Processes, Threads & Synchronization

Introduction

You will learn:

- what a process is and what a thread is
- why you'd use multiple threads in a process
- how to create processes and threads and how to detect when they die
- how to synchronize among threads using mutexes, condvars, ...

Topics:

→ **Processes and Threads**

Processes

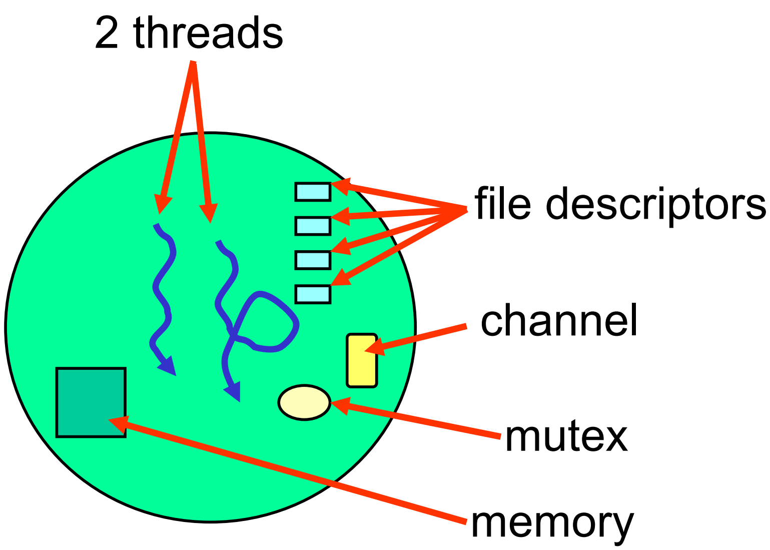
Threads

Synchronization

Conclusion

What is a process?

- a program loaded into memory
- identified by a process id, commonly abbreviated as **`pid`**
- owns resources:
  - memory, including code and data
  - open files
  - identity - user id, group id
  - timers
  - and more



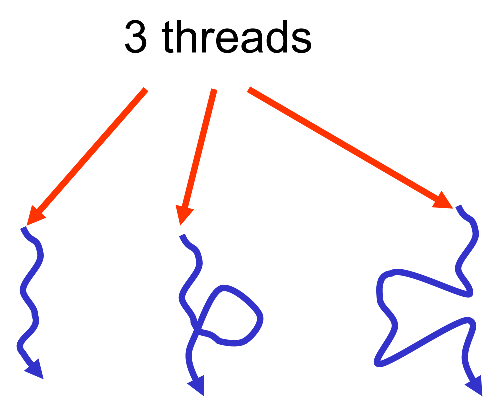
Resources owned by one process are protected from other processes

What is a thread?

– a thread is a single flow of execution or control

– a thread has some attributes:

- priority
- scheduling algorithm
- register set
- CPU mask for multicore
- signal mask
- and others

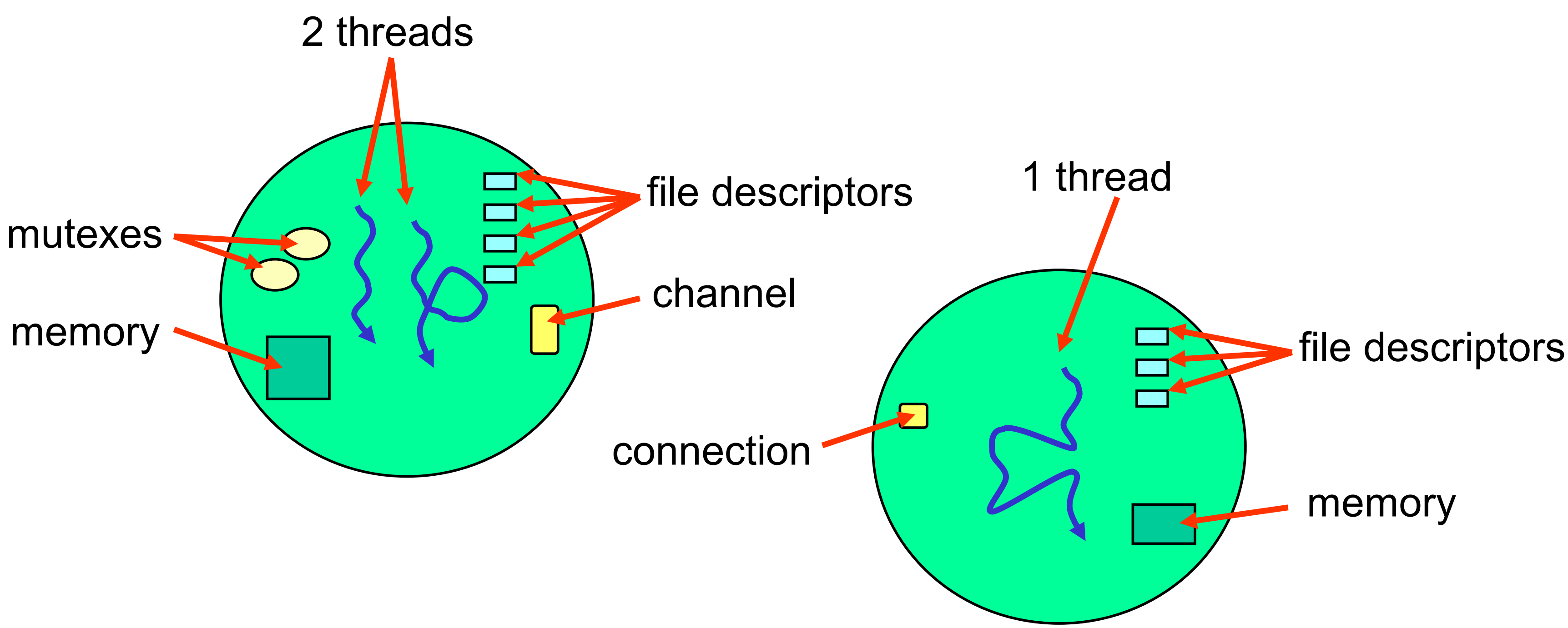


– all its attributes have to do with running code

Threads run in a process:

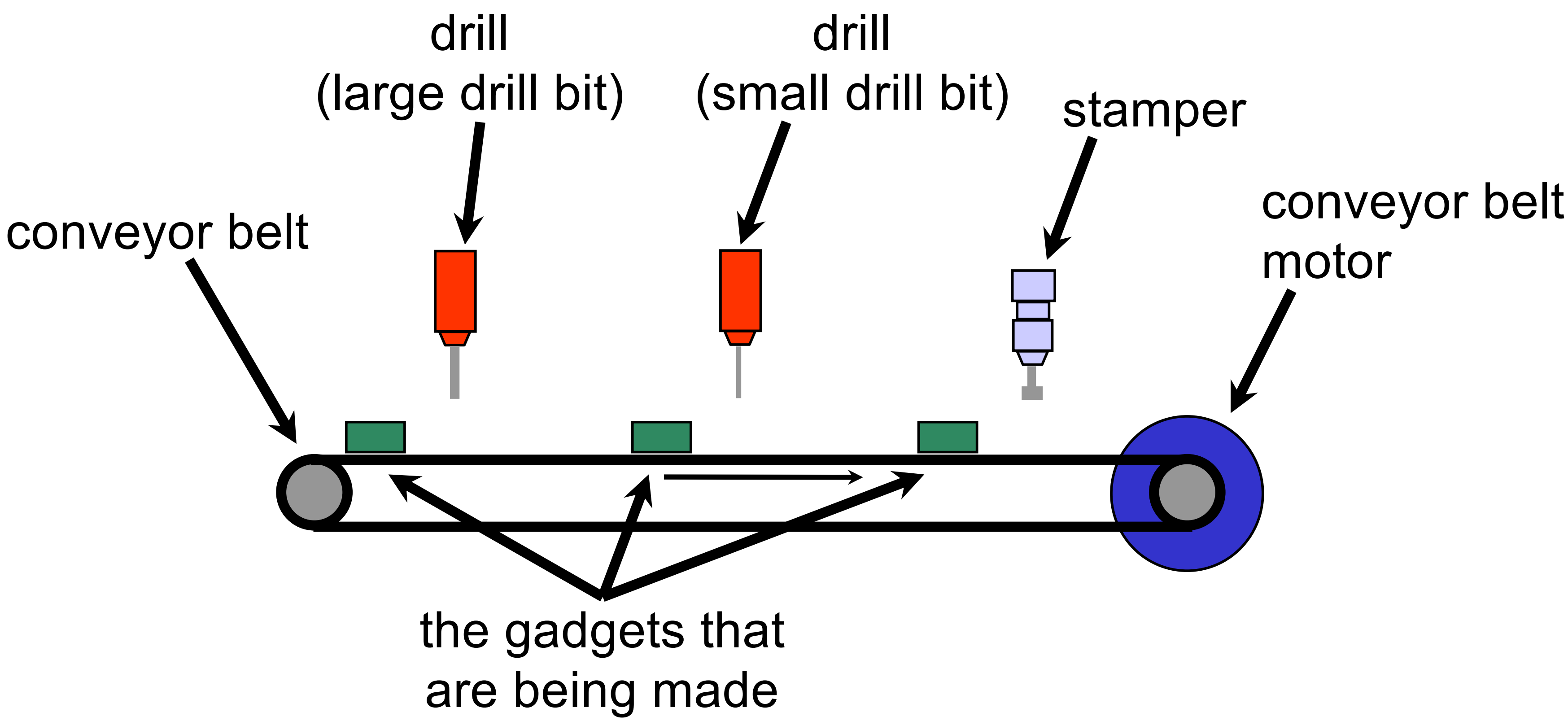
– a process must have at least one thread

