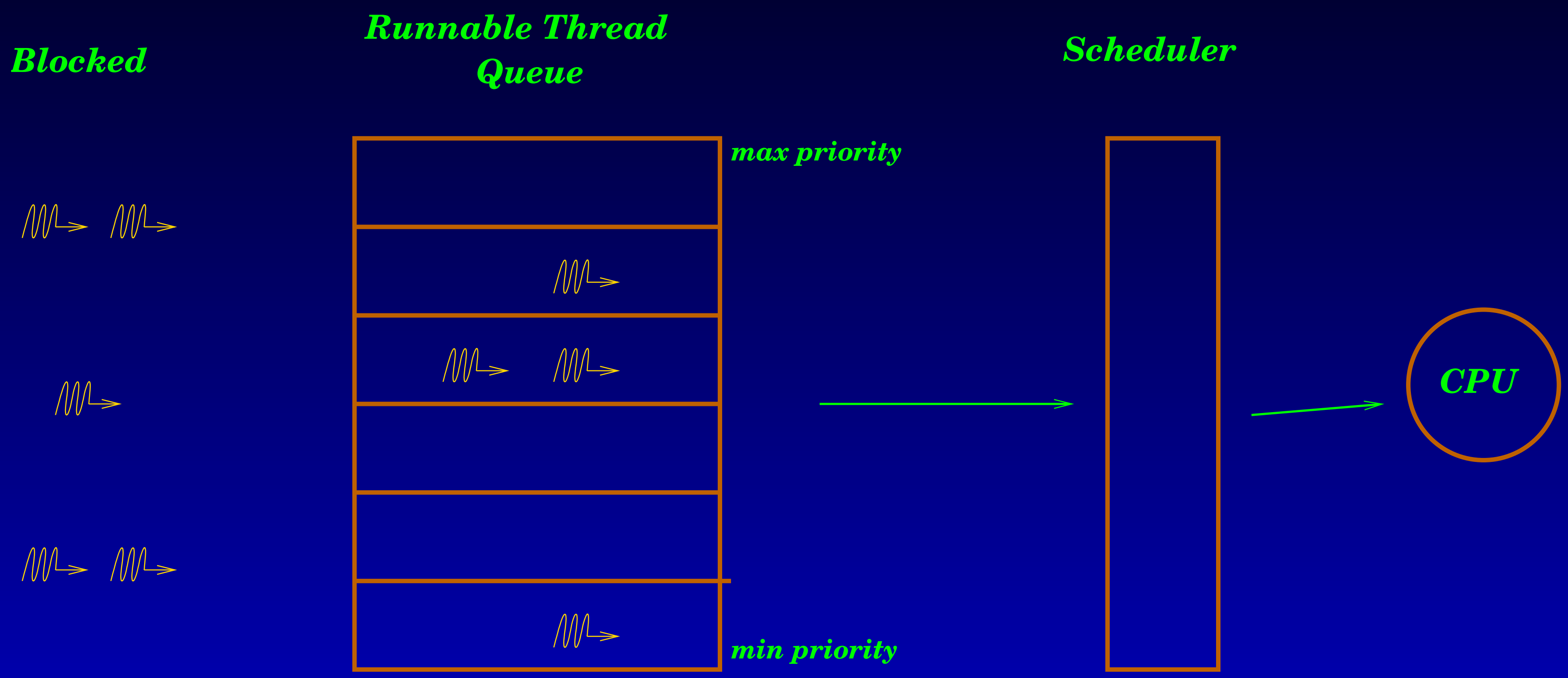
- With allocation domains, if threads are all blocked (say, C and D), then the CPUs in that domain may be idle even though the other domain's CPUs are running full bore with waiting threads

Scheduling Priority



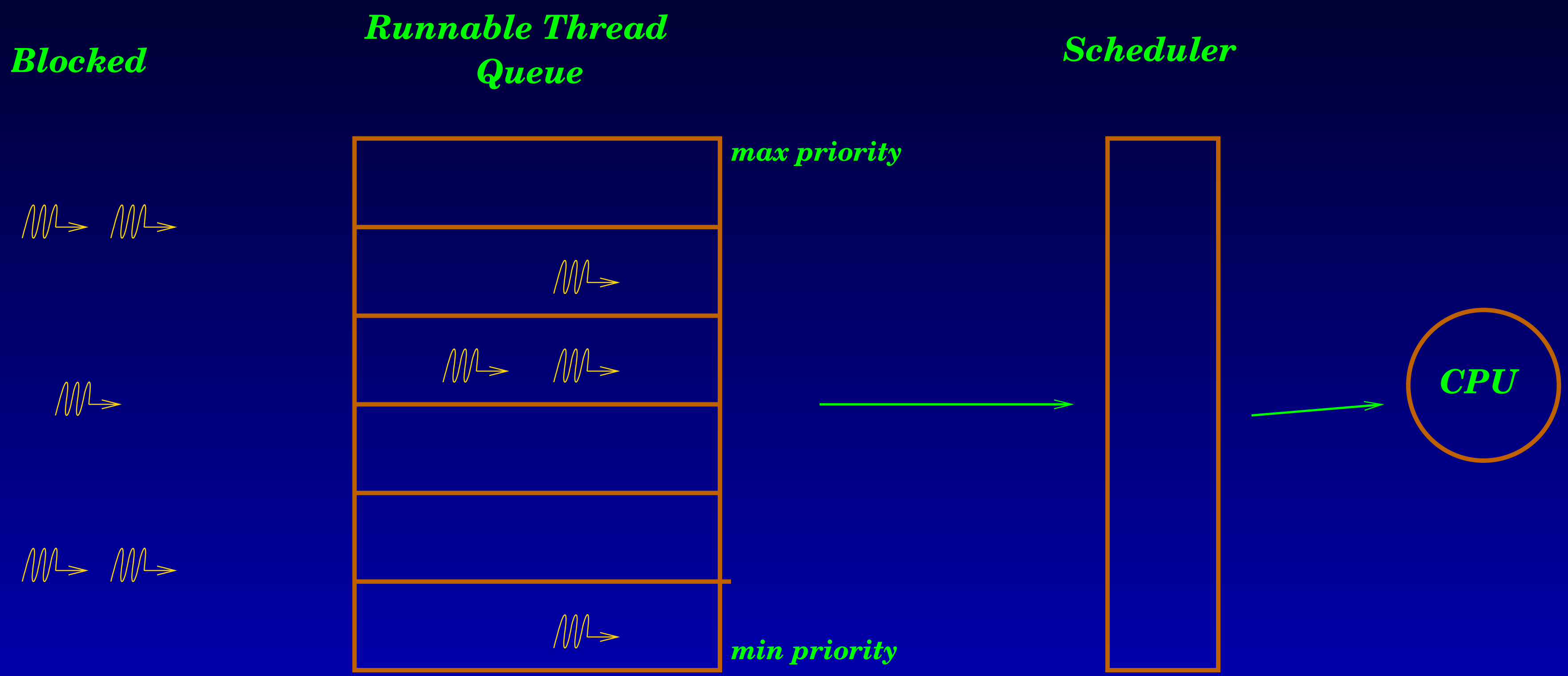
- Threads are placed on priority queues

Scheduling Priority



- Threads are placed on priority queues
- The scheduler looks for the highest priority thread available to run

Scheduling Priority



- Threads are placed on priority queues
- The scheduler looks for the highest priority thread available to run
- Whenever a higher priority thread becomes runnable than the currently running thread's priority, the lower priority thread is interrupted and put at the end of its priority queue. This is known as an ***involuntary context switch***.

Scheduling Policy

Scheduling Policy: determines how long a thread runs after having been moved from a priority queue to a processing slot.

SCHED_FIFO : (first-in first-out) threads run until they block or exit. When a blocked thread again becomes runnable, it is placed at the back of its priority queue

Scheduling Policy

Scheduling Policy: determines how long a thread runs after having been moved from a priority queue to a processing slot.

SCHED_FIFO : (first-in first-out) threads run until they block or exit. When a blocked thread again becomes runnable, it is placed at the back of its priority queue

SCHED_RR : (round robin) allows a thread to run for a fixed amount of time known as a ***quantum***. Such an interrupted thread is placed at the back of its priority queue.

Scheduling Policy

Scheduling Policy: determines how long a thread runs after having been moved from a priority queue to a processing slot.

SCHED_FIFO : (first-in first-out) threads run until they block or exit. When a blocked thread again becomes runnable, it is placed at the back of its priority queue

SCHED_RR : (round robin) allows a thread to run for a fixed amount of time known as a ***quantum***. Such an interrupted thread is placed at the back of its priority queue.

SCHED_OTHER : (default) up to the implementation; usually its a time-sharing policy (ie. RR) with priority adjustments that give i/o bound threads preference (vs cpu-bound). (ie. threads that don't use all of their quantum before blocking)

Scheduling Policy

Scheduling Policy: determines how long a thread runs after having been moved from a priority queue to a processing slot.

SCHED_FIFO : (first-in first-out) threads run until they block or exit. When a blocked thread again becomes runnable, it is placed at the back of its priority queue

SCHED_RR : (round robin) allows a thread to run for a fixed amount of time known as a ***quantum***. Such an interrupted thread is placed at the back of its priority queue.

SCHED_OTHER : (default) up to the implementation; usually its a time-sharing policy (ie. RR) with priority adjustments that give i/o bound threads preference (vs cpu-bound). (ie. threads that don't use all of their quantum before blocking)

These attributes may be modified (and may require root privilege to do so) if one of the following is #defined:

Scheduling Policy

Scheduling Policy: determines how long a thread runs after having been moved from a priority queue to a processing slot.

