Processes, Threads & Synchronization

QNX SOFTWARE SYSTEMS

NOTES:

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, semaphores, ...

NOTES:

Topics:

→ **Processes and Threads**

Processes

Threads

Synchronization

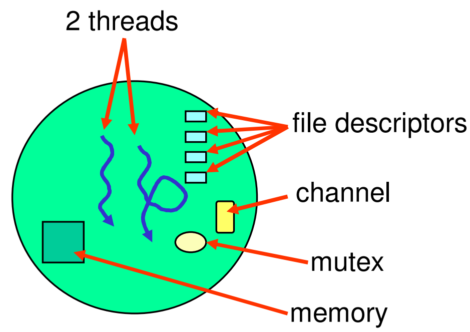
Conclusion

NOTES:

Processes

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

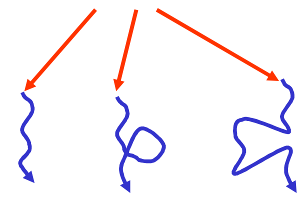
NOTES:

Threads

QNX SOFTWARE SYSTEMS

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 SMP
  - signal mask
  - and others
- all its attributes have to do with running code

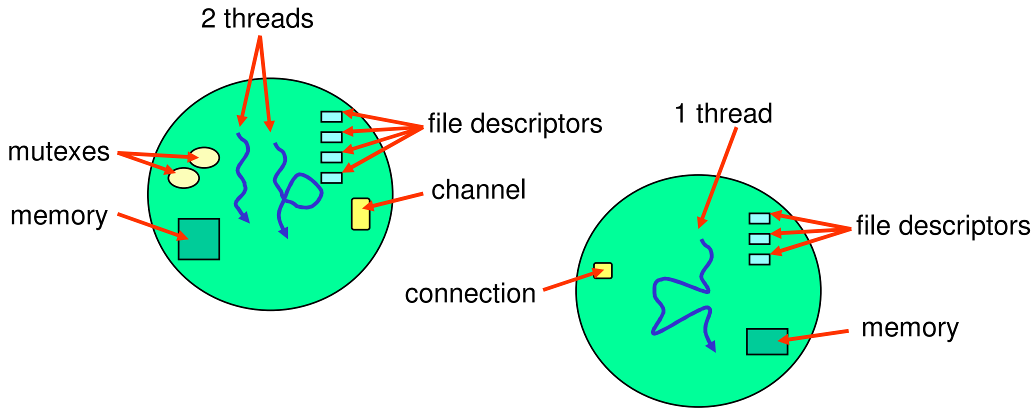


NOTES:

Each thread also has its own **errno** value.

Threads run in a process:

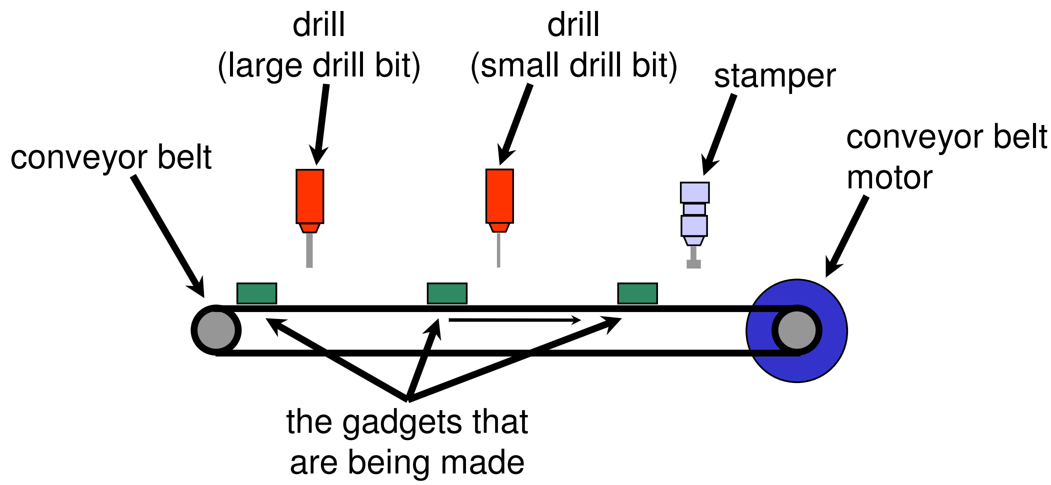
- a process must have at least one thread
- threads in a process share all the process resources