– threads in a process share all the process resources



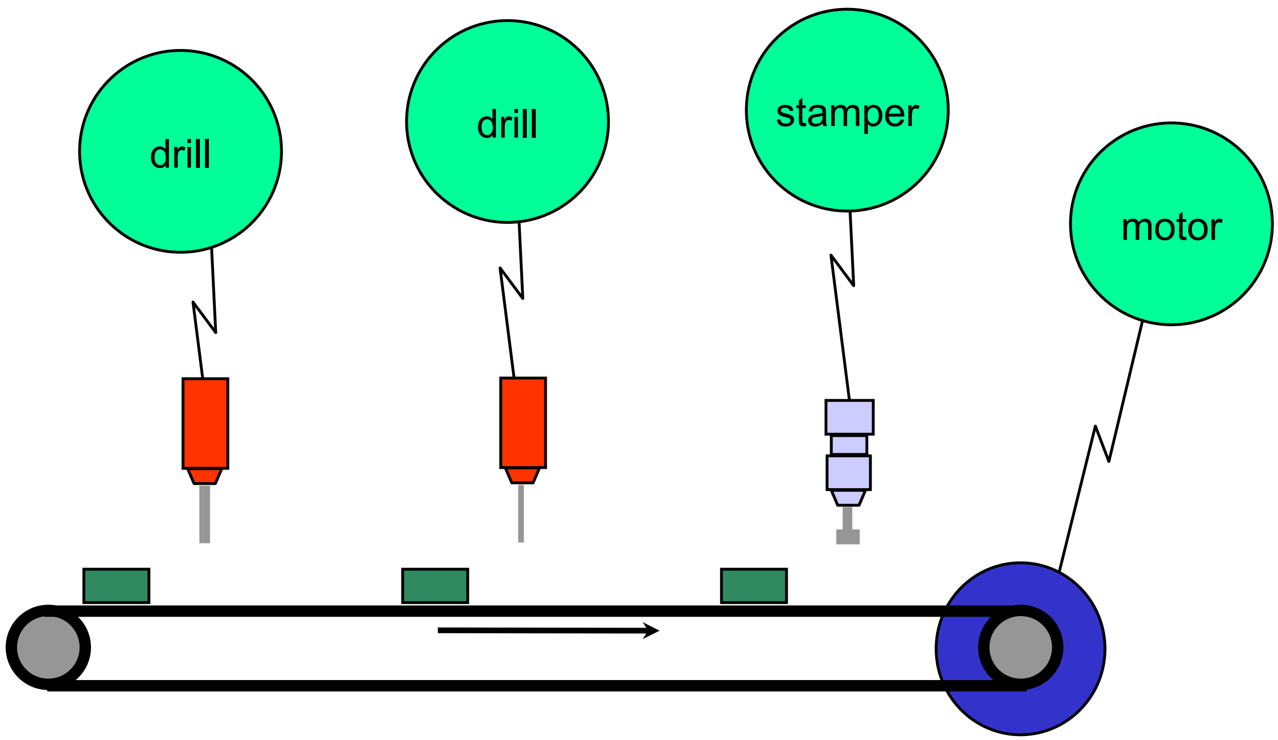
Threads run code, processes own resources

Example - Assembly line

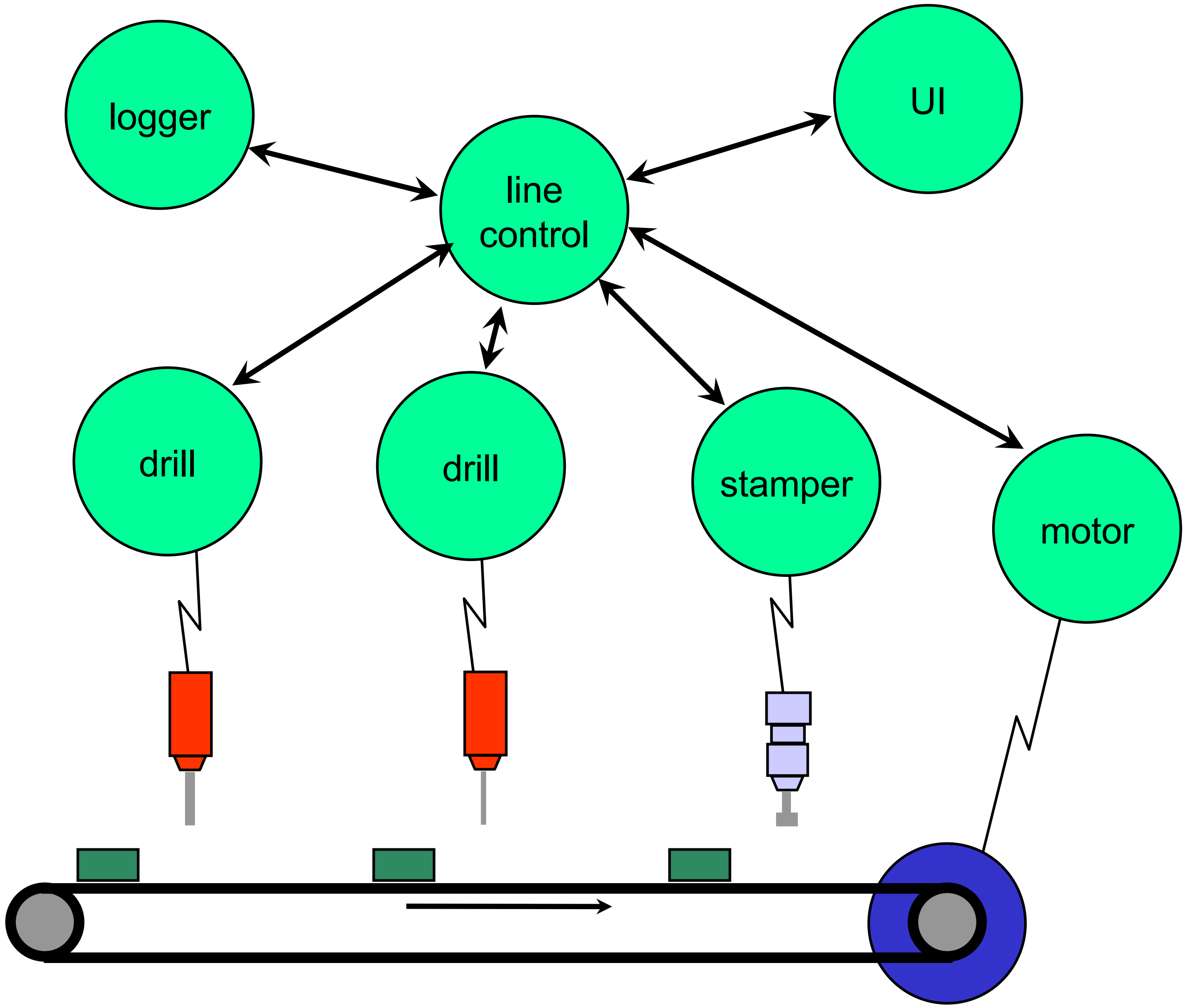


Processes

The processes that monitor and control these devices:



Of course, there will be more levels:

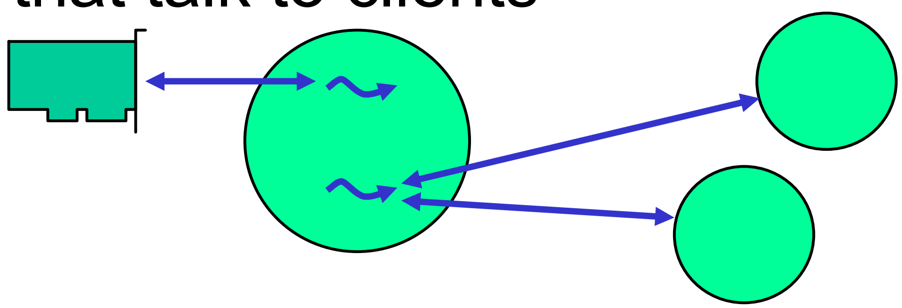


Process opacity:

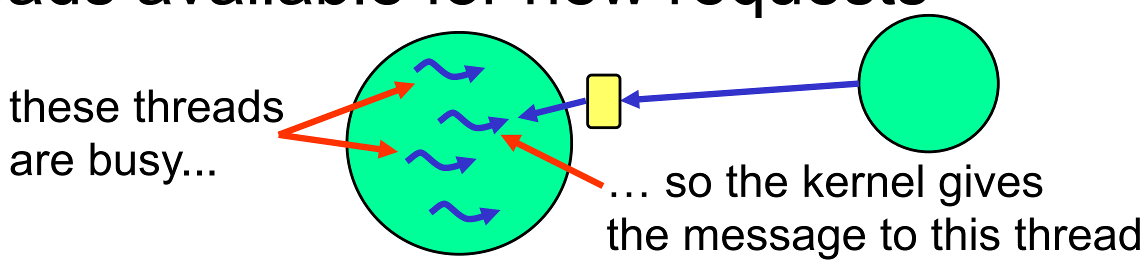
- one process should not be aware of the threads in another process
  - threads are an implementation detail of the process that they are in
- why?
  - object oriented design - the process is the object.
  - flexibility in how processes are written - it might use only one thread, it might use multiple threads, the threads may be dynamically created and destroyed as needed, …
  - scalability and configurability - if clients find servers using names then servers can be moved around. Intermediate servers can be added, servers can be put on other nodes of a network, server can be scaled up or down by adding or removing threads

Some examples of multithreaded processes:

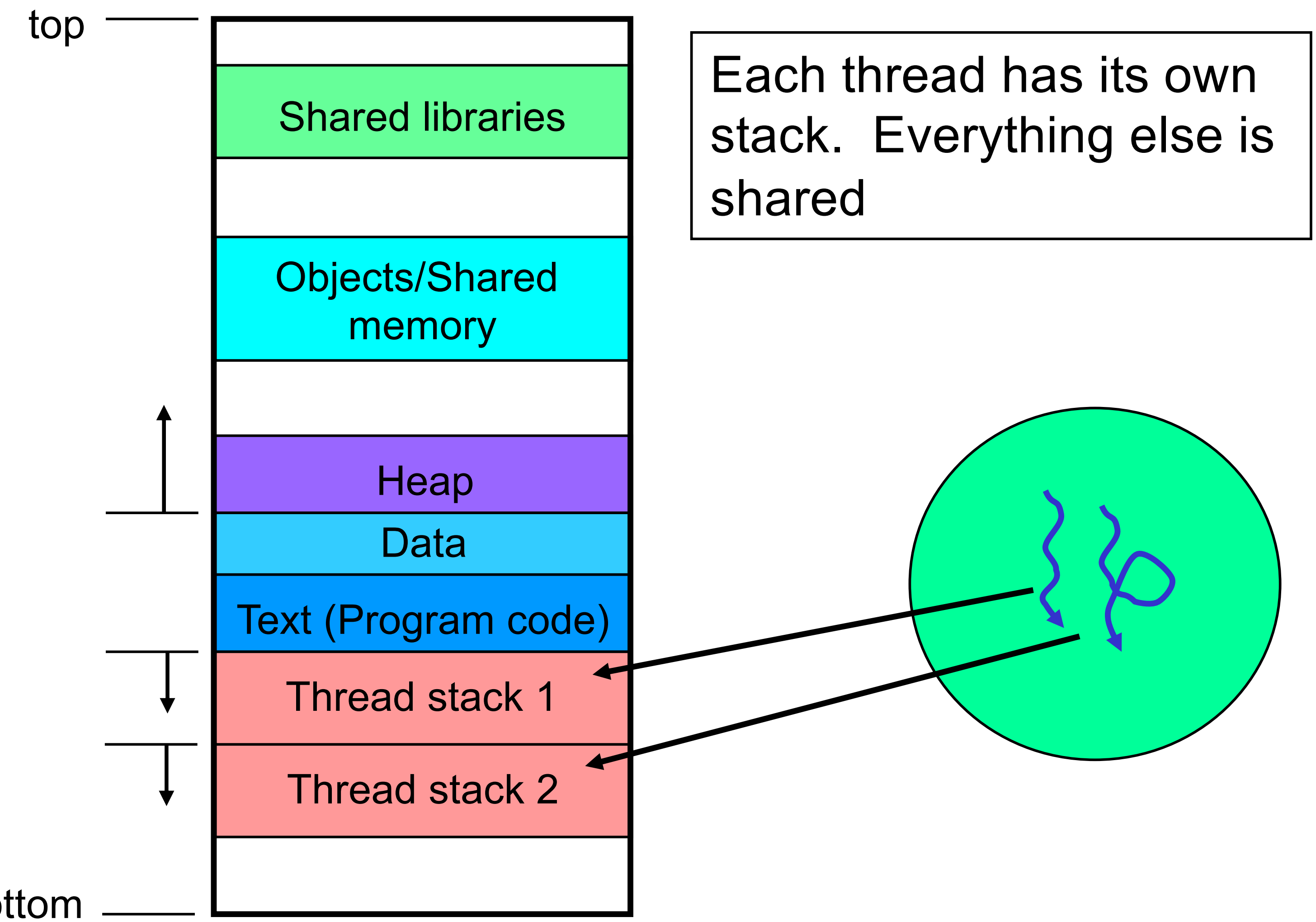
- high priority, time-critical thread dedicated to handling hardware requests as soon as they come in; other thread(s) that talk to clients



- pool of worker threads. If one or more threads are busy handling previous requests there are still other threads available for new requests

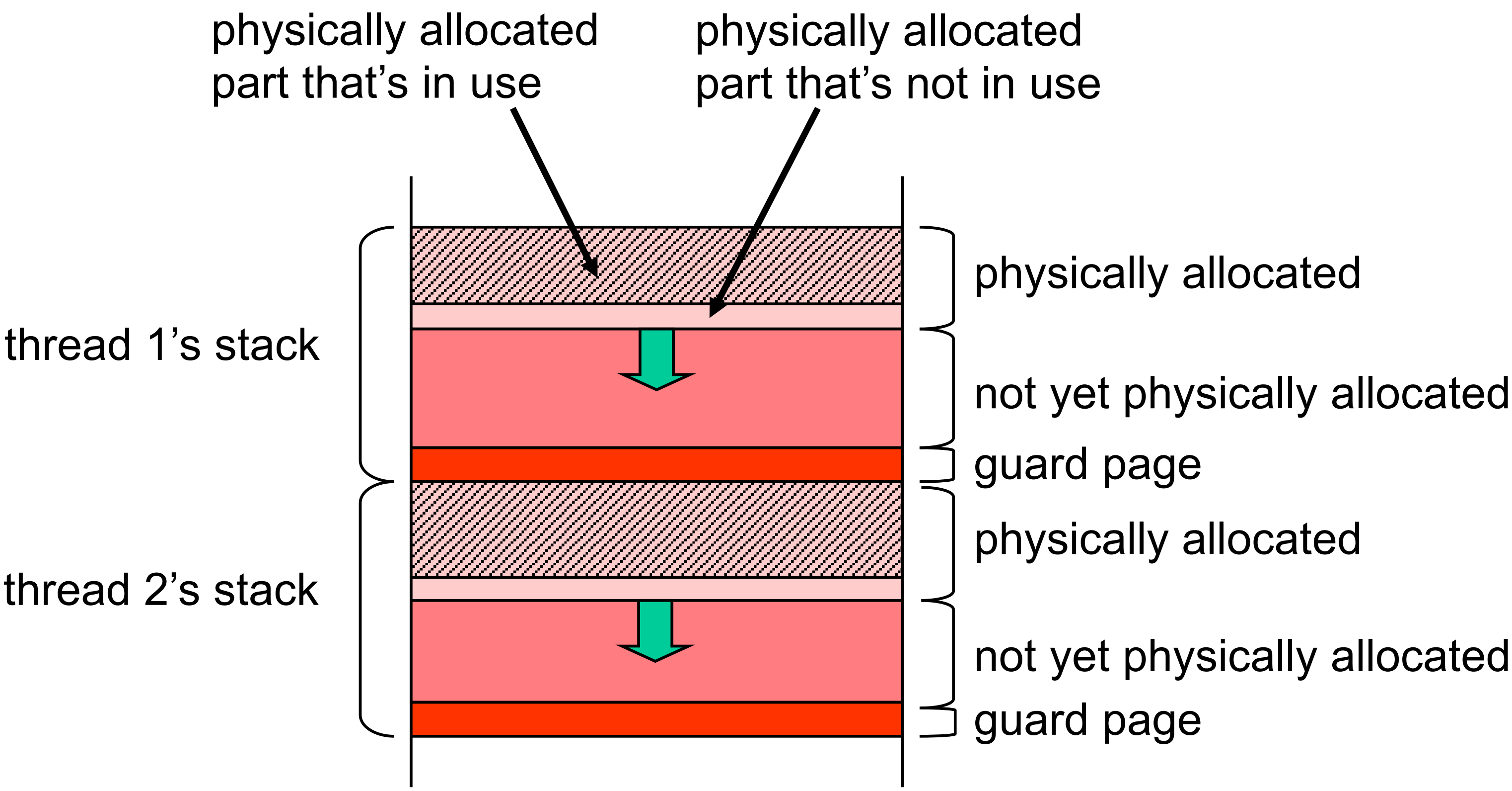


Virtual address space of a process without ASLR:



Thread stacks:

– each thread's stack has a maximum size but not all of it will necessarily be physically allocated



Topics:

Processes and Threads

Processes

- Creation
- Detecting termination

Threads

Synchronization

Conclusion

There are a number of process creation calls:

- *fork()*
  - create a copy of the calling process
- *exec*()*
  - load a program from storage to transform the calling process
- *posix_spawn()*, *spawn*()*
  - load a new program creating a new process for it

Process Creation - fork()

fork() will create a copy of your process:

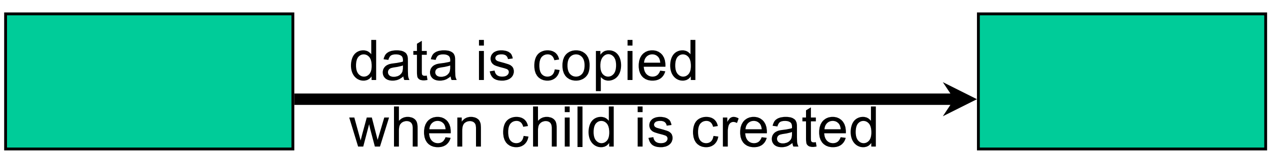
- the child will:
  - be an identical copy of the parent
  - start from the *fork()*
  - initially have the same data as the parent
- *fork()* in a multi-threaded process is best avoided
- *fork()* returns child's pid for the parent and 0 for the child

parent

```
pid = fork();
if (pid > 0) {
  // parent does this section
} else if (pid == 0 ) {
  // child does this section
} else {
  // error return to parent
}
```

child

```
pid = fork();
if (pid > 0) {
  // parent does this section
} else if (pid == 0 ) {
  // child does this section
} else {
  // error return to parent
}
```



Fork() - Inherited resources

What resources get inherited?