SCHED_FIFO : (first-in first-out) threads run until they block or exit. When a blocked thread again becomes runnable, it is placed at the back of its priority queue

SCHED_RR : (round robin) allows a thread to run for a fixed amount of time known as a ***quantum***. Such an interrupted thread is placed at the back of its priority queue.

SCHED_OTHER : (default) up to the implementation; usually its a time-sharing policy (ie. RR) with priority adjustments that give i/o bound threads preference (vs cpu-bound). (ie. threads that don't use all of their quantum before blocking)

These attributes may be modified (and may require root privilege to do so) if one of the following is #defined:

_POSIX_THREAD_PRIORITY_SCHEDULING

Scheduling Policy

Scheduling Policy: determines how long a thread runs after having been moved from a priority queue to a processing slot.

SCHED_FIFO : (first-in first-out) threads run until they block or exit. When a blocked thread again becomes runnable, it is placed at the back of its priority queue

SCHED_RR : (round robin) allows a thread to run for a fixed amount of time known as a ***quantum***. Such an interrupted thread is placed at the back of its priority queue.

SCHED_OTHER : (default) up to the implementation; usually its a time-sharing policy (ie. RR) with priority adjustments that give i/o bound threads preference (vs cpu-bound). (ie. threads that don't use all of their quantum before blocking)

These attributes may be modified (and may require root privilege to do so) if one of the following is #defined:

_POSIX_THREAD_PRIORITY_SCHEDULING

_POSIX_THREAD_ATTR_PRIORITY_SCHEDULING

Threads: Scope

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);
```

- The `pthread_attr_setscope()` function sets the thread's contention scope for the attribute object `attr`.

Threads: Scope

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);
```

- The pthread_attr_setscope() function sets the thread's contention scope for the attribute object attr.
- Posix provides two possible values: PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS.

Threads: Scope

#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);

- The pthread_attr_setscope() function sets the thread's contention scope for the attribute object attr.
- Posix provides two possible values: PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS.
- Posix does requires that Unix implementations support at least one of these scopes; linux only supports PTHREAD_SCOPE_SYSTEM.

Threads: Scope

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);
```

- The pthread_attr_setscope() function sets the thread's contention scope for the attribute object attr.
- Posix provides two possible values: PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS.
- Posix does requires that Unix implementations support at least one of these scopes; linux only supports PTHREAD_SCOPE_SYSTEM.
- The pthread_attr_getscope() function returns the current scope attribute of the thread attribute object attr.

Threads: Scope

#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);

- The pthread_attr_setscope() function sets the thread's contention scope for the attribute object attr.
- Posix provides two possible values: PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS.
- Posix does requires that Unix implementations support at least one of these scopes; linux only supports PTHREAD_SCOPE_SYSTEM.
- The pthread_attr_getscope() function returns the current scope attribute of the thread attribute object attr.

(See thrdscope.c)

Threads: Policy

```
#include <pthread.h>
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
```

- The pthread_attr_setschedpolicy() function sets the thread's scheduling policy for the attribute object attr.

Threads: Policy

#include <pthread.h>
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);

- The pthread_attr_setschedpolicy() function sets the thread's scheduling policy for the attribute object attr.
- The pthread_attr_getschedpolicy() function returns the current scheduling policy attribute of the thread attribute object attr.

Threads: Policy

```
#include <pthread.h>
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
```

- The pthread_attr_setschedpolicy() function sets the thread's scheduling policy for the attribute object attr.
- The pthread_attr_getschedpolicy() function returns the current scheduling policy attribute of the thread attribute object attr.
- policy may be SCHED_FIFO, SCHED_RR, or SCHED_OTHER.

Threads: Policy

```
#include <pthread.h>
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
```

- The pthread_attr_setschedpolicy() function sets the thread's scheduling policy for the attribute object attr.
- The pthread_attr_getschedpolicy() function returns the current scheduling policy attribute of the thread attribute object attr.
- policy may be SCHED_FIFO, SCHED_RR, or SCHED_OTHER.

(See thrdpolicy.c)

Threads: Priority Min/Max

```
#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- Processes with numerically higher priority values are scheduled before processes with numerically lower priority values

Threads: Priority Min/Max

#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

- Processes with numerically higher priority values are scheduled before processes with numerically lower priority values
- Scheduling algorithms may differ in their priority max/min values

Threads: Priority Min/Max

```
#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- Processes with numerically higher priority values are scheduled before processes with numerically lower priority values
- Scheduling algorithms may differ in their priority max/min values
- Supported policy values are: SCHED_FIFO, SCHED_RR, or SCHED_OTHER

Threads: Priority Min/Max

```
#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- Processes with numerically higher priority values are scheduled before processes with numerically lower priority values
- Scheduling algorithms may differ in their priority max/min values
- Supported policy values are: SCHED_FIFO, SCHED_RR, or SCHED_OTHER
- sched_get_priority_max(): returns the maximum priority value that can be used with the given policy

Threads: Priority Min/Max

```
#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- Processes with numerically higher priority values are scheduled before processes with numerically lower priority values
- Scheduling algorithms may differ in their priority max/min values
- Supported policy values are: SCHED_FIFO, SCHED_RR, or SCHED_OTHER
- sched_get_priority_max(): returns the maximum priority value that can be used with the given policy
- sched_get_priority_min(): returns the minimum priority value that can be used with the given policy

Threads: Priority Min/Max

#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

- Processes with numerically higher priority values are scheduled before processes with numerically lower priority values
- Scheduling algorithms may differ in their priority max/min values
- Supported policy values are: SCHED_FIFO, SCHED_RR, or SCHED_OTHER
- sched_get_priority_max(): returns the maximum priority value that can be used with the given policy
- sched_get_priority_min(): returns the minimum priority value that can be used with the given policy
- Posix requires at least 32 priority levels for SCHED_FIFO and SCHED_RR.

Threads: Priority Min/Max

```
#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- Processes with numerically higher priority values are scheduled before processes with numerically lower priority values
- Scheduling algorithms may differ in their priority max/min values
- Supported `policy` values are: `SCHED_FIFO`, `SCHED_RR`, or `SCHED_OTHER`
- `sched_get_priority_max()`: returns the maximum priority value that can be used with the given `policy`
- `sched_get_priority_min()`: returns the minimum priority value that can be used with the given `policy`
- Posix requires at least 32 priority levels for `SCHED_FIFO` and `SCHED_RR`.

(*See thrdprio.c*)

Threads: Setting Scheduling Parameters

#include <pthread.h>
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);

```
struct sched_param {
    int sched_priority; // only required member
}
```

1. Declare a sched_param structure object

Threads: Setting Scheduling Parameters

```
#include <pthread.h>
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);
    struct sched_param {
        int sched_priority; // only required member
        }
```

1. Declare a sched_param structure object
2. Pick a policy via pthread_attr_setsched()

Threads: Setting Scheduling Parameters

```
#include <pthread.h>
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);
    struct sched_param {
        int sched_priority; // only required member
        }
```

1. Declare a sched_param structure object
2. Pick a policy via pthread_attr_setsched()
3. Pick a priority within the policy's min/max bounds

Threads: Setting Scheduling Parameters

```
#include <pthread.h>
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);
    struct sched_param {
        int sched_priority; // only required member
        }
```

1. Declare a sched_param structure object
2. Pick a policy via pthread_attr_setsched()
3. Pick a priority within the policy's min/max bounds
4. Assign it to ***.sched_priority

Threads: Setting Scheduling Parameters

```
#include <pthread.h>
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);
    struct sched_param {
        int sched_priority; // only required member
        }
```

1. Declare a `sched_param` structure object
2. Pick a policy via `pthread_attr_setsched()`
3. Pick a priority within the policy's min/max bounds
4. Assign it to `***.sched_priority`
5. Use it with `pthread_attr_setschedparam()`

Threads: Setting Scheduling Parameters

```
#include <pthread.h>
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);
    struct sched_param {
        int sched_priority; // only required member
        }
```

1. Declare a `sched_param` structure object
2. Pick a policy via `pthread_attr_setsched()`
3. Pick a priority within the policy's min/max bounds
4. Assign it to `***.sched_priority`
5. Use it with `pthread_attr_setschedparam()`

- This sets up a static priority attribute for use with `pthread_create()`.

Threads: Dynamic Priority

#include <pthread.h>
pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);
pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);

- The pthread_setschedparam() function sets the scheduling policy and parameters of the specified thread.

Threads: Dynamic Priority

#include <pthread.h>
pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);
pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);

- The pthread_setschedparam() function sets the scheduling policy and parameters of the specified thread.
- Policies available are the usual three: SCHED_FIFO, SCHED_RR, and SCHED_OTHER.

Threads: Dynamic Priority

#include <pthread.h>

pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);

pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);

- The `pthread_setschedparam()` function sets the scheduling policy and parameters of the specified `thread`.
- Policies available are the usual three: `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`.
- Set priority with `***.sched_priority` in a `sched_param` object.

Threads: Dynamic Priority

#include <pthread.h>
pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);
pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);

- The pthread_setschedparam() function sets the scheduling policy and parameters of the specified thread.
- Policies available are the usual three: SCHED_FIFO, SCHED_RR, and SCHED_OTHER.
- Set priority with ***.sched_priority in a sched_param object.
- You may set a thread's priority higher than the current thread's priority, causing the current thread to be *immediately pre-empted*.

Threads: Inheritance

```
#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr,int inheritsched);
int pthread_attr_getinheritsched(pthread_attr_t *attr,int *inheritsched);
```

- You may have child threads inherit scheduling attributes from the parent thread instead of explicitly specifying it

Threads: Inheritance

#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr,int inheritsched);
int pthread_attr_getinheritsched(pthread_attr_t *attr,int *inheritsched);

- You may have child threads inherit scheduling attributes from the parent thread instead of explicitly specifying it
- inheritsched may have one of two values:

Threads: Inheritance

#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr,int inheritsched);
int pthread_attr_getinheritsched(pthread_attr_t *attr,int *inheritsched);

- You may have child threads inherit scheduling attributes from the parent thread instead of explicitly specifying it
- inheritsched may have one of two values:

PTHREAD_INHERIT_SCHED (default) threads created using this inheritance attribute take their scheduling attributes from the parent thread. Other scheduling attributes in attr, if any, are ignored.