Threads run code, processes own resources

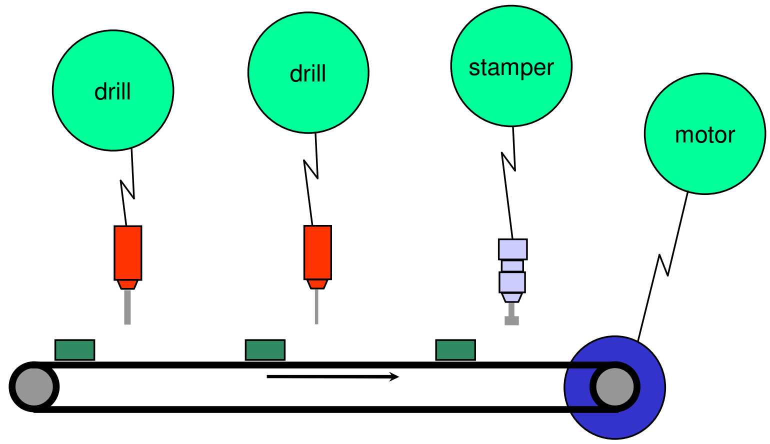
NOTES:

Example - Assembly line



NOTES:

The processes that monitor and control these devices:

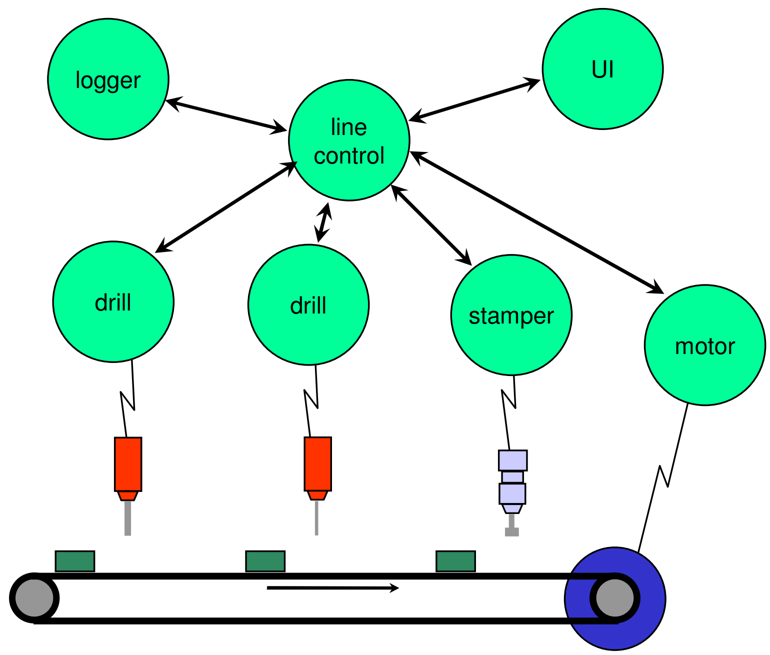


NOTES:

Good things about this process model:

- if we add new devices to the assembly line we need only add new processes, the existing ones are not affected
- if we remove devices from the assembly line we simply don't run the processes that monitor and control those devices
- if we find a bug in the motor process (for example) we need only fix the motor process. The other processes are not affected.

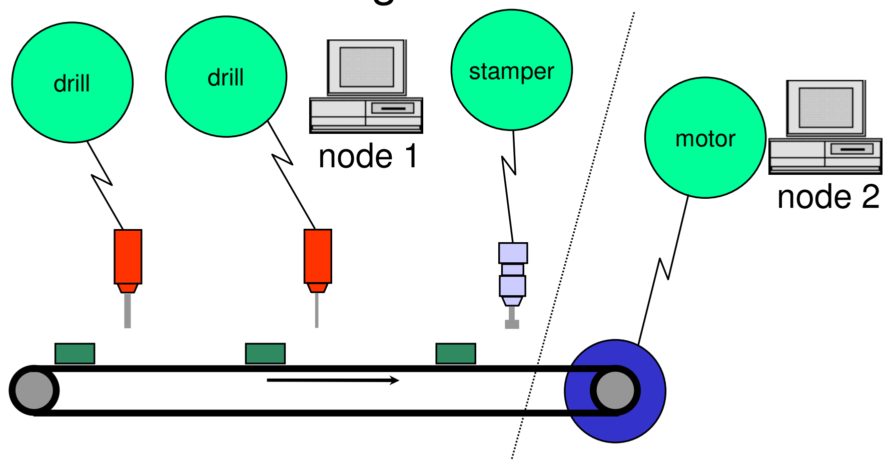
Of course, there will be more levels:



NOTES:

Another advantage to this process model:

- they can be spread out across multiple QNX nodes and still work together:



NOTES:

This may be done for a variety of reasons:

- conveyor belt/motor system may have been provided by a different manufacturer than the drills and stamper
- the motor may physically be very far away from the drills and stamper. It may be desirable to have the QNX node controlling the motor to be physically near the motor.
- the QNX node controlling and monitoring the motor may be deeply embedded in special hardware dedicated to that motor device

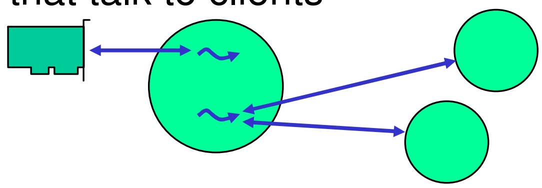
Process opacity:

- 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

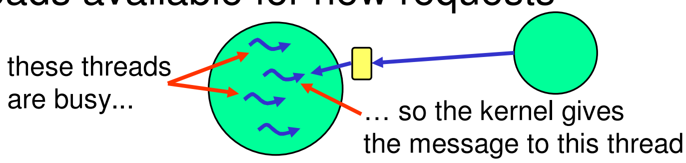
NOTES:

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



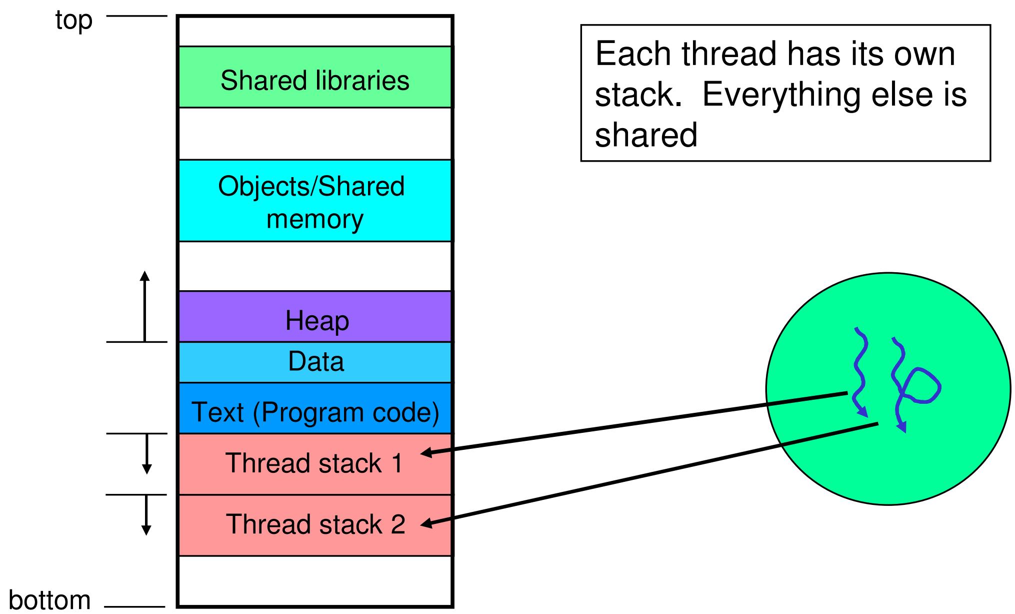
- pool of worker threads. If one or more threads are busy handling previous requests there are still other threads available for new requests



NOTES:

Process Virtual Address Space

Virtual address space of a process:

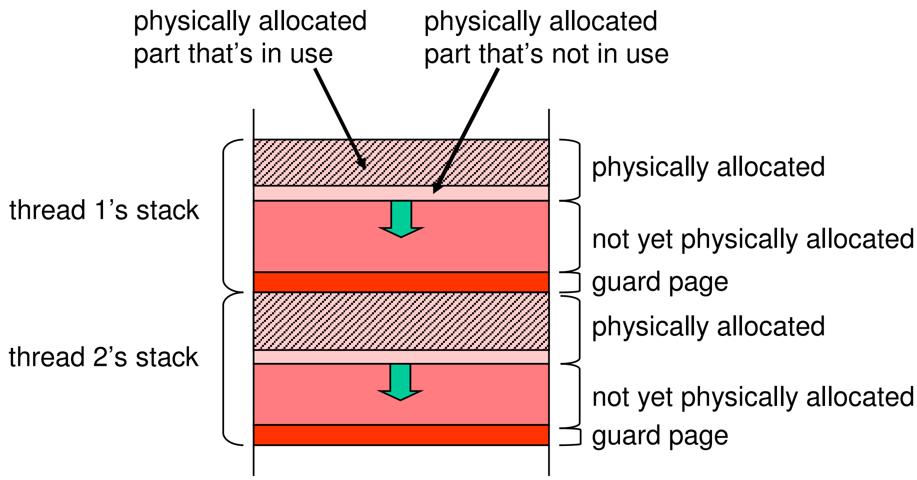


NOTES:

Thread Stacks

Thread stacks:

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



Processes, Threads & Synchronization
2010/06/22 R11

14

NOTES:

Only enough pages of physical memory for the amount of stack you've used are allocated at any one time.

The guard page is how the kernel detects when you've used up all of your stack. Trying to write into the guard page will cause the processor to generate a fault which the kernel will handle by setting a SIGSEGV signal on your thread, terminating the thread and, usually, your process too.

Topics:

Processes and Threads

Processes

- Creation
- Detecting termination

Threads

Synchronization

Conclusion

NOTES:

Process Creation

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
- *spawn()*, *spawn*()*
  - load a new program creating a new process for it

NOTES:

Process Creation - fork()

fork() will create a copy of your process:

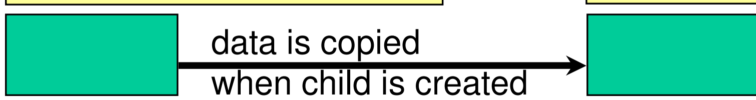
- the child will:
  - be an identical copy of the parent
  - start from the *fork()*
  - initially have the same data as the parent
- QNX does not support *fork()* in a multi-threaded process
- *fork()* returns child's pid for the parent and 0 for the child

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
}
```



NOTES:

In doing the copy of the data, in fact the entire address space of the parent will be replicated. Any read-only sections of the parent's address space (e.g. code or shared library) will be mapped into the child. The child will also get mappings of any shared objects, or even hardware mappings, the parent had. Any writeable data, including heap and stack, from the parent will be copied to the child.

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

NOTES:

Process Creation - exec*()

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

NOTES:

Process Creation - spawn*()

To run a new program:

- use the *spawn*()* calls or *spawn()*
  - 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
- *spawn()* does the actual work
  - gives more control
  - more complex to use

NOTES:

Fork & exec vs spawn

- fork & exec is traditional Unix way
  - portable
  - inefficient
- 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
  - can be done from a multi-threaded process

NOTES:

spawn(), and *spawnp()* were in a draft of POSIX 1003.1d, but were dropped before the final release.

Topics:

Processes and Threads

Processes

– Creation

– Detecting termination

Threads

Synchronization

Conclusion

NOTES:

Detecting Process Termination

We'll consider three cases:

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

NOTES:

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

NOTES:

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

NOTES:

These notifications are delivered by the O.S. as pulses, and require setting specific flags when creating a channel. See *ChannelCreate()* documentation for more detail, in particular the **`_NTO_CHF_COID_DISCONNECT`** and **`_NTO_CHF_DISCONNECT`** flags.

This is discussed in more detail in the IPC module.

Process Termination - Other cases

Other methods of getting notification of process death:

- use the High Availability Manager (**`ham`**) from the High Availability Framework
- register for notification of system daemons with *procmgr_event_notify()*
- register for core dump notification
  - this is what **`dumper`** does

NOTES:

Topics:

Processes and Threads

Processes

Threads

- Creation
- Detecting termination
- Operations

Synchronization

Conclusion

NOTES:

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, ...

NOTES:

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

NOTES:

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(), …*

NOTES:

Generally, you would call *pthread_attr_init()* to initialize an attributes structure for use, and then you would call the other *pthread_attr_set*()* functions to manipulate that attributes structure. For the pages that follow, we will assume that *pthread_attr_init()* has always been called.

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);
```


NOTES:

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);
```


NOTES:

Stack size and allocation for the main thread is control by compile (really post-linker) directives.

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;
```


NOTES:

The “partly automatic” form, where we specify the size but not the data area is very useful. Using this form, the kernel allocates a stack (of the desired size) for us dynamically, and then ***automatically deallocates*** the stack upon termination of the thread -- we don’t have to worry about cleaning up after ourselves. (Same thing happens with the fully automatic form -- the kernal allocates a stack (of default size) and is then responsible for cleaning it up after the thread has gone away).