- inherited:
  - file descriptors (fds)
  - any thread attributes that inherit (e.g. priority, scheduling algorithm, signal mask, io privilege)
  - uid, gid, umask, process group, session
  - address space is replicated
- not inherited:
  - side channel connections (coids)
  - channels (chids)
  - timers

The *exec*()* family of functions replace the current process environment with a new program loaded from storage

- process id (`pid`) remains the same
- inheritance is mostly same as *fork()* except:
  - address space is created new
  - inheritance of file descriptors (fds) is configurable on a per fd basis
- arguments and environment variables may be passed to the new program
- these functions will not return unless an error occurs

To run a new program:

- use *posix_spawn()* or the *spawn*()* calls
  - will load and run a program in a new process
  - will return the pid of the child process
  - inheritance rules follow that of *fork()* and *exec*()*
- *spawn*()* are convenience functions
- *posix_spawn()* gives you more control
  - more complex to use
  - portable

Fork & exec vs spawn

- fork & exec is traditional Unix way
  - portable
  - inefficient
  - very complex to do safely in a multi-threaded process
- spawn does this as a single operation
  - avoids the copy of data segment,
  - avoids a lot of setup and initialization that will immediately get torn down again
  - fewer calls

Topics:

- **Processes and Threads**
- **Processes**
  - Creation
  - Detecting termination
- **Threads**
- **Synchronization**
- **Conclusion**

Detecting Process Termination

We'll consider three cases:

- detecting the termination of a child
  - this is the only behaviour POSIX describes
- client-server relationship
- death pulse

Process Termination - Child death

When a child dies:

- the parent will be sent a **`SIGCHLD`** signal
  - **`SIGCHLD`** does not terminate a process
- the parent can determine why the child died by calling *waitpid()* or other *wait*()* functions
- if the parent does not wait on the child, the child will become a zombie
  - a zombie uses no CPU, most resources it owns are freed, but an entry remains in the process table to hold its exit status
  - **`signal(SIGCHLD, SIG_IGN)`** in the parent will prevent the notification of death and creation of zombies

If you have a client-server relationship:

- a server can get notification if any of its clients die
- a client can get notification if any of its servers die
- these apply for QNX message passing
- these notifications are also delivered in case of severed network connection
- these notifications happen on death, but can happen otherwise as well
  - but only happen when the relationship between client and server is severed

Process Termination – Death pulse

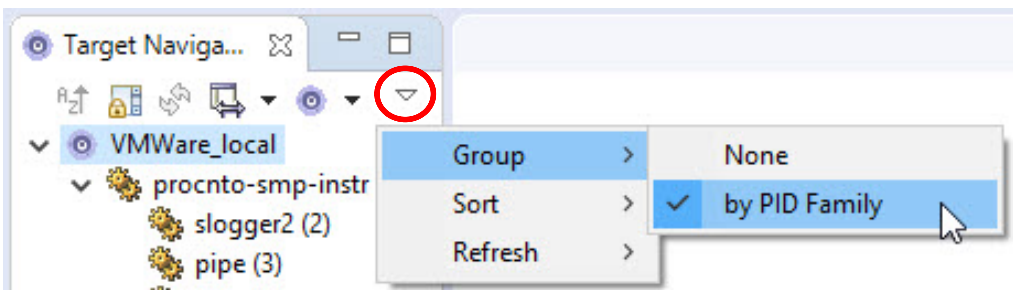
You may ask the OS to deliver a notification whenever a process dies:

- register for notification of process death with *procmgr_event_notify()*
- request **`PROCMGR_EVENT_PROCESS_DEATH`** notification
- can request any type of event (signal, pulse, etc) but a pulse ("death pulse") is usually easiest

- pulses and events are covered in the IPC module

Exercise:

- in your **thread** project:
- look at and run **death_pulse.c**
- look at and run **spawn_example.c**
- look at the parent-child relationship
  - try **pidin family** at the command line
  - try group->by PID family in the Target Navigator



- after sleep dies, notice the zombie
  - the zombie goes away after the parent *wait()*s on it
  - killing the parent will also destroy the zombie

Topics:

- **Processes and Threads**
- **Processes**
- **Threads**
  - Creation
  - Operations
- **Synchronization**
- **Conclusion**

QNX supplies three APIs for dealing with threads:

- kernel functions
  - e.g. *ThreadCreate()*, *SyncMutexLock()*
- POSIX functions
  - e.g. *pthread_create()*, *pthread_mutex_lock()*
- C11 thread functions
  - e.g. *thrd_create()*, *mtx_lock()*
- the portable (POSIX, C11) functions are built on top of the kernel calls
- we recommend using the portable functions
- this section will teach the POSIX functions

Topics:

Processes and Threads

Processes

Threads

- Creation
- Operations

Synchronization

Conclusion

Thread Creation - pthread_create()

To create a thread, use:

```
pthread_create (pthread_t *tid, pthread_attr_t *attr,
                void *(*func) (void *), void *arg);
```

Example:

```
pthread_create (&tid, &attr, &func, &arg);
```

- the thread will start execution in *func()*. *func()* is the "main" for the thread. All other parameters can be `NULL`
- on return from *pthread_create()*, the `tid` parameter will contain the tid (thread id) of the newly created thread
- `arg` is miscellaneous data of your choosing to be passed to *func()*
- `attr` allows you to specify thread attributes such as what priority to run at, ...

Setting up thread attributes

```
pthread_attr_t attr;

pthread_attr_init(&attr);
... /* set up the pthread_attr_t structure */
pthread_create (&tid, &attr, &func, &arg);
```

- *pthread_attr_init()* sets the **`pthread_attr_t`** members to their default values
- we'll talk about some of the things you might set in the attribute structure

Functions for setting attributes

– initializing, destroying

pthread_attr_init(), *pthread_attr_destroy()*

– setting it up

pthread_attr_setdetachstate(), *pthread_attr_setinheritsched()*, *pthread_attr_setschedparam()*, *pthread_attr_setschedpolicy()*, *pthread_attr_setstackaddr()*, *pthread_attr_setstacksize(), …*

Setting priority and scheduling algorithm:

```
struct sched_param param;
pthread_attr_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
param.sched_priority = 15;
pthread_attr_setschedparam (&attr, &param);
pthread_attr_setschedpolicy (&attr, SCHED_RR);
pthread_create (NULL, &attr, func, arg);
```

Thread Attributes - Stack Allocation

You can control the thread's stack allocation:

– to set the maximum size:

```
pthread_attr_setstacksize (&attr, size);
```

– to provide your own buffer for the stack:

```
pthread_attr_setstackaddr (&attr, addr);
```

– to force stack allocation on thread creation:

```
pthread_attr_setstacklazy (&attr,
  PTHREAD_STACK_NOTLAZY);
```

Thread Attributes - Stack Allocation

Thread stack allocation can be automatic:

```
size = 0;                 // default size
addr = NULL;              // OS allocates
```

partly automatic:

```
size = desired_size;
addr = NULL;              // OS allocates
```

or totally manual:

```
size = sizeof (*stack_ptr);
addr = stack_ptr;
```

Your stack size should be the sum of:

```
PTHREAD_STACK_MIN +
platform_required_amount_for_code;
```

Topics:

Processes and Threads

Processes

Threads

- Creation
- Operations

Synchronization

Conclusion

Some thread operations:

pthread_exit (retval)	terminate the calling thread
pthread_join(tid, &retval)	wait for thread to die & get return value
pthread_kill (tid, signo)	set signal **`signo`** on thread **`tid`**
pthread_cancel(tid)	cancel a thread – request that it terminate
pthread_detach (tid)	make the thread detached (i.e. unjoinable)
tid = pthread_self ()	find out your thread id
pthread_setname_np()	name your thread

Set/get priority and scheduling algorithm:

– setting:

```
struct sched_param param;

param.sched_priority = new_value;
pthread_setschedparam (tid, policy, &param);
```

– getting:

```
int policy;
struct sched_param param;

pthread_getschedparam (0, &policy, &param);
printf("my priority is %d\n", param.sched_priority);
```

Waiting for threads to die & finding out why

– if a thread is "joinable" then you can wait for it to die

```
pthread_create (&tid, …, worker_thread, …);
// at this point, worker_thread is running
// ... do stuff
// now check if worker_thread died or wait for
// it to die if it hasn't already
pthread_join (tid, &return_status);
```

– if it dies before the call to *pthread_join()* then *pthread_join()* returns immediately and **`return_status`** contains the thread's return value or the value passed to *pthread_exit()*

– once a *pthread_join*() is done, the information about the dead thread is gone

Thread Termination

Thread termination and clean up:

- threads may self-terminate:
  - thread is self-terminating, so should clean up first as needed
  - calls **`pthread_exit(void *return_val)`**
    - another thread in the process can get the **`return_val`** with *pthread_join()*
  - return from the thread function
    - the wrapping (libc) code will call *pthread_exit()*
- a thread may terminate another thread in the process:
  - *pthread_cancel()* requests that the thread die
    - may be delayed, or have a clean-up handler
  - *pthread_abort()* kills the thread immediately

Note: *pthread_kill()* either:

- the signal is masked, ignored, or handled
- OR kills the entire process

Both processes and threads can die:

- if any thread calls *exit()*, then the process dies and any/all remaining threads are terminated
- the first thread in a process is called the "main thread" because it calls the function *main()*
  - if *main()* returns, *exit()* is called and the process dies
  - if the main thread goes away (e.g. *pthread_exit()*), its stack is not released
    - important things like arguments and the environment data are stored in its stack
- if all threads in a process die, the process also dies
  - normal exit processing will not happen
- if a process dies from a signal, normal exit processing will not happen
- however a process dies, all process resources (e.g. memory, fds, connections, channels, etc) will be freed/released/destroyed