Threads: Inheritance

#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr,int inheritsched);
int pthread_attr_getinheritsched(pthread_attr_t *attr,int *inheritsched);

- You may have child threads inherit scheduling attributes from the parent thread instead of explicitly specifying it
- inheritsched may have one of two values:

PTHREAD_INHERIT_SCHED (default) threads created using this inheritance attribute take their scheduling attributes from the parent thread.
Other scheduling attributes in attr, if any, are ignored.

PTHREAD_EXPLICIT_SCHED threads created using attr take their scheduling attributes from attr, *not* the parent thread

Threads: Inheritance

#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr,int inheritsched);

int pthread_attr_getinheritsched(pthread_attr_t *attr,int *inheritsched);

- You may have child threads inherit scheduling attributes from the parent thread instead of explicitly specifying it
- inheritsched may have one of two values:

  PTHREAD_INHERIT_SCHED (default) threads created using this inheritance attribute take their scheduling attributes from the parent thread. Other scheduling attributes in attr, if any, are ignored.

  PTHREAD_EXPLICIT_SCHED threads created using attr take their scheduling attributes from attr, *not* the parent thread
- pthread_attr_getinheritsched() returns the inherit scheduler attribute in *inheritsched

Threads: Cancelling Threads

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- The `pthread_cancel()` function sends a cancellation request to the target `thread`.

Threads: Cancelling Threads

#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);

- The pthread_cancel() function sends a cancellation request to the target thread.
- The thread may control its cancelability state via pthread_setcancelstate().

Threads: Cancelling Threads

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- The pthread_cancel() function sends a cancellation request to the target thread.
- The thread may control its cancelability state via pthread_setcancelstate().
- The cancellation request will remain queued until the thread enables cancellation.

Threads: Cancelling Threads

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- The pthread_cancel() function sends a cancellation request to the target thread.
- The thread may control its cancelability state via pthread_setcancelstate().
- The cancellation request will remain queued until the thread enables cancellation.
- Use pthread_setcancelstate() to set its *state* to PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE, as wanted

Threads: Cancelling Threads

#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);

- The pthread_cancel() function sends a cancellation request to the target thread.
- The thread may control its cancelability state via pthread_setcancelstate().
- The cancellation request will remain queued until the thread enables cancellation.
- Use pthread_setcancelstate() to set its *state* to PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE, as wanted
- Use pthread_setcanceltype() to set its cancellability *type*, which may be either

Threads: Cancelling Threads

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- The pthread_cancel() function sends a cancellation request to the target thread.
- The thread may control its cancelability state via pthread_setcancelstate().
- The cancellation request will remain queued until the thread enables cancellation.
- Use pthread_setcancelstate() to set its *state* to PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE, as wanted
- Use pthread_setcanceltype() to set its cancellability *type*, which may be either
  PTHREAD_CANCEL_DEFERRED (default) cancellation requests are deferred until the thread next calls a function that is a cancellation point. See man 7 pthreads for a list of such functions.

Threads: Cancelling Threads

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- The pthread_cancel() function sends a cancellation request to the target thread.
- The thread may control its cancelability state via pthread_setcancelstate().
- The cancellation request will remain queued until the thread enables cancellation.
- Use pthread_setcancelstate() to set its *state* to PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE, as wanted
- Use pthread_setcanceltype() to set its cancellability *type*, which may be either

PTHREAD_CANCEL_DEFERRED (default) cancellation requests are deferred until the thread next calls a function that is a cancellation point. See man 7 pthreads for a list of such functions.

PTHREAD_CANCEL_ASYNCHRONOUS the thread can be cancelled at any time, usually immediately upon receipt of such a request.

Threads: Cancelling Threads

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- The pthread_cancel() function sends a cancellation request to the target thread.
- The thread may control its cancelability state via pthread_setcancelstate().
- The cancellation request will remain queued until the thread enables cancellation.
- Use pthread_setcancelstate() to set its *state* to PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE, as wanted
- Use pthread_setcanceltype() to set its cancellability *type*, which may be either

  PTHREAD_CANCEL_DEFERRED (default) cancellation requests are deferred until the thread next calls a function that is a cancellation point. See man 7 pthreads for a list of such functions.

  PTHREAD_CANCEL_ASYNCHRONOUS the thread can be cancelled at any time, usually immediately upon receipt of such a request.
- The *set&get* operations performed by the two setcancel functions are *atomic*

Threads: Cancelling Threads

#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);

- The pthread_cancel() function sends a cancellation request to the target thread.
- The thread may control its cancelability state via pthread_setcancelstate().
- The cancellation request will remain queued until the thread enables cancellation.
- Use pthread_setcancelstate() to set its *state* to PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE, as wanted
- Use pthread_setcanceltype() to set its cancellability *type*, which may be either

  PTHREAD_CANCEL_DEFERRED (default) cancellation requests are deferred until the thread next calls a function that is a cancellation point. See man 7 pthreads for a list of such functions.

  PTHREAD_CANCEL_ASYNCHRONOUS the thread can be cancelled at any time, usually immediately upon receipt of such a request.
- The *set&get* operations performed by the two setcancel functions are *atomic*

(See thrd_defines.c)

Threads: Controlled Cleanup

#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *), void *arg);

void pthread_cleanup_pop(int execute);

- The pthread_cleanup_push() function schedules a routine to be called when a thread is terminating

  (but not when its returning from its starting routine)

Threads: Controlled Cleanup

#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);

- The pthread_cleanup_push() function schedules a routine to be called when a thread is terminating

  (but not when its returning from its starting routine)

- When the function *routine* is invoked later, it will be given *arg* as its argument

Threads: Controlled Cleanup

#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);

- The pthread_cleanup_push() function schedules a routine to be called when a thread is terminating

  (but not when its returning from its starting routine)

- When the function *routine* is invoked later, it will be given *arg* as its argument

- The pthread_cleanup_pop() function removes the routine at the top of the cleanup stack and (optionally) invokes it.

Threads: Controlled Cleanup

#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);

- The pthread_cleanup_push() function schedules a routine to be called when a thread is terminating

  (but not when its returning from its starting routine)

- When the function *routine* is invoked later, it will be given *arg* as its argument
- The pthread_cleanup_pop() function removes the routine at the top of the cleanup stack and (optionally) invokes it.
- When the thread is cancelled or terminated via pthread_exit(), all stacked cleanup handlers are popped and executed in the reverse of pushing order

Threads: Controlled Cleanup

#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);

- The pthread_cleanup_push() function schedules a routine to be called when a thread is terminating

  (but not when its returning from its starting routine)

- When the function *routine* is invoked later, it will be given *arg* as its argument
- The pthread_cleanup_pop() function removes the routine at the top of the cleanup stack and (optionally) invokes it.
- When the thread is cancelled or terminated via pthread_exit(), all stacked cleanup handlers are popped and executed in the reverse of pushing order
- When pthread_cleanup_pop() is called with a non-zero argument, the topmost cleanup handler is popped and executed

Threads: Controlled Cleanup

#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);

- The pthread_cleanup_push() function schedules a routine to be called when a thread is terminating

  (but not when its returning from its starting routine)

- When the function *routine* is invoked later, it will be given *arg* as its argument
- The pthread_cleanup_pop() function removes the routine at the top of the cleanup stack and (optionally) invokes it.
- When the thread is cancelled or terminated via pthread_exit(), all stacked cleanup handlers are popped and executed in the reverse of pushing order
- When pthread_cleanup_pop() is called with a non-zero argument, the topmost cleanup handler is popped and executed
- The thread cleanup function resembled atexit() (which pushes process-level cleanup function(s))

Thread Synchronization

- pthread_join(): a thread can suspend its execution until another thread terminates

Thread Synchronization

- pthread_join(): a thread can suspend its execution until another thread terminates
- **mutexes**: mutual exclusion lock. These control access to data by denying access to other threads; they resemble lightweight semaphores

Thread Synchronization

- pthread_join(): a thread can suspend its execution until another thread terminates
- **mutexes**: mutual exclusion lock. These control access to data by denying access to other threads; they resemble lightweight semaphores
- **condition variable** functions: lets threads synchronize on the value of a c.v.

Thread Synchronization

- pthread_join(): a thread can suspend its execution until another thread terminates
- **mutexes**: mutual exclusion lock. These control access to data by denying access to other threads; they resemble lightweight semaphores
- **condition variable** functions: lets threads synchronize on the value of a c.v.
- **#include <pthread.h>**
  int pthread_once(pthread_once_t *once_control,void (*init_routine)(void));
  pthread_once_t once_control = PTHREAD_ONCE_INIT;
  pthread_once(): the first call to this function by any thread in a process, with a given once_control, shall succeed in calling the routine().
  Subsequent calls will not succeed.
  Allows one to specify a thread initialization routine.

Thread Synchronization

- pthread_join(): a thread can suspend its execution until another thread terminates
- **mutexes**: mutual exclusion lock. These control access to data by denying access to other threads; they resemble lightweight semaphores
- **condition variable** functions: lets threads synchronize on the value of a c.v.
- **#include <pthread.h>**
  int pthread_once(pthread_once_t *once_control,void (*init_routine)(void));
  pthread_once_t once_control = PTHREAD_ONCE_INIT;
  pthread_once(): the first call to this function by any thread in a process, with a given once_control, shall succeed in calling the routine().
  Subsequent calls will not succeed.
  Allows one to specify a thread initialization routine.
  - One has multiple threads
  - Each thread has a call to pthread_once()
  - Only the first such call will succeeded in calling routine()

Sync: Mutex Variables

- Useful for protecting *critical sections* of code; sections that should be run by one and only one thread at a time; its useful for granting exclusive access to a resource/data.

Sync: Mutex Variables

- Useful for protecting *critical sections* of code; sections that should be run by one and only one thread at a time; its useful for granting exclusive access to a resource/data.
- On 32-bit or 64-bit systems, a double or structure copy, multiply, etc may involve several loads and stores, and so will be vulnerable to being halfway-complete on a context switch

Sync: Mutex Variables

- Useful for protecting *critical sections* of code; sections that should be run by one and only one thread at a time; its useful for granting exclusive access to a resource/data.
- On 32-bit or 64-bit systems, a double or structure copy, multiply, etc may involve several loads and stores, and so will be vulnerable to being halfway-complete on a context switch
- Create and initialize a mutex: usually done in the main process before creating any threads (or via pthread_once())

Sync: Mutex Variables

- Useful for protecting *critical sections* of code; sections that should be run by one and only one thread at a time; its useful for granting exclusive access to a resource/data.
- On 32-bit or 64-bit systems, a double or structure copy, multiply, etc may involve several loads and stores, and so will be vulnerable to being halfway-complete on a context switch
- Create and initialize a mutex: usually done in the main process before creating any threads (or via pthread_once())
- Use mutexes via pthread_mutex_lock() or pthread_mutex_trylock() (see later slide)

Sync: Initializing a Mutex

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t
*restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Destroying an initialized mutex that is unlocked is safe, but destroying a locked mutex is *undefined behavior*.

Sync: Initializing a Mutex

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t
*restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Destroying an initialized mutex that is unlocked is safe, but destroying a locked mutex is *undefined behavior*.
- Use PTHREAD_MUTEX_INITIALIZER for easy statically-allocated mutex initialization. Same as using pthread_mutex_init() with attr set to null.

Sync: Initializing a Mutex

#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

- Destroying an initialized mutex that is unlocked is safe, but destroying a locked mutex is *undefined behavior*.
- Use PTHREAD_MUTEX_INITIALIZER for easy statically-allocated mutex initialization. Same as using pthread_mutex_init() with attr set to null.
- Mutexes may themselves have attributes (see later slide)

Sync: Locking/Unlocking a Mutex

```
#include <pthread.h>

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

- Only one thread may own/hold a mutex at a time

Sync: Locking/Unlocking a Mutex

```
#include <pthread.h>

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

- Only one thread may own/hold a mutex at a time
- pthread_mutex_trylock() returns a 0 on success or -1 on failure to be able to acquire a lock
- Potential problems:

Sync: Locking/Unlocking a Mutex

```
#include <pthread.h>

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

- Only one thread may own/hold a mutex at a time
- pthread_mutex_trylock() returns a 0 on success or -1 on failure to be able to acquire a lock
- Potential problems:
  - Locking an uninitialized mutex

Sync: Locking/Unlocking a Mutex

```
#include <pthread.h>

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

- Only one thread may own/hold a mutex at a time
- pthread_mutex_trylock() returns a 0 on success or -1 on failure to be able to acquire a lock
- Potential problems:
  - Locking an uninitialized mutex
  - Locking a mutex your thread already owns

Sync: Locking/Unlocking a Mutex

```
#include <pthread.h>

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

- Only one thread may own/hold a mutex at a time
- pthread_mutex_trylock() returns a 0 on success or -1 on failure to be able to acquire a lock
- Potential problems:
  - Locking an uninitialized mutex
  - Locking a mutex your thread already owns
  - Unlocking a mutex your thread does not own

Sync: Locking/Unlocking a Mutex

```
#include <pthread.h>

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

- Only one thread may own/hold a mutex at a time
- pthread_mutex_trylock() returns a 0 on success or -1 on failure to be able to acquire a lock
- Potential problems:
  - Locking an uninitialized mutex
  - Locking a mutex your thread already owns
  - Unlocking a mutex your thread does not own

(See thrdmutex.c)

Sync: Mutex Attributes

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init    (pthread_mutexattr_t *attr);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- `pthread_mutexattr_destroy()` releases any resources held by a mutex attribute. Using it after destroying it is *undefined behavior*.

Sync: Mutex Attributes

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init    (pthread_mutexattr_t *attr);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- pthread_mutexattr_destroy() releases any resources held by a mutex attribute. Using it after destroying it is *undefined behavior*.
- pthread_mutexattr_init() initializes a mutex with default attributes; *undefined behavior* results when applying this function to an already initialized mutex

Sync: Mutex Attributes

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init     (pthread_mutexattr_t *attr);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- pthread_mutexattr_destroy() releases any resources held by a mutex attribute. Using it after destroying it is *undefined behavior*.
- pthread_mutexattr_init() initializes a mutex with default attributes; *undefined behavior* results when applying this function to an already initialized mutex
- One may set a mutex attribute to one of four types, which determines what happens when a thread locks a mutex it already owns:

Sync: Mutex Attributes

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init      (pthread_mutexattr_t *attr);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- `pthread_mutexattr_destroy()` releases any resources held by a mutex attribute. Using it after destroying it is *undefined behavior*.
- `pthread_mutexattr_init()` initializes a mutex with default attributes; *undefined behavior* results when applying this function to an already initialized mutex
- One may set a mutex attribute to one of four types, which determines what happens when a thread locks a mutex it already owns:

PTHREAD_MUTEX_NORMAL suspends the thread forever, because the mutex is never unlocked by the owner thread itself.

Sync: Mutex Attributes

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init      (pthread_mutexattr_t *attr);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- pthread_mutexattr_destroy() releases any resources held by a mutex attribute. Using it after destroying it is *undefined behavior*.
- pthread_mutexattr_init() initializes a mutex with default attributes; *undefined behavior* results when applying this function to an already initialized mutex
- One may set a mutex attribute to one of four types, which determines what happens when a thread locks a mutex it already owns:

PTHREAD_MUTEX_NORMAL suspends the thread forever, because the mutex is never unlocked by the owner thread itself.

PTHREAD_MUTEX_RECURSIVE records the qty of times the owning thread locks the mutex. The thread must unlock it that many times to release the mutex.

Sync: Mutex Attributes

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init    (pthread_mutexattr_t *attr);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- pthread_mutexattr_destroy() releases any resources held by a mutex attribute. Using it after destroying it is *undefined behavior*.
- pthread_mutexattr_init() initializes a mutex with default attributes; *undefined behavior* results when applying this function to an already initialized mutex
- One may set a mutex attribute to one of four types, which determines what happens when a thread locks a mutex it already owns:

PTHREAD_MUTEX_NORMAL suspends the thread forever, because the mutex is never unlocked by the owner thread itself.

PTHREAD_MUTEX_RECURSIVE records the qty of times the owning thread locks the mutex. The thread must unlock it that many times to release the mutex.

PTHREAD_MUTEX_ERRORCHECK returns -1 with errno=EDEADLCK

Sync: Mutex Attributes

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init     (pthread_mutexattr_t *attr);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- pthread_mutexattr_destroy() releases any resources held by a mutex attribute. Using it after destroying it is *undefined behavior*.
- pthread_mutexattr_init() initializes a mutex with default attributes; *undefined behavior* results when applying this function to an already initialized mutex
- One may set a mutex attribute to one of four types, which determines what happens when a thread locks a mutex it already owns:

PTHREAD_MUTEX_NORMAL suspends the thread forever, because the mutex is never unlocked by the owner thread itself.

PTHREAD_MUTEX_RECURSIVE records the qty of times the owning thread locks the mutex. The thread must unlock it that many times to release the mutex.

PTHREAD_MUTEX_ERRORCHECK returns -1 with errno=EDEADLCK

PTHREAD_MUTEX_DEFAULT (default) results in *undefined behavior*. Attempting to unlock an unlocked mutex of this type also results in *undefined behavior*. An implementation may have PTHREAD_MUTEX_DEFAULT map to one of the other three types.

Sync: Deadlock

- Like semaphores, threads can produce deadlocks via mutexes: (aka **deadly embrace**)

Sync: Deadlock

- Like semaphores, threads can produce deadlocks via mutexes: (aka **deadly embrace**)

Process *a*	Locks resource A

Sync: Deadlock

- Like semaphores, threads can produce deadlocks via mutexes: (aka **deadly embrace**)

Process *a*	Locks resource A
Process *b*	Locks resource B

Sync: Deadlock

- Like semaphores, threads can produce deadlocks via mutexes: (aka **deadly embrace**)

Process *a*	Locks resource A
Process *b*	Locks resource B
Process *a*	tries to lock resource B, fails and blocks

Sync: Deadlock

- Like semaphores, threads can produce deadlocks via mutexes: (aka **deadly embrace**)

Process *a*	Locks resource A
Process *b*	Locks resource B
Process *a*	tries to lock resource B, fails and blocks
Process *b*	tries to lock resource A, fails and blocks

Sync: Deadlock

- Like semaphores, threads can produce deadlocks via mutexes: (aka **deadly embrace**)

Process *a*	Locks resource A
Process *b*	Locks resource B
Process *a*	tries to lock resource B, fails and blocks
Process *b*	tries to lock resource A, fails and blocks

- One way to prevent these deadlocks is to require that multiple locks must always be acquired in the same sequence by all threads

Sync: Deadlock

- Like semaphores, threads can produce deadlocks via mutexes: (aka **deadly embrace**)

Process *a*	Locks resource A
Process *b*	Locks resource B
Process *a*	tries to lock resource B, fails and blocks
Process *b*	tries to lock resource A, fails and blocks

- One way to prevent these deadlocks is to require that multiple locks must always be acquired in the same sequence by all threads
- Then, Process *b* would have tried to lock resource A first, and would have failed because *a* held the lock

Sync: Priority Inversion

Process A has high priority

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Process *C* runs, locks resource R

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Process *C* runs, locks resource R

A starts, blocks on “wait for R”

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Process *C* runs, locks resource R

A starts, blocks on “wait for R”

B starts, has higher priority than *C*, hogs the cpu

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Process *C* runs, locks resource R

A starts, blocks on “wait for R”

B starts, has higher priority than *C*, hogs the cpu

Thus *A* effectively ends up as a “lower priority” process than *B*!