Topics:

Processes and Threads

Processes

Threads

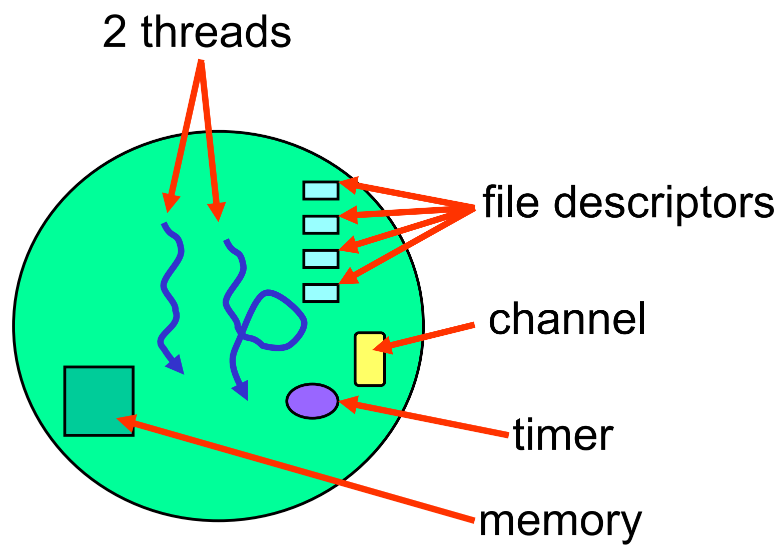
Synchronization

- mutexes
- condvars
- atomic operations

Conclusion

Threads within a process share:

- Timers
- Channels
- Connections
- Memory Access
- File pointers / descriptors
- Signal Handlers



Synchronization - Problems

Threads introduce new solutions, but new problems as well:

Common memory areas:

- multiple writers can overwrite each other's values,
- readers don't know when data is stable or valid,

Similar problems occur with other shared resources...

These are problems with:

SYNCHRONIZATION

In this section, we'll see some tools for solving these problems:

- mutexes
- condvars
- atomic operations

Other synchronization tools we won't see:

- semaphores:
  - POSIX counting semaphores for counted dispatch
- reader/writer locks:
  - allows multiple readers and no writers or
  - only one writer and no readers
- once control:
  - a way of having some code be executed at most once for the life of a process
  - useful for initializing a library
- thread local storage
  - a way of setting aside memory on a per-thread basis and getting it back later
  - good way for a library to keep per-thread data without knowing that it is being used in a multi-threaded process

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- atomic operations

Conclusion

Mutual Exclusion

"Mutual exclusion" means only *one* thread:

- is allowed into a critical section of code at a time
- is allowed to access a particular piece of data at a time

POSIX provides the following calls:

– administration

```
pthread_mutex_init (pthread_mutex_t *,
                    pthread_mutexattr_t *);
pthread_mutex_destroy (pthread_mutex_t *);
```

– usage

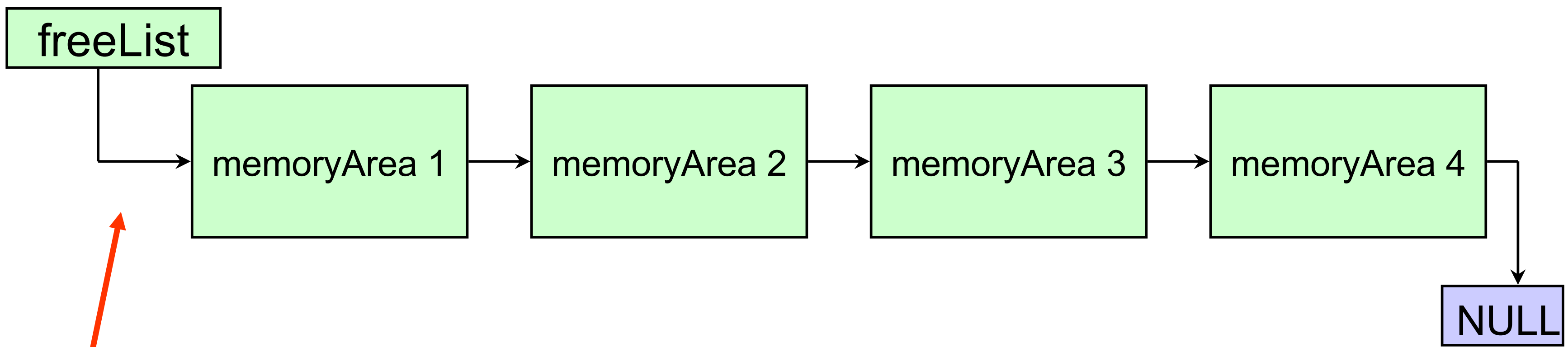
```
pthread_mutex_lock (pthread_mutex_t *);
pthread_mutex_unlock (pthread_mutex_t *);
```

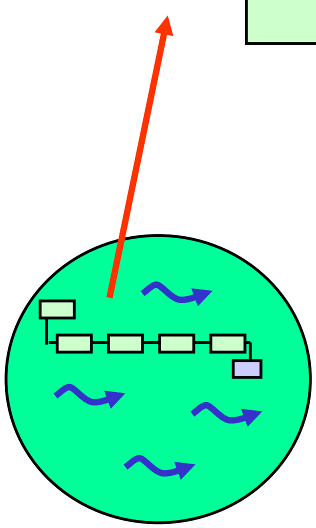
A simple example:

```
pthread_mutex_t myMutex;

init () {
    ...
    // create the mutex for use
    pthread_mutex_init (&myMutex, NULL);
    ...
}
thread_func () {
    ...
    // obtain the mutex, wait if necessary
    pthread_mutex_lock (&myMutex);
    // critical data manipulation area
    // end of critical region, release mutex
    pthread_mutex_unlock (&myMutex);
    ...
}
cleanup () {
    pthread_mutex_lock(&myMutex);
    pthread_mutex_destroy (&myMutex);
}
```

use default attributes

Consider a buffer allocation tool, buf_alloc():





A number of threads request memory from *buf_alloc()*. Internally, *buf_alloc()* maintains a “free list” of memory blocks that are available for allocation. All threads in the process use the same free list.

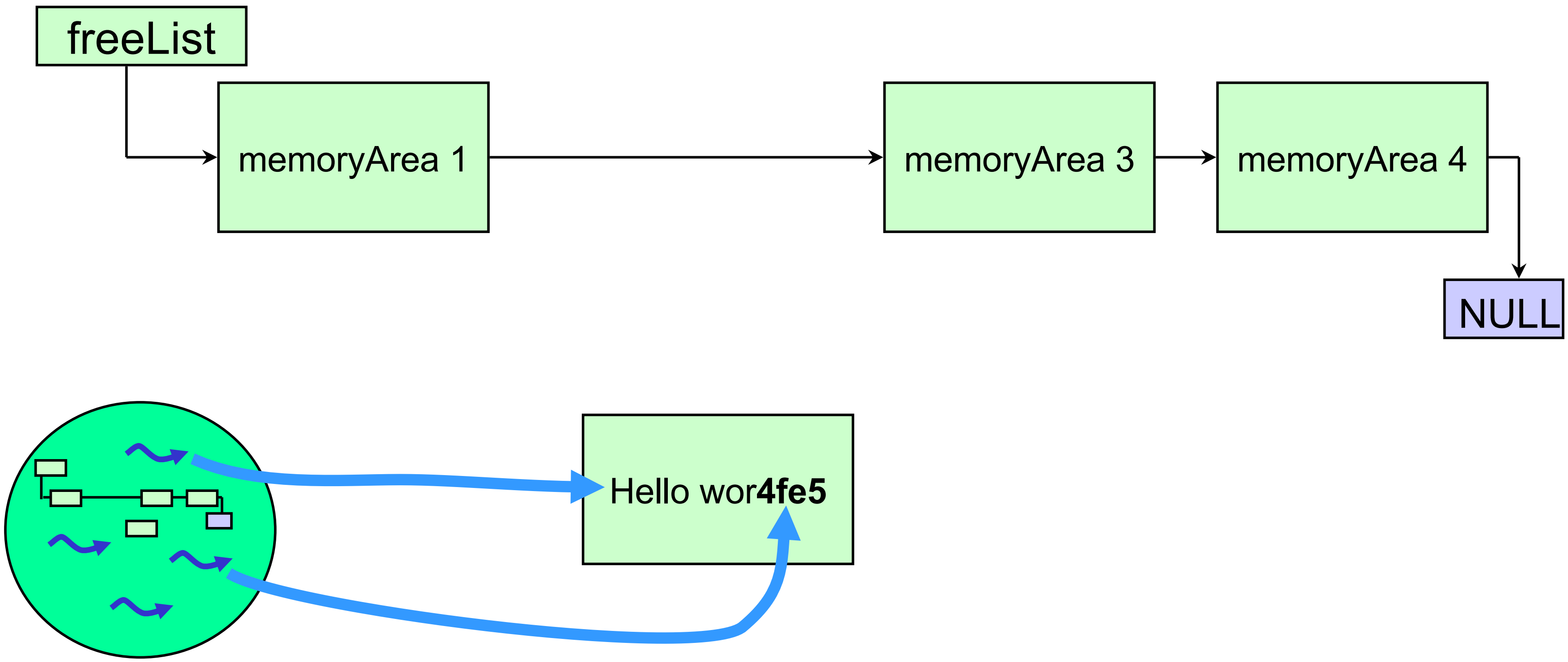
buf_alloc() source might look something like this:

```
void *
buf_alloc (int nbytes)
{
   …
   while (freeList && freeList -> size != nbytes) {
     freeList = freeList -> next;
   }
   if (freeList) {
     … // mark block as used, and return block address to caller
     return (freeList -> memory_block);
   }
   …
}
```

Now consider a number of threads that use buf_alloc():

```
thread1 ()
{
  char    *data;

  data = buf_alloc (64);
}

thread2 ()
{
  char    *other_data;

  other_data = buf_alloc (64);
}
```

Something bad happens!