Sync: Priority Inversion

Process A has high priority

Process B has medium priority

Process C has low priority

R is a resource

Process C runs, locks resource R

A starts, blocks on “wait for R”

B starts, has higher priority than C, hogs the cpu

Thus A effectively ends up as a “lower priority” process than B!

This situation actually happened to the Mars Pathfinder, with

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Process *C* runs, locks resource R

A starts, blocks on “wait for R”

B starts, has higher priority than *C*, hogs the cpu

Thus *A* effectively ends up as a “lower priority” process than *B*!

This situation actually happened to the Mars Pathfinder, with

A bus management; it moved data along the bus

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Process *C* runs, locks resource R

A starts, blocks on "wait for R"

B starts, has higher priority than *C*, hogs the cpu

Thus *A* effectively ends up as a "lower priority" process than *B*!

This situation actually happened to the Mars Pathfinder, with

A	bus management; it moved data along the bus
B	a communications task

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Process *C* runs, locks resource R

A starts, blocks on “wait for R”

B starts, has higher priority than *C*, hogs the cpu

Thus *A* effectively ends up as a “lower priority” process than *B*!

This situation actually happened to the Mars Pathfinder, with

A	bus management; it moved data along the bus
B	a communications task
C	a meteorological task

Sync: Priority Inversion

Process A has high priority

Process B has medium priority

Process C has low priority

R is a resource

Process C runs, locks resource R

A starts, blocks on “wait for R”

B starts, has higher priority than C, hogs the cpu

Thus A effectively ends up as a “lower priority” process than B!

This situation actually happened to the Mars Pathfinder, with

A	bus management; it moved data along the bus
B	a communications task
C	a meteorological task
R	the bus

Sync: Priority Inversion

Process *A* has high priority

Process *B* has medium priority

Process *C* has low priority

R is a resource

Process *C* runs, locks resource R

A starts, blocks on "wait for R"

B starts, has higher priority than *C*, hogs the cpu

Thus *A* effectively ends up as a "lower priority" process than *B*!

This situation actually happened to the Mars Pathfinder, with

A	bus management; it moved data along the bus
B	a communications task
C	a meteorological task
R	the bus

Mars Pathfinder's o/s had a mutex with a boolean parameter (attribute) supporting **priority inheritance**. It was off; when on, the process *C* would have inherited the priority of *A* when *A* blocked on the mutex. Simple fix; but painful to find.

Sync: Avoiding Priority Inversion

#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

- **Priority inheritance** allows a mutex to elevate the priority of its owner to that of the highest priority waiting thread

Sync: Avoiding Priority Inversion

```
#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

- **Priority inheritance** allows a mutex to elevate the priority of its owner to that of the highest priority waiting thread
- This feature is available if `_POSIX_THREAD_PRIO_INHERIT` is defined

Sync: Avoiding Priority Inversion

#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

- **Priority inheritance** allows a mutex to elevate the priority of its owner to that of the highest priority waiting thread
- This feature is available if _POSIX_THREAD_PRIO_INHERIT is defined
- Procedure:

Sync: Avoiding Priority Inversion

#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

- **Priority inheritance** allows a mutex to elevate the priority of its owner to that of the highest priority waiting thread
- This feature is available if `_POSIX_THREAD_PRIO_INHERIT` is defined
- Procedure:
  1. initialize mutex attribute

Sync: Avoiding Priority Inversion

#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

- **Priority inheritance** allows a mutex to elevate the priority of its owner to that of the highest priority waiting thread
- This feature is available if _POSIX_THREAD_PRIO_INHERIT is defined
- Procedure:
  1. initialize mutex attribute
  2. use pthread_mutexattr_getprotocol(); if its not _POSIX_THREAD_PRIO_INHERIT,

Sync: Avoiding Priority Inversion

```
#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict
protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

- **Priority inheritance** allows a mutex to elevate the priority of its owner to that of the highest priority waiting thread
- This feature is available if _POSIX_THREAD_PRIO_INHERIT is defined
- Procedure:
  1. initialize mutex attribute
  2. use pthread_mutexattr_getprotocol(); if its not _POSIX_THREAD_PRIO_INHERIT,
  3. then use pthread_mutexattr_setprotocol() with _POSIX_THREAD_PRIO_INHERIT

Sync: Avoiding Priority Inversion

#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

- **Priority inheritance** allows a mutex to elevate the priority of its owner to that of the highest priority waiting thread
- This feature is available if _POSIX_THREAD_PRIO_INHERIT is defined
- Procedure:
  1. initialize mutex attribute
  2. use pthread_mutexattr_getprotocol(); if its not _POSIX_THREAD_PRIO_INHERIT,
  3. then use pthread_mutexattr_setprotocol() with _POSIX_THREAD_PRIO_INHERIT
  4. Initialize the mutex with attr via pthread_mutex_init()

Sync: Avoiding Priority Inversion

#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

- **Priority inheritance** allows a mutex to elevate the priority of its owner to that of the highest priority waiting thread
- This feature is available if _POSIX_THREAD_PRIO_INHERIT is defined
- Procedure:
  1. initialize mutex attribute
  2. use pthread_mutexattr_getprotocol(); if its not _POSIX_THREAD_PRIO_INHERIT,
  3. then use pthread_mutexattr_setprotocol() with _POSIX_THREAD_PRIO_INHERIT
  4. Initialize the mutex with attr via pthread_mutex_init()

(see thrdprioinherit.c)

Sync: Avoiding Priority Inversion, con't.

- Available protocols:

PTHREAD_PRIO_NONE a thread's priority and scheduling will not be affected by its mutex ownership

Sync: Avoiding Priority Inversion, con't.

- Available protocols:

  PTHREAD_PRIO_NONE a thread's priority and scheduling will not be affected by its mutex ownership

  PTHREAD_PRIO_INHERIT higher priority threads elevate the mutex owning thread's priority. This elevation does not move the owning thread to the back of the (new) priority queue.

Sync: Avoiding Priority Inversion, con't.

- Available protocols:

  PTHREAD_PRIO_NONE a thread's priority and scheduling will not be affected by its mutex ownership

  PTHREAD_PRIO_INHERIT higher priority threads elevate the mutex owning thread's priority. This elevation does not move the owning thread to the back of the (new) priority queue.

  PTHREAD_PRIO_PROTECT owning thread will execute at the highest priority of all threads initialized with this mutex attribute

Sync: Avoiding Priority Inversion, con't.

- Available protocols:

  PTHREAD_PRIO_NONE a thread's priority and scheduling will not be affected by its mutex ownership

  PTHREAD_PRIO_INHERIT higher priority threads elevate the mutex owning thread's priority. This elevation does not move the owning thread to the back of the (new) priority queue.

  PTHREAD_PRIO_PROTECT owning thread will execute at the highest priority of all threads initialized with this mutex attribute

- Alternative: just don't use mutexes with threads of differing priorities.

Sync: Avoiding Priority Inversion, con't.

- Available protocols:

  PTHREAD_PRIO_NONE a thread's priority and scheduling will not be affected by its mutex ownership

  PTHREAD_PRIO_INHERIT higher priority threads elevate the mutex owning thread's priority. This elevation does not move the owning thread to the back of the (new) priority queue.

  PTHREAD_PRIO_PROTECT owning thread will execute at the highest priority of all threads initialized with this mutex attribute

- Alternative: just don't use mutexes with threads of differing priorities.

- Note that under Linux:

Sync: Avoiding Priority Inversion, con't.

- Available protocols:

  PTHREAD_PRIO_NONE a thread's priority and scheduling will not be affected by its mutex ownership

  PTHREAD_PRIO_INHERIT higher priority threads elevate the mutex owning thread's priority. This elevation does not move the owning thread to the back of the (new) priority queue.

  PTHREAD_PRIO_PROTECT owning thread will execute at the highest priority of all threads initialized with this mutex attribute

- Alternative: just don't use mutexes with threads of differing priorities.
- Note that under Linux:
  - Protocols are named _POSIX_THREAD_PRIO_INHERIT and _POSIX_THREAD_PRIO_PROTECT

Sync: Avoiding Priority Inversion, con't.

- Available protocols:

  PTHREAD_PRIO_NONE a thread's priority and scheduling will not be affected by its mutex ownership

  PTHREAD_PRIO_INHERIT higher priority threads elevate the mutex owning thread's priority. This elevation does not move the owning thread to the back of the (new) priority queue.

  PTHREAD_PRIO_PROTECT owning thread will execute at the highest priority of all threads initialized with this mutex attribute

- Alternative: just don't use mutexes with threads of differing priorities.
- Note that under Linux:
  - Protocols are named _POSIX_THREAD_PRIO_INHERIT and _POSIX_THREAD_PRIO_PROTECT
  - There is no _POSIX_THREAD_PRIO_NONE (use zero for this)

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition
  3. If the condition is false, call pthread_cond_wait() (ie. block on a c.v.). This also releases the mutex.

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition
  3. If the condition is false, call pthread_cond_wait() (ie. block on a c.v.). This also releases the mutex.
  4. Re-test the condition. If ok, unlock the mutex. (mutex is locked on wake-up and being first-to-run)

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition
  3. If the condition is false, call `pthread_cond_wait()` (ie. block on a c.v.). This also releases the mutex.
  4. Re-test the condition. If ok, unlock the mutex. (mutex is locked on wake-up and being first-to-run)
  5. Otherwise goto step 3

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition
  3. If the condition is false, call pthread_cond_wait() (ie. block on a c.v.). This also releases the mutex.
  4. Re-test the condition. If ok, unlock the mutex. (mutex is locked on wake-up and being first-to-run)
  5. Otherwise goto step 3
- Some other thread(s):

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition
  3. If the condition is false, call pthread_cond_wait() (ie. block on a c.v.). This also releases the mutex.
  4. Re-test the condition. If ok, unlock the mutex. (mutex is locked on wake-up and being first-to-run)
  5. Otherwise goto step 3
- Some other thread(s):
  1. Lock the mutex (perhaps to do work and modify data)

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition
  3. If the condition is false, call pthread_cond_wait() (ie. block on a c.v.). This also releases the mutex.
  4. Re-test the condition. If ok, unlock the mutex. (mutex is locked on wake-up and being first-to-run)
  5. Otherwise goto step 3
- Some other thread(s):
  1. Lock the mutex (perhaps to do work and modify data)
  2. Test for condition

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition
  3. If the condition is false, call pthread_cond_wait() (ie. block on a c.v.). This also releases the mutex.
  4. Re-test the condition. If ok, unlock the mutex. (mutex is locked on wake-up and being first-to-run)
  5. Otherwise goto step 3
- Some other thread(s):
  1. Lock the mutex (perhaps to do work and modify data)
  2. Test for condition
  3. If true, signal all threads waiting on the c.v.
     This will wake up all threads waiting on the c.v.

Sync: Condition Variables

- A **condition variable** lets threads synchronize on a condition, which often involves the state/values of data
- Setup:
  1. Lock a mutex
  2. Test for a condition
  3. If the condition is false, call pthread_cond_wait() (ie. block on a c.v.). This also releases the mutex.
  4. Re-test the condition. If ok, unlock the mutex. (mutex is locked on wake-up and being first-to-run)
  5. Otherwise goto step 3
- Some other thread(s):
  1. Lock the mutex (perhaps to do work and modify data)
  2. Test for condition
  3. If true, signal all threads waiting on the c.v.
     This will wake up all threads waiting on the c.v.
- In case of a broadcast, the desired condition may no longer hold due to some other c.v.-waiting thread's data modification(s)

Sync: Initializing a Condition Variable

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

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

int pthread_cond_destroy(pthread_cond_t *cond);

- Use the first method for a default c.v.

Sync: Initializing a Condition Variable

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);

- Use the first method for a default c.v.
- Use the second method if you want a c.v. with specific attributes

Sync: Initializing a Condition Variable

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);

- Use the first method for a default c.v.
- Use the second method if you want a c.v. with specific attributes
- The two functions initialize or destroy a c.v.

Sync: Initializing a Condition Variable

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);

- Use the first method for a default c.v.
- Use the second method if you want a c.v. with specific attributes
- The two functions initialize or destroy a c.v.
- A destroyed c.v. may be re-initialized using pthread_cond_init().

Sync: Initializing a Condition Variable

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);

- Use the first method for a default c.v.
- Use the second method if you want a c.v. with specific attributes
- The two functions initialize or destroy a c.v.
- A destroyed c.v. may be re-initialized using pthread_cond_init().
- Don't use copies of cv in calls to pthread_cond_wait(), etc; use only the cv itself.

  (behavior is otherwise undefined)

Sync: Waiting for a Condition Variable

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

- If the condition one wants (say, a counter having reached some value) does not hold, one may call either of the two functions above to wait for a c.v. signal

Sync: Waiting for a Condition Variable

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

- If the condition one wants (say, a counter having reached some value) does not hold, one may call either of the two functions above to wait for a c.v. signal
- The thread will block upon calling one of these functions.

Sync: Waiting for a Condition Variable

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

- If the condition one wants (say, a counter having reached some value) does not hold, one may call either of the two functions above to wait for a c.v. signal
- The thread will block upon calling one of these functions.
- The thread must hold the mutex mtx in a locked state.

Sync: Waiting for a Condition Variable

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

- If the condition one wants (say, a counter having reached some value) does not hold, one may call either of the two functions above to wait for a c.v. signal
- The thread will block upon calling one of these functions.
- The thread must hold the mutex mtx in a locked state.
- These calls will atomically release the mutex and cause the calling thread to block on the c.v.; when the thread is re-started, it will again hold the mutex in a locked state.

Sync: Waiting for a Condition Variable

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

- If the condition one wants (say, a counter having reached some value) does not hold, one may call either of the two functions above to wait for a c.v. signal
- The thread will block upon calling one of these functions.
- The thread must hold the mutex mtx in a locked state.
- These calls will atomically release the mutex and cause the calling thread to block on the c.v.; when the thread is re-started, it will again hold the mutex in a locked state.
- The abstime is an *absolute time*:

Sync: Waiting for a Condition Variable

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

- If the condition one wants (say, a counter having reached some value) does not hold, one may call either of the two functions above to wait for a c.v. signal
- The thread will block upon calling one of these functions.
- The thread must hold the mutex mtx in a locked state.
- These calls will atomically release the mutex and cause the calling thread to block on the c.v.; when the thread is re-started, it will again hold the mutex in a locked state.
- The abstime is an *absolute time*:

```
struct timespec {
        time_t tv_sec; // seconds
        long tv_nsec; // nano-seconds
        }
```

Sync: Waiting for a Condition Variable

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

- If the condition one wants (say, a counter having reached some value) does not hold, one may call either of the two functions above to wait for a c.v. signal
- The thread will block upon calling one of these functions.
- The thread must hold the mutex mtx in a locked state.
- These calls will atomically release the mutex and cause the calling thread to block on the c.v.; when the thread is re-started, it will again hold the mutex in a locked state.
- The abstime is an *absolute time*:

```
struct timespec {
        time_t tv_sec; // seconds
        long tv_nsec; // nano-seconds
        }
```

(see thrdcond.c)

Sync: Signalling via Condition Variable

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_signal(pthread_cond_t *cond);

- The scheduling policy controls the order in which threads are unblocked

Sync: Signalling via Condition Variable

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);

- The scheduling policy controls the order in which threads are unblocked
- Multiple awakened threads will contend for the mutex according to scheduling policy and priority

Sync: Signalling via Condition Variable

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

- The scheduling policy controls the order in which threads are unblocked
- Multiple awakened threads will contend for the mutex according to scheduling policy and priority
- Mutexes and condition variables may not be released from within an asynchronous signal handler

Sync: Signalling via Condition Variable

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);

- The scheduling policy controls the order in which threads are unblocked
- Multiple awakened threads will contend for the mutex according to scheduling policy and priority
- Mutexes and condition variables may not be released from within an asynchronous signal handler
- These functions unblock threads which are blocked on a c.v.

Sync: Signalling via Condition Variable

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);

- The scheduling policy controls the order in which threads are unblocked
- Multiple awakened threads will contend for the mutex according to scheduling policy and priority
- Mutexes and condition variables may not be released from within an asynchronous signal handler
- These functions unblock threads which are blocked on a c.v.
- The pthread_cond_broadcast() function unblocks *all* threads currently blocked on the c.v.

Sync: Signalling via Condition Variable

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);

- The scheduling policy controls the order in which threads are unblocked
- Multiple awakened threads will contend for the mutex according to scheduling policy and priority
- Mutexes and condition variables may not be released from within an asynchronous signal handler
- These functions unblock threads which are blocked on a c.v.
- The `pthread_cond_broadcast()` function unblocks *all* threads currently blocked on the c.v.
- The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on the c.v. (if any)

Sync: Signalling via Condition Variable

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);

- The scheduling policy controls the order in which threads are unblocked
- Multiple awakened threads will contend for the mutex according to scheduling policy and priority
- Mutexes and condition variables may not be released from within an asynchronous signal handler
- These functions unblock threads which are blocked on a c.v.
- The `pthread_cond_broadcast()` function unblocks *all* threads currently blocked on the c.v.
- The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on the c.v. (if any)
- These two functions have no effect if no threads are blocked waiting for the c.v.

Sync: Reader/Writer Locks

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);

- Assume that some data may be read by multiple threads. Readers happily coexist. However, a writer thread needs exclusive access.

Sync: Reader/Writer Locks

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);

- Assume that some data may be read by multiple threads. Readers happily coexist. However, a writer thread needs exclusive access.
- A reader, then, wants to set a read lock that allows other readers read access, but blocks writers

Sync: Reader/Writer Locks

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);

- Assume that some data may be read by multiple threads. Readers happily coexist. However, a writer thread needs exclusive access.
- A reader, then, wants to set a read lock that allows other readers read access, but blocks writers
- Writers will wait until they can get exclusive access

Sync: Reader/Writer Locks

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);

- Assume that some data may be read by multiple threads. Readers happily coexist. However, a writer thread needs exclusive access.
- A reader, then, wants to set a read lock that allows other readers read access, but blocks writers
- Writers will wait until they can get exclusive access
- `pthread_rwlock_init()` allocates resources required for the rwlock, initializing it to an unlocked state (with `attr` attributes, if any)

Sync: Reader/Writer Locks

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);

- Assume that some data may be read by multiple threads. Readers happily coexist. However, a writer thread needs exclusive access.
- A reader, then, wants to set a read lock that allows other readers read access, but blocks writers
- Writers will wait until they can get exclusive access
- `pthread_rwlock_init()` allocates resources required for the rwlock, initializing it to an unlocked state (with `attr` attributes, if any)
- Read/write locks can be used any number of times without being re-initialized

Sync: Reader/Writer Locks

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);

- Assume that some data may be read by multiple threads. Readers happily coexist. However, a writer thread needs exclusive access.
- A reader, then, wants to set a read lock that allows other readers read access, but blocks writers
- Writers will wait until they can get exclusive access
- `pthread_rwlock_init()` allocates resources required for the rwlock, initializing it to an unlocked state (with `attr` attributes, if any)
- Read/write locks can be used any number of times without being re-initialized
- `pthread_rwlock_destroy()` destroys the read-write lock object, releasing its resources.