Mutual Exclusion

The problem is, multiple threads can get in each other's way!

What we need is *exclusive* access to the "`freeList`" data structure!

We'll use a MUTEX to do this...

Let's fix the `buf_alloc` routine:

```
pthread_mutex_t                    alloc_mutex;

void *
buf_alloc (int nbytes)
{
   …
   pthread_mutex_lock (&alloc_mutex);
   while (freeList && freeList -> size != nbytes) {
     freeList = freeList -> next;
   }
   if (freeList) {
     … // mark block as used, and return block
     block = freeList -> memory_block;
     pthread_mutex_unlock (&alloc_mutex);
     return (block);
   }
   pthread_mutex_unlock (&alloc_mutex);
   …
}
```

critical section

Mutex Initialization

To explicitly initialize the mutex:

```
pthread_mutex_init (&alloc_mutex, NULL);
```

If successful, this ensures that all appropriate resources have been allocated for the mutex.

A simple method for mutex initialization:

```
// static initialization of Mutex

pthread_mutex_t alloc_mutex =
  PTHREAD_MUTEX_INITIALIZER;

void *
buf_alloc (int nbytes)
{
  . . .
  // MUTEX will be initialized the first time
  // it is used …
  pthread_mutex_lock (&alloc_mutex);
  . . .
```

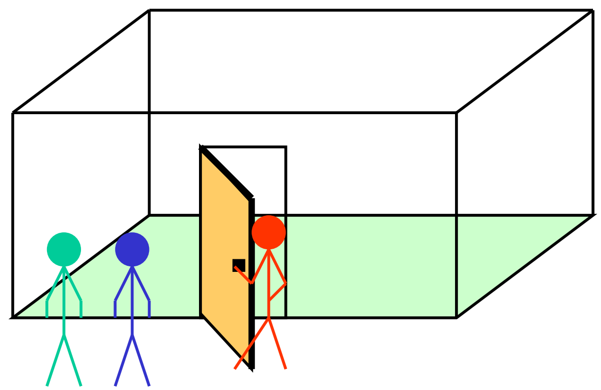
Mark as: “not in use” and “to be initialized the first time that it is used”.

By default, mutexes cannot be shared between processes

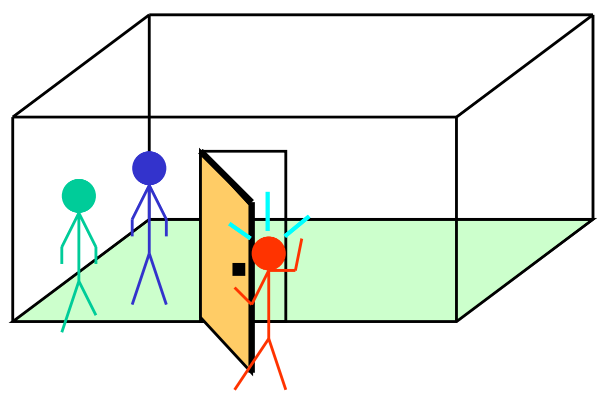
- to make them shared, set the **PTHREAD_PROCESS_SHARED** flag for the mutex
- the mutex should be in shared memory
- e.g.:

```
pthread_mutexattr_t mutex_attr;
pthread_mutex_t *mutex;
pthread_mutexattr_init( &mutex_attr );
pthread_mutexattr_setpshared( &mutex_attr,
  PTHREAD_PROCESS_SHARED);
mutex = (pthread_mutex_t *)shmem_ptr;
pthread_mutex_init( mutex, &mutex_attr );
```

A danger with mutexes



– a mutex is like a lock on a door. To get into the room you must unlock the door. Only one person can unlock it at a time



– but there is nothing stopping someone from going around the door! This is what happens when you forget to use the mutex

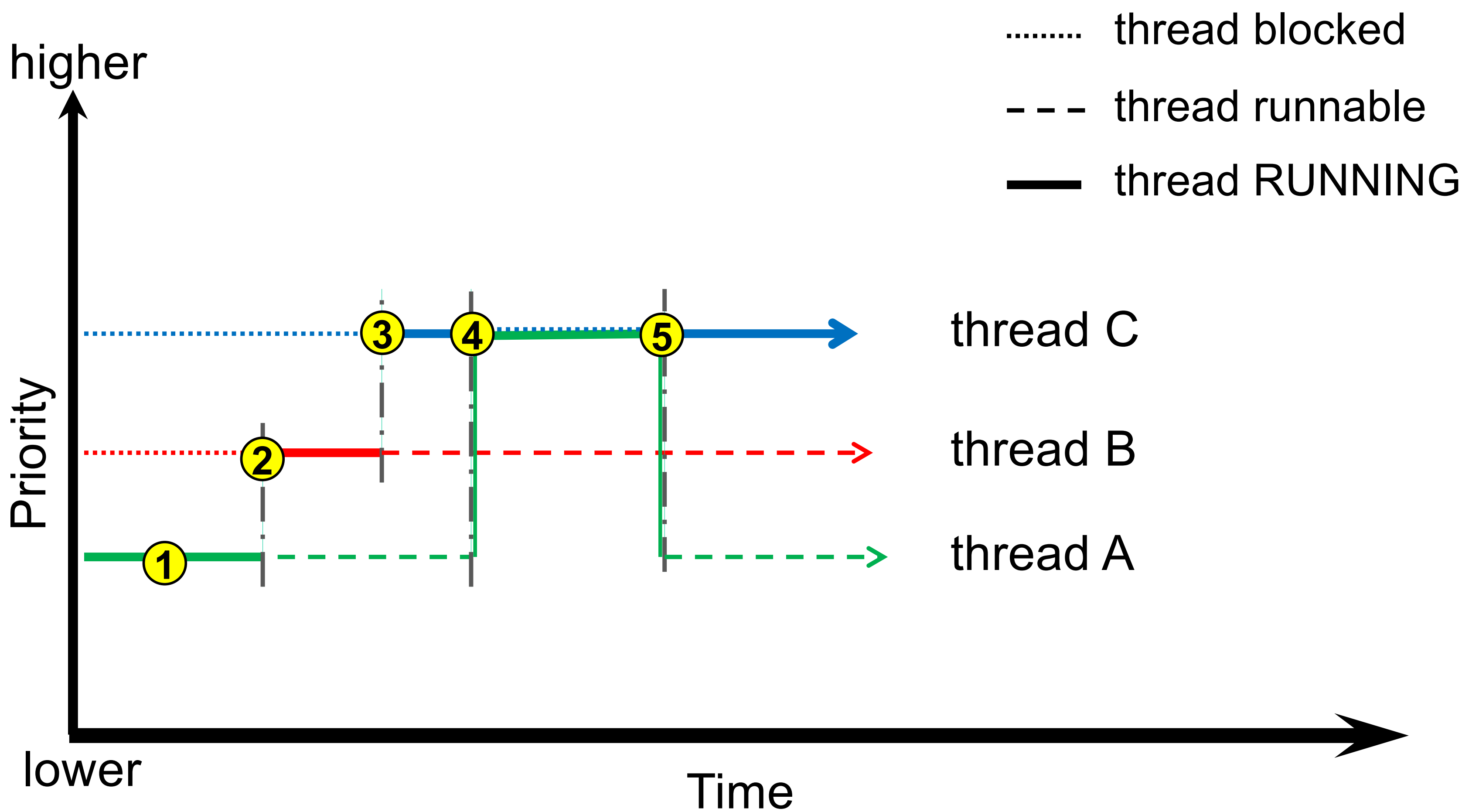
Deadlock:

- Issue: Locking multiple mutexes in multiple threads could lead to a deadlock:
  - thread 1 locks mutex A
  - thread 2 locks mutex B
  - thread 1 then attempts to acquire mutex B which thread 2 still owns
  - thread 2 then attempts to acquire mutex A which thread 1 still owns
  - deadlock as both threads are MUTEX Blocked on each other's mutex

  with more threads, locking more mutexes, it is possible to have a circular deadlock which are much more difficult to spot
- Design Solution:
  - establish a locking hierarchy that specifies the order in which threads acquire mutexes. Ensure that all threads acquire the mutexes in the specified order to avoid deadlock

Priority Inheritance

Threads with a mutex locked can have their priority temporarily boosted:



Designing with Mutexes - Mutex Locked Time

Keep mutexes locked for short periods

- keep a mutex locked for as short a time as reasonable
- while one thread has the mutex locked, other threads that want the resource protected by the mutex have to wait
  - the thread may spend time doing work at a higher than normal priority
- with QNX there is the added benefit that if the mutex is not locked, and you try to lock it, the amount of code you'll end up doing is very short and very fast...

pthread_mutex_lock() is very efficient:

```
int
pthread_mutex_lock (pthread_mutex_t *mutex)
{
    int owner, ret, id = LIBC_TLS() -> owner;

    // is it unlocked?
    if ((owner = _smp_cmpxchg (&mutex -> owner, 0, id)) == 0) {
      ++mutex -> count;
      return (EOK);
    }
    // is it locked by me?
    if ((owner & ~_NTO_SYNC_WAITING) == id) {
      if ((mutex -> count & _NTO_SYNC_NONRECURSIVE) == 0) {
            ++mutex -> count;
            return (EOK);
      }
      return (EDEADLK);
    }
    // someone else owns it, wait for it
    if ((ret = SyncMutexLock_r ((sync_t *) mutex)) != EOK) {
      return (ret);
    }
    // we have it, so bump the count
    ++mutex -> count;
    return (EOK);
}
```

Uncontested Access, Fast!

note: code simplified for clarity

Exercise:

- in your **`thread`** project:
- look at nomutex.c
  - run the program
  - change **`NUMTHREADS`** to **`4`**
  - build & run the program again
    - note the change in behaviour (not equal *printf()*s appear)
- modify mutex_sync.c to fix the problem
  - add a mutex to control access to the critical section
    - lock and unlock it where appropriate
    - don't keep it locked for too long
  - how does this affect performance?