Sync: Reader Locking

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_rdlock() attempts to read-lock the rwlock. It'll usually succeed unless a writer holds a write lock on it, and there are no writer threads blocked on the lock.

Sync: Reader Locking

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_rdlock() attempts to read-lock the rwlock. It'll usually succeed unless a writer holds a write lock on it, and there are no writer threads blocked on the lock.
- Implementations may limit the number of simultaneous readers, so be sure to check for a -1 return value.

Sync: Reader Locking

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_rdlock() attempts to read-lock the rwlock. It'll usually succeed unless a writer holds a write lock on it, and there are no writer threads blocked on the lock.
- Implementations may limit the number of simultaneous readers, so be sure to check for a -1 return value.
- The calling thread may deadlock if it attempts to get a read lock when it already holds a write lock with rwlock

Sync: Reader Locking

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_rdlock() attempts to read-lock the rwlock. It'll usually succeed unless a writer holds a write lock on it, and there are no writer threads blocked on the lock.
- Implementations may limit the number of simultaneous readers, so be sure to check for a -1 return value.
- The calling thread may deadlock if it attempts to get a read lock when it already holds a write lock with rwlock
- Threads may hold multiple concurrent read locks on rwlock; it must perform matching unlocks to release the lock.

Sync: Reader Locking

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_rdlock() attempts to read-lock the rwlock. It'll usually succeed unless a writer holds a write lock on it, and there are no writer threads blocked on the lock.
- Implementations may limit the number of simultaneous readers, so be sure to check for a -1 return value.
- The calling thread may deadlock if it attempts to get a read lock when it already holds a write lock with rwlock
- Threads may hold multiple concurrent read locks on rwlock; it must perform matching unlocks to release the lock.
- pthread_rwlock_tryrdlock() either acquires the lock or fails, returning immediately. It will never block.

Sync: Reader Locking

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_rdlock() attempts to read-lock the rwlock. It'll usually succeed unless a writer holds a write lock on it, and there are no writer threads blocked on the lock.
- Implementations may limit the number of simultaneous readers, so be sure to check for a -1 return value.
- The calling thread may deadlock if it attempts to get a read lock when it already holds a write lock with rwlock
- Threads may hold multiple concurrent read locks on rwlock; it must perform matching unlocks to release the lock.
- pthread_rwlock_tryrdlock() either acquires the lock or fails, returning immediately. It will never block.
- If a thread is blocked waiting for a read-write lock for reading, and it receives a signal, after its signal handler returns the thread will resume waiting.

Sync: Writer Locking

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_wrlock() attempts to write lock rwlock, and will acquire the write lock if no other thread (either reader or writer) already holds the lock.

Sync: Writer Locking

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_wrlock() attempts to write lock rwlock, and will acquire the write lock if no other thread (either reader or writer) already holds the lock.
- On failure to acquire the lock, pthread_rwlock_wrlock() will block until it can acquire the lock.

Sync: Writer Locking

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_wrlock() attempts to write lock rwlock, and will acquire the write lock if no other thread (either reader or writer) already holds the lock.
- On failure to acquire the lock, pthread_rwlock_wrlock() will block until it can acquire the lock.
- The calling thread may deadlock if it already holds the read-write lock (whether as a read lock or as a write lock)

Sync: Writer Locking

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_wrlock() attempts to write lock rwlock, and will acquire the write lock if no other thread (either reader or writer) already holds the lock.
- On failure to acquire the lock, pthread_rwlock_wrlock() will block until it can acquire the lock.
- The calling thread may deadlock if it already holds the read-write lock (whether as a read lock or as a write lock)
- pthread_rwlock_trywrlock() either succeeds at acquiring a write lock or fails and returns immediately, but does not block

Sync: Writer Locking

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_wrlock() attempts to write lock rwlock, and will acquire the write lock if no other thread (either reader or writer) already holds the lock.
- On failure to acquire the lock, pthread_rwlock_wrlock() will block until it can acquire the lock.
- The calling thread may deadlock if it already holds the read-write lock (whether as a read lock or as a write lock)
- pthread_rwlock_trywrlock() either succeeds at acquiring a write lock or fails and returns immediately, but does not block
- Signal receipt by a thread blocked while attempting to acquire a write lock will resume waiting after its signal handler returns.

Sync: Releasing Read-Write Locks

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_unlock() releases any lock held on rwlock by the calling thread, whether it is a read or a write lock

Sync: Releasing Read-Write Locks

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_unlock() releases any lock held on rwlock by the calling thread, whether it is a read or a write lock
- Unlocking a rwlock not held by the calling thread is undefined behavior.

Sync: Releasing Read-Write Locks

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_unlock() releases any lock held on rwlock by the calling thread, whether it is a read or a write lock
- Unlocking a rwlock not held by the calling thread is undefined behavior.
- If there are other threads holding a read lock on rwlock, then the object remains in the read locked state

Sync: Releasing Read-Write Locks

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_unlock() releases any lock held on rwlock by the calling thread, whether it is a read or a write lock
- Unlocking a rwlock not held by the calling thread is undefined behavior.
- If there are other threads holding a read lock on rwlock, then the object remains in the read locked state
- Only when all read locks are released will rwlock be truly unlocked (with no owners)

Sync: Releasing Read-Write Locks

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

- pthread_rwlock_unlock() releases any lock held on rwlock by the calling thread, whether it is a read or a write lock
- Unlocking a rwlock not held by the calling thread is undefined behavior.
- If there are other threads holding a read lock on rwlock, then the object remains in the read locked state
- Only when all read locks are released will rwlock be truly unlocked (with no owners)

(see thrd_rdlock.c)

Sync: Reader/Writer Lock Attributes

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t * restrict attr, int *restrict pshared);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);

- Currently there is but one attribute required to be available for rwlocks:

Sync: Reader/Writer Lock Attributes

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t * restrict attr, int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);

- Currently there is but one attribute required to be available for rwlocks:

PTHREAD_PROCESS_PRIVATE only threads within the process which created the reader/writer lock may manipulate it.

Sync: Reader/Writer Lock Attributes

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t * restrict attr, int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);

- Currently there is but one attribute required to be available for rwlocks:

PTHREAD_PROCESS_PRIVATE only threads within the process which created the reader/writer lock may manipulate it.

PTHREAD_PROCESS_SHARED any process with access to the rwlock's memory may manipulate it

Sync: Reader/Writer Lock Attributes

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t * restrict attr, int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);

- Currently there is but one attribute required to be available for rwlocks:

  PTHREAD_PROCESS_PRIVATE only threads within the process which created the reader/writer lock may manipulate it.

  PTHREAD_PROCESS_SHARED any process with access to the rwlock's memory may manipulate it

- The SHARED feature is available only if the symbolic constant _POSIX_THREAD_PROCESS_SHARED is defined in unistd.h to be a value greater than zero.

Sync: Posix Semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
```

- These are also known as *unnamed semaphores*

Sync: Posix Semaphores

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

- These are also known as *unnamed semaphores*
- `sem_init()` initializes the semaphore with the given `value`.

Sync: Posix Semaphores

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

- These are also known as *unnamed semaphores*
- sem_init() initializes the semaphore with the given value.
- When pshared is 0: the semaphore is shared amongst the process's threads. sem should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)

Sync: Posix Semaphores

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

- These are also known as *unnamed semaphores*
- sem_init() initializes the semaphore with the given value.
- When pshared is 0: the semaphore is shared amongst the process's threads. sem should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)
- When pshared is 1: the semaphore is shared amongst processes, too

Sync: Posix Semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
```

- These are also known as *unnamed semaphores*
- `sem_init()` initializes the semaphore with the given `value`.
- When `pshared` is 0: the semaphore is shared amongst the process's threads. `sem` should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)
- When `pshared` is 1: the semaphore is shared amongst processes, too
- In the latter case, `sem` should point to shared memory (IPC) or `mmap()`'d memory

Sync: Posix Semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
```

- These are also known as *unnamed semaphores*
- sem_init() initializes the semaphore with the given value.
- When pshared is 0: the semaphore is shared amongst the process's threads. sem should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)
- When pshared is 1: the semaphore is shared amongst processes, too
- In the latter case, sem should point to shared memory (IPC) or mmap()'d memory
- Use sem_post(), sem_wait(), and/or sem_trywait() to work with unnamed semaphores

Sync: Posix Semaphores

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

- These are also known as *unnamed semaphores*
- sem_init() initializes the semaphore with the given value.
- When pshared is 0: the semaphore is shared amongst the process's threads. sem should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)
- When pshared is 1: the semaphore is shared amongst processes, too
- In the latter case, sem should point to shared memory (IPC) or mmap()'d memory
- Use sem_post(), sem_wait(), and/or sem_trywait() to work with unnamed semaphores
- Unnamed semaphores are also known as "Posix" semaphores

Sync: Posix Semaphores

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

- These are also known as *unnamed semaphores*
- sem_init() initializes the semaphore with the given value.
- When pshared is 0: the semaphore is shared amongst the process's threads. sem should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)
- When pshared is 1: the semaphore is shared amongst processes, too
- In the latter case, sem should point to shared memory (IPC) or mmap()'d memory
- Use sem_post(), sem_wait(), and/or sem_trywait() to work with unnamed semaphores
- Unnamed semaphores are also known as "Posix" semaphores
- Unnamed semaphores have **kernel persistence** *(semaphores will exist until the system is shut down or sem_unlink() is used to remove them)*

Sync: Posix Semaphores

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

- These are also known as *unnamed semaphores*
- sem_init() initializes the semaphore with the given value.
- When pshared is 0: the semaphore is shared amongst the process's threads. sem should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)
- When pshared is 1: the semaphore is shared amongst processes, too
- In the latter case, sem should point to shared memory (IPC) or mmap()'d memory
- Use sem_post(), sem_wait(), and/or sem_trywait() to work with unnamed semaphores
- Unnamed semaphores are also known as "Posix" semaphores
- Unnamed semaphores have **kernel persistence** *(semaphores will exist until the system is shut down or sem_unlink() is used to remove them)*
- sem_destroy() destroys (releases resources) held by the semaphore *sem

Sync: Posix Semaphores

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

- These are also known as *unnamed semaphores*
- sem_init() initializes the semaphore with the given value.
- When pshared is 0: the semaphore is shared amongst the process's threads. sem should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)
- When pshared is 1: the semaphore is shared amongst processes, too
- In the latter case, sem should point to shared memory (IPC) or mmap()'d memory
- Use sem_post(), sem_wait(), and/or sem_trywait() to work with unnamed semaphores
- Unnamed semaphores are also known as "Posix" semaphores
- Unnamed semaphores have **kernel persistence** *(semaphores will exist until the system is shut down or sem_unlink() is used to remove them)*
- sem_destroy() destroys (releases resources) held by the semaphore *sem
- Destroying a currently held semaphore produces undefined behavior.

Sync: Posix Semaphores

#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

- These are also known as *unnamed semaphores*
- sem_init() initializes the semaphore with the given value.
- When pshared is 0: the semaphore is shared amongst the process's threads. sem should point to memory visible to all threads (ie. global variables, dynamically allocated variables, etc)
- When pshared is 1: the semaphore is shared amongst processes, too
- In the latter case, sem should point to shared memory (IPC) or mmap()'d memory
- Use sem_post(), sem_wait(), and/or sem_trywait() to work with unnamed semaphores
- Unnamed semaphores are also known as "Posix" semaphores
- Unnamed semaphores have **kernel persistence** *(semaphores will exist until the system is shut down or sem_unlink() is used to remove them)*
- sem_destroy() destroys (releases resources) held by the semaphore *sem
- Destroying a currently held semaphore produces undefined behavior.
- Using a destroyed semaphore produces undefined behavior (unless it has been re-initialized)

Sync: Using Posix Semaphores

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_post(sem_t *sem);
```

- `sem_wait()` The thread/process will wait (block) until the semaphore's value becomes greater than or equal to one. It then atomically decrements the semaphore's value (ie. locks it) and unblocks.

Sync: Using Posix Semaphores

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_post(sem_t *sem);
```

- sem_wait() The thread/process will wait (block) until the semaphore's value becomes greater than or equal to one. It then atomically decrements the semaphore's value (ie. locks it) and unblocks.
- sem_trywait() Like sem_wait(), except that the function will not block when sem_wait() would have, instead returning -1/EAGAIN.

Sync: Using Posix Semaphores

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_post(sem_t *sem);
```

- sem_wait() The thread/process will wait (block) until the semaphore's value becomes greater than or equal to one. It then atomically decrements the semaphore's value (ie. locks it) and unblocks.
- sem_trywait() Like sem_wait(), except that the function will not block when sem_wait() would have, instead returning -1/EAGAIN.
- sem_timedwait() is like sem_wait(), but will also unblock when the absolute time specified by abs_timeout is reached.

Sync: Using Posix Semaphores

#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_post(sem_t *sem);

- sem_wait() The thread/process will wait (block) until the semaphore's value becomes greater than or equal to one. It then atomically decrements the semaphore's value (ie. locks it) and unblocks.
- sem_trywait() Like sem_wait(), except that the function will not block when sem_wait() would have, instead returning -1/EAGAIN.
- sem_timedwait() is like sem_wait(), but will also unblock when the absolute time specified by abs_timeout is reached.
- sem_post() increments (ie. unlocks) a semaphore *sem. Any thread or process blocked on this semaphore will then be woken up and will unlock the semaphore.

Sync: Using Posix Semaphores

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_post(sem_t *sem);
```

- sem_wait() The thread/process will wait (block) until the semaphore's value becomes greater than or equal to one. It then atomically decrements the semaphore's value (ie. locks it) and unblocks.
- sem_trywait() Like sem_wait(), except that the function will not block when sem_wait() would have, instead returning -1/EAGAIN.
- sem_timedwait() is like sem_wait(), but will also unblock when the absolute time specified by abs_timeout is reached.
- sem_post() increments (ie. unlocks) a semaphore *sem. Any thread or process blocked on this semaphore will then be woken up and will unlock the semaphore.
- Mutexes can be released *only by the owner thread*.

Sync: Using Posix Semaphores

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_post(sem_t *sem);
```

- sem_wait() The thread/process will wait (block) until the semaphore's value becomes greater than or equal to one. It then atomically decrements the semaphore's value (ie. locks it) and unblocks.
- sem_trywait() Like sem_wait(), except that the function will not block when sem_wait() would have, instead returning -1/EAGAIN.
- sem_timedwait() is like sem_wait(), but will also unblock when the absolute time specified by abs_timeout is reached.
- sem_post() increments (ie. unlocks) a semaphore *sem. Any thread or process blocked on this semaphore will then be woken up and will unlock the semaphore.
- Mutexes can be released *only by the owner thread*.
- Posix semaphores can be released *by any thread with access*.

see thrd_unnamedsem.c, thrd_timedwait.c

Sync: Posix Named Semaphores

```
#include <fcntl.h>      // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
```

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore

Sync: Posix Named Semaphores

```
#include <fcntl.h>     // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
```

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore
- A named semaphore's name begins with a "/" and is a null-terminated string of up to _PC_NAME_MAX+1 characters *(see seelimits.c)*

Sync: Posix Named Semaphores

```
#include <fcntl.h>     // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
```

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore
- A named semaphore's name begins with a "/" and is a null-terminated string of up to _PC_NAME_MAX+1 characters *(see seelimits.c)*
- Two processes or threads can operate on the same named semaphore by passing sem_open() the same name

Sync: Posix Named Semaphores

```
#include <fcntl.h>      // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
```

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore
- A named semaphore's name begins with a "/" and is a null-terminated string of up to _PC_NAME_MAX+1 characters *(see seelimits.c)*
- Two processes or threads can operate on the same named semaphore by passing sem_open() the same name
- Like unnamed semaphores, use sem_wait() and sem_post() calls to manipulate it

Sync: Posix Named Semaphores

```
#include <fcntl.h>     // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
```

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore
- A named semaphore's name begins with a "/" and is a null-terminated string of up to _PC_NAME_MAX+1 characters *(see seelimits.c)*
- Two processes or threads can operate on the same named semaphore by passing sem_open() the same name
- Like unnamed semaphores, use sem_wait() and sem_post() calls to manipulate it
- Use sem_close() to destroy a named semaphore

Sync: Posix Named Semaphores

```
#include <fcntl.h>      // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
```

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore
- A named semaphore's name begins with a “/” and is a null-terminated string of up to _PC_NAME_MAX+1 characters *(see seelimits.c)*
- Two processes or threads can operate on the same named semaphore by passing sem_open() the same name
- Like unnamed semaphores, use sem_wait() and sem_post() calls to manipulate it
- Use sem_close() to destroy a named semaphore
- All open named semaphores are automatically closed upon process termination

Sync: Posix Named Semaphores

```
#include <fcntl.h>      // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
```

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore
- A named semaphore's name begins with a "/" and is a null-terminated string of up to _PC_NAME_MAX+1 characters *(see seelimits.c)*
- Two processes or threads can operate on the same named semaphore by passing sem_open() the same name
- Like unnamed semaphores, use sem_wait() and sem_post() calls to manipulate it
- Use sem_close() to destroy a named semaphore
- All open named semaphores are automatically closed upon process termination
- oflag accepts values such as O_CREAT

Sync: Posix Named Semaphores

```
#include <fcntl.h>      // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
```

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore
- A named semaphore's name begins with a "/" and is a null-terminated string of up to _PC_NAME_MAX+1 characters *(see seelimits.c)*
- Two processes or threads can operate on the same named semaphore by passing sem_open() the same name
- Like unnamed semaphores, use sem_wait() and sem_post() calls to manipulate it
- Use sem_close() to destroy a named semaphore
- All open named semaphores are automatically closed upon process termination
- oflag accepts values such as O_CREAT
- mode specifies permission flags; both read and write permissions should be enabled for processes that can manipulate the named semaphore (S_IRWXU, S_IRWXG, S_IRWXO)

Sync: Posix Named Semaphores

#include <fcntl.h> // For O_* constants
#include <sys/stat.h> // For mode constants
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);
int sem_close(sem_t *sem);

- sem_open() creates a new Posix semaphore or opens a pre-existing semaphore
- A named semaphore's name begins with a "/" and is a null-terminated string of up to _PC_NAME_MAX+1 characters *(see seelimits.c)*
- Two processes or threads can operate on the same named semaphore by passing sem_open() the same name
- Like unnamed semaphores, use sem_wait() and sem_post() calls to manipulate it
- Use sem_close() to destroy a named semaphore
- All open named semaphores are automatically closed upon process termination
- oflag accepts values such as O_CREAT
- mode specifies permission flags; both read and write permissions should be enabled for processes that can manipulate the named semaphore (S_IRWXU, S_IRWXG, S_IRWXO)
- If the semaphore with a given name already exists, both oflag and mode will simply be ignored.