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- atomic operations

Conclusion

Consider a simple case where:

- we need to block, waiting for another thread to change a variable:

```
volatile int state;

thread_1 ()
{
  while (1) {
    // wait until "state" changes,
    // then, perform some work
  }
}
```

- condvars provide a mechanism for doing this

The condvar calls:

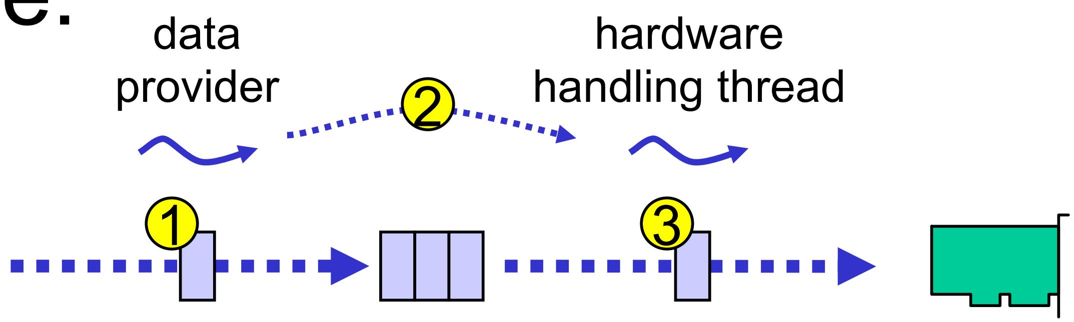
– administration

```
pthread_cond_init (pthread_cond_t *,
                   pthread_condattr_t *);
pthread_cond_destroy (pthread_cond_t *);
```

– usage

```
pthread_cond_wait (pthread_cond_t *,
                   pthread_mutex_t *);
pthread_cond_signal (pthread_cond_t *);
pthread_cond_broadcast (pthread_cond_t *);
```

An example:



1. a data provider thread gets some data, likely from a client process, and adds it to a queue
2. it tells the hardware handling thread that data's there
3. the hardware writing thread wakes up, removes the data from the queue and writes it to the hardware

To do all this we need two things:

- a mutex to make sure that the two threads don't access the queue data structure at the same time
- a mechanism for the data provider thread to tell the hardware writing thread to wake up

Hardware handling thread's code:

```
while (1) {
  pthread_mutex_lock (&mutex);           // get exclusive access
  while (!data_ready)
    pthread_cond_wait (&cond, &mutex);   // we wait here

  /* get and decouple data from the queue */
  while ((data = get_data_and_remove_from_queue ()) != NULL) {
    pthread_mutex_unlock (&mutex);
    write_to_hardware (data); // pretend to do this
    free (data);              // we don't need it after this
    pthread_mutex_lock (&mutex);
  }
  data_ready = 0;               // reset flag
  pthread_mutex_unlock (&mutex);
  /* do further work */
}
```

Data providing thread's code:

```
pthread_mutex_lock (&mutex);      // get exclusive access
add_to_queue (buf);
data_ready = 1;                   // set the flag
pthread_cond_signal (&cond);      // notify a waiter
pthread_mutex_unlock (&mutex);    // release exclusivity
```

Let’s zoom in on the “wait”:

`pthread_cond_wait (&condvar, &mutex);`

- **`unlock`**
- **`wait`**
- becomes runnable
- **`lock`**

`unlock`

– allows other threads to get access

`wait`

– this is the actual “waiting” part

becomes runnable

– just because the thread is no longer waiting, that doesn’t mean that it gets the CPU. The lock doesn’t happen until the function is returning.

`lock`

– ensures that we once again have access

Condvars - Example

Why did we do this test?

```
while (1) {
  pthread_mutex_lock (&mutex);           // get exclusive access
  while (!data_ready )
    pthread_cond_wait (&cond, &mutex);   // we wait here
  ...
  data_ready = 0;                        // reset flag
  pthread_mutex_unlock (&mutex);
}
```

- if you signal a condvar when no thread is waiting then the signal is lost
- it's possible the signal was sent between 1 and 2 but since we weren't waiting for it yet, the signal was lost
- so as well as signalling the condvar, the signaller also sets the **`data_ready`** flag (does **`data_ready = 1`**)

Signalling vs broadcasting:

– Threads 1, 2 and 3 (all at the same priority) are waiting for a change via *pthread_cond_wait()*,

– Thread 4 makes a change, signals the variable via *pthread_cond_signal()*,

– The longest waiting thread (let's say "2") is informed of the change, and tries to acquire the mutex (done automatically by the *pthread_cond_wait()*),

– Thread 2 tests against its condition, and either performs some work, or goes back to sleep

What about threads 1 and 3?

– they never see the change!

If we change the example:

– use *pthread_cond_broadcast()* instead of *pthread_cond_signal()*,

– then all three threads will get the signal

- they will all become runnable, but only one can lock the mutex at a time so they will end up taking turns.

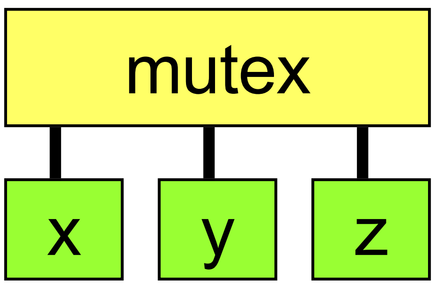
Why use one over the other?

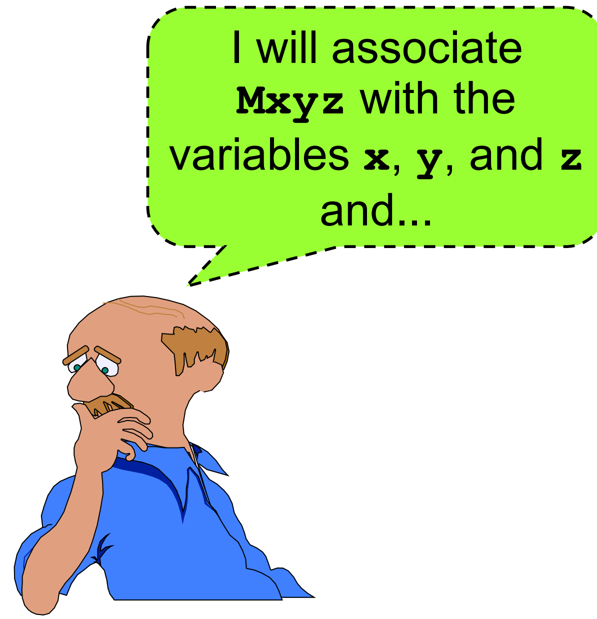
- use a signal if:
  - you have only one waiting thread, or
  - you need only one thread to do the work and you don't care which one
- use a broadcast if you have multiple threads and:
  - they all need to do something, or
  - they don't all need to do something but you don't know which one(s) to wake up

Condvars

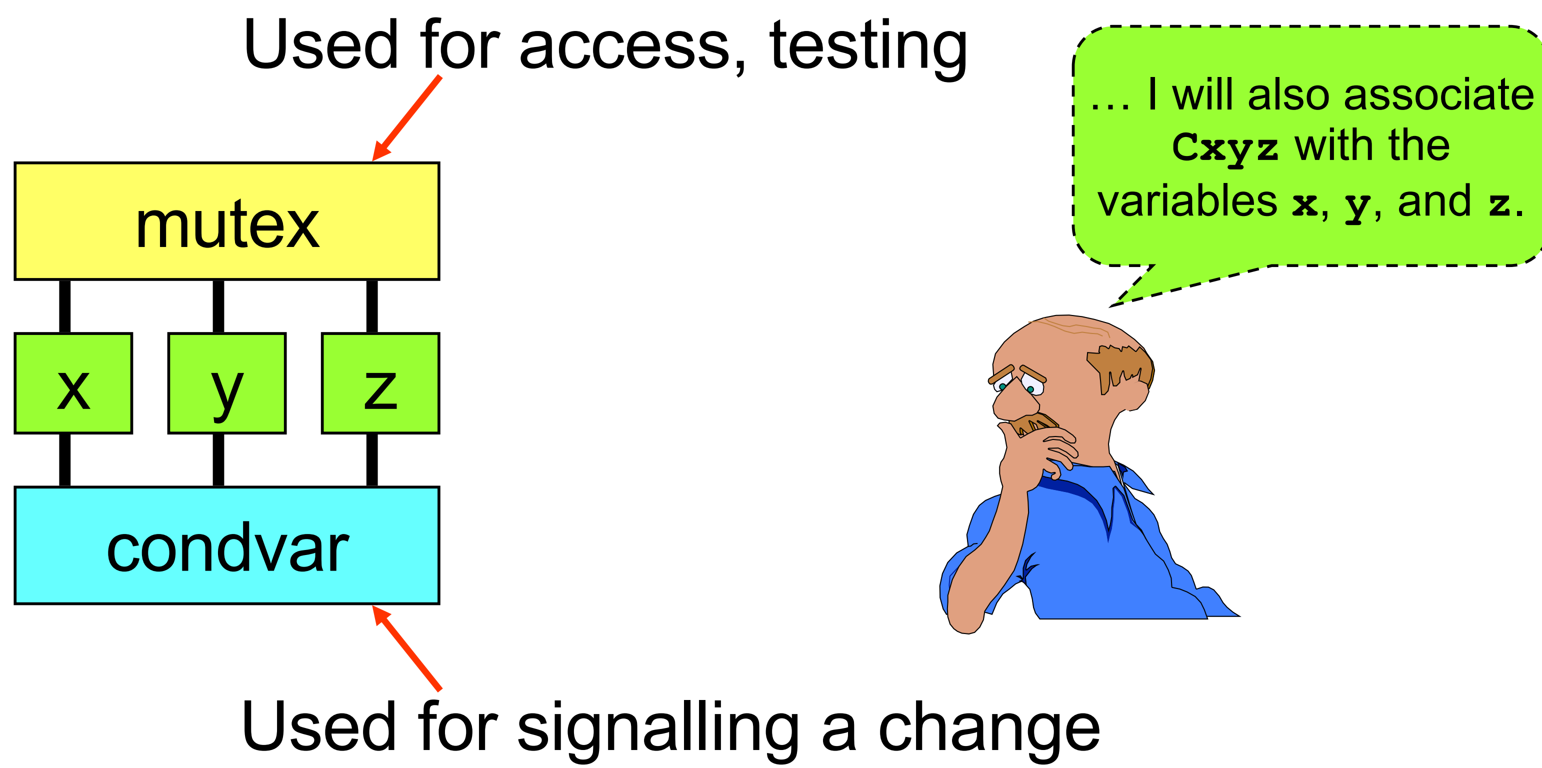
Let’s examine condvars in a little more detail:

The programmer associates a mutex with one or more variables, for locking

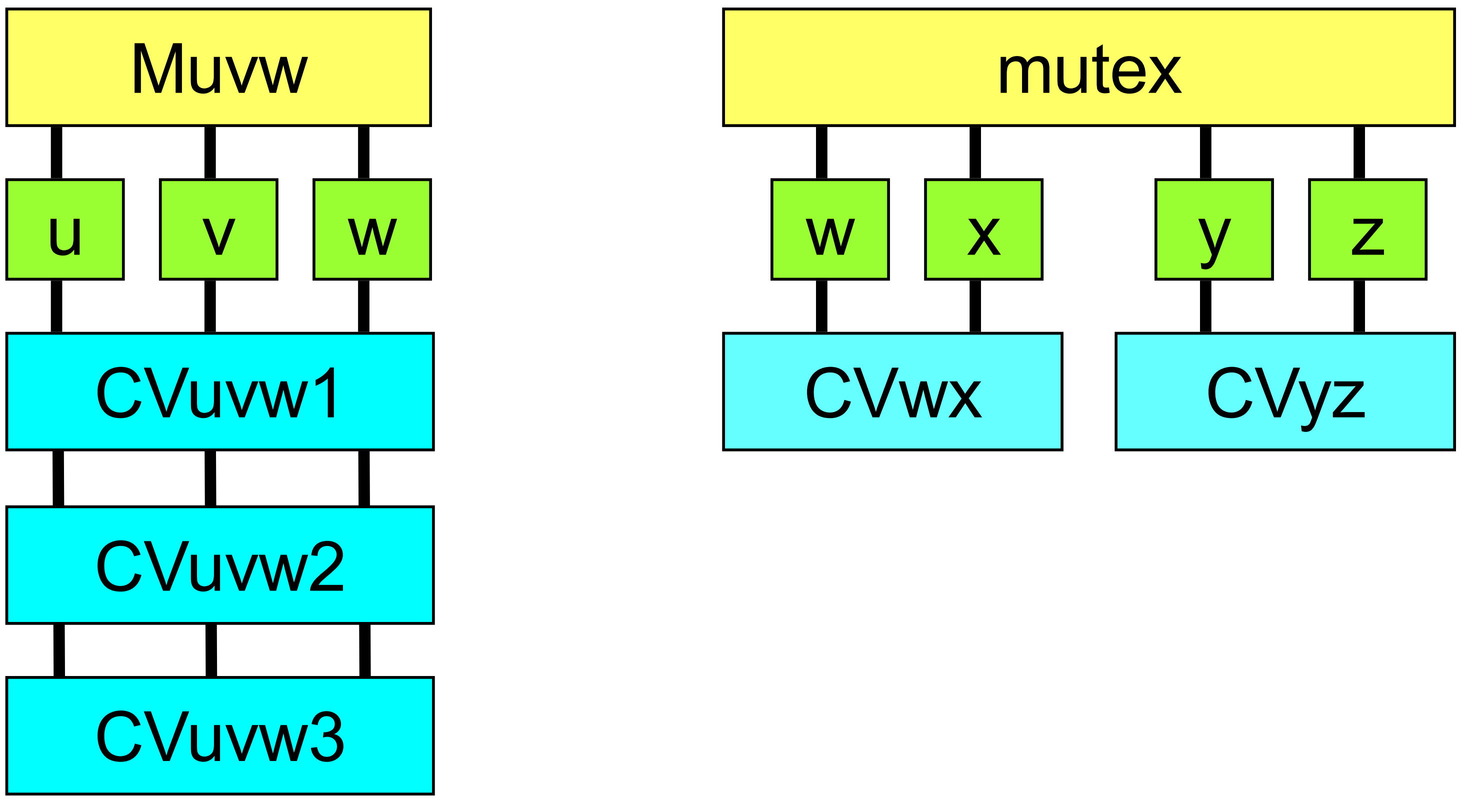




And, the programmer associates a condition variable as well:



The association need not be one-to-one:



By default, condvars cannot be shared between processes

- to make them shared, set the **PTHREAD_PROCESS_SHARED** flag for the condvar
- the condvar should be in shared memory
- e.g.:

```
pthread_condattr_t cond_attr;
pthread_cond_t *cond;
pthread_condattr_init( &cond_attr );
pthread_condattr_setpshared( &cond_attr,
  PTHREAD_PROCESS_SHARED);
cond = (pthread_cond_t *)shmem_ptr;
pthread_cond_init(cond, &cond_attr );
```

Producer / Consumer example:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;
volatile int    state = 0;
volatile int    product = 0;

void *consume (void *arg) {
    while (1) {
        pthread_mutex_lock (&mutex);
        while (state == 0) {
            pthread_cond_wait (&cond, &mutex);
        }
        printf ("Consumed %d\n", product);
        state = 0;
        pthread_cond_signal (&cond);
        pthread_mutex_unlock (&mutex);
        do_consumer_work ();
    }
    return (0);
}
```

Condvars

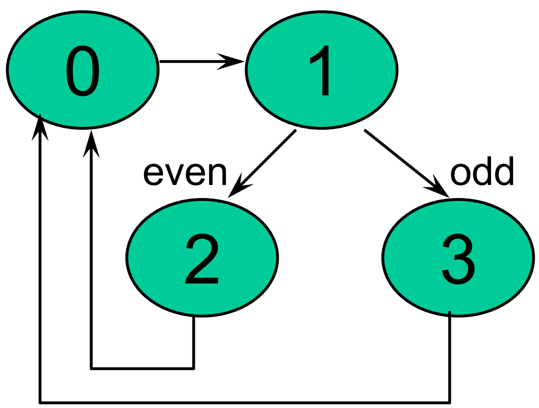
```
void *produce (void *arg) {
    while (1) {
        pthread_mutex_lock (&mutex);
        while (state == 1) {
            pthread_cond_wait (&cond, &mutex);
        }
        printf ("Produced %d\n", product++);
        state = 1;
        pthread_cond_signal (&cond);
        pthread_mutex_unlock (&mutex);
        do_producer_work ();
    }
    return (0);
}

int main () {
    pthread_create (NULL, NULL, &produce, NULL);
    consume (NULL);
    return (EXIT_SUCCESS);
}
```

Exercise:

– in your `thread` project:

– modify the source for `condvar.c` to:

- have a 4 state state-machine
- have 4 threads running, each handling a particular state
- use only one condition variable
- state 1 will maintain a counter and go to state 2 or 3 based on this counter



Topics:

For short operations, such as incrementing a variable:

`atomic_add`	does += some value
`atomic_add_value`	*atomic_add()*, and returns original
`atomic_clr`	does &= ~some value
`atomic_clr_value`	*atomic_clr()*, and returns original
`atomic_set`	does |= some value
`atomic_set_value`	*atomic_set()*, and returns original
`atomic_sub`	does -= some value
`atomic_sub_value`	*atomic_sub()*, and returns original
`atomic_toggle`	does ^= some value
`atomic_toggle_value`	*atomic_toggle()*, and returns original

These functions:

- are guaranteed to complete correctly despite pre-emption or interruption
- can be used between two threads (even on SMP)
- can be used between a thread and an ISR

Topics:

- Processes and Threads
- Processes
- Threads
- Synchronization
- → Conclusion

Conclusion

You learned:

- what a process is and what a thread is
- why you’d use threads
- how to create processes and threads
- how to detect when processes and threads die
- how to synchronize among threads using mutexes, condvars, and other methods