The totally manual approach is excellent for stack depth usage monitoring -- place a signature throughout the stack, and let the thread run. When the thread has finished, you can analyze how much of the signature got damaged, which is a direct indication of how much stack the thread used!

The **`pidin`** utility, when run with the **`mem`** option, shows which threads were created with a manually specified stack by putting a ‘*’ beside their stack information The first thread in a process (main) is done manually so that if it *pthread_exit()*s, its stack is still available (so that the args are still available).

In the IDE, stacks can be found in the Memory Information view (usually viewed in the QNX Memory Analysis or QNX System Information perspectives).

Topics:

Processes and Threads

Processes

Threads

- Creation
- Detecting termination
- Operations

Synchronization

Conclusion

NOTES:

Thread Termination - Joining

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

NOTES:

Unlike in the case of processes, where only the parent can wait for a child and get its exit status, any other thread in the process can call *phread_join()* and get the exit status of the dead thread.

You can make a thread unjoinable. This is called “detached”. You cannot wait for the thread to die and when the thread dies, nothing will be remembered about it. To make it detached at thread creation time:

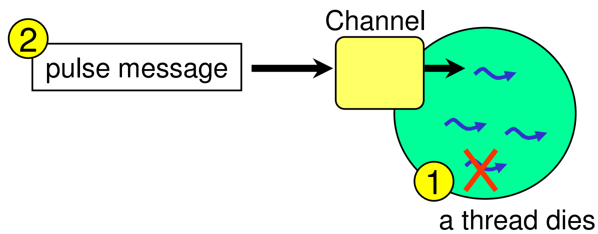
```
pthread_attr_init (&attr);
pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
pthread_create (&tid, &attr, worker_thread, NULL);
```

The kernel can send you a pulse message:

- to get a pulse message when any thread in your process dies, set the **`_NTO_CHF_THREAD_DEATH`** flag when you create your channel

```
chid = ChannelCreate (_NTO_CHF_THREAD_DEATH);
```

- in the pulse message, the code will be **`_PULSE_CODE_THREADDEATH`** and the value will be the tid of the thread that died



NOTES:

But, this is almost completely useless. Generally either you know when a thread is dying (it called pthread_exit(), or some other thread in your process terminated it), or the thread is “crashing” (e.g. SIGSEGV) in which case the entire process is crashing, and there won’t be another thread around to get this notification.

Topics:

Processes and Threads

Processes

Threads

- Creation
- Detecting termination
- → Operations

Synchronization

Conclusion

NOTES:

Some thread operations:

pthread_exit (retval)	terminate the calling thread
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

NOTES:

Regarding *pthread_exit()*... If the thread is joinable, the value **retval** is made available to any threads joining the terminating thread; otherwise if the thread is detached, all system resources allocated to the thread are immediately reclaimed, meaning that the **retval** value is lost. If any thread except the *main()* thread simply drops off of the end of the function at which it was invoked, this will implicitly call *pthread_exit()* as well.

Regarding *pthread_kill()*… If **sig** is zero, then no signal is issued, but error checking is done (this is a convenient way to test if the thread still exists):

```
if (pthread_kill (tid, 0) == EOK) {
    // thread "tid" exists
}
```

Regarding *pthread_detach()*… Once a thread has been created, it cannot make itself joinable again.

Regarding *pthread_equal()*… The thread ID returned is effectively an integer. Therefore, the **pthread_equal()** function call really just does:

```
#define pthread_equal(t1,t2) (t1==t2)
```

… it's a POSIX thing!

Thread Operations - Priorities

Set/get priority and scheduling algorithm:

– setting:

```
struct sched_param param;

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

see the documentation for other members for sporadic scheduling

– getting:

```
int policy;
struct sched_param param;

pthread_getschedparam (tid, &policy, &param);
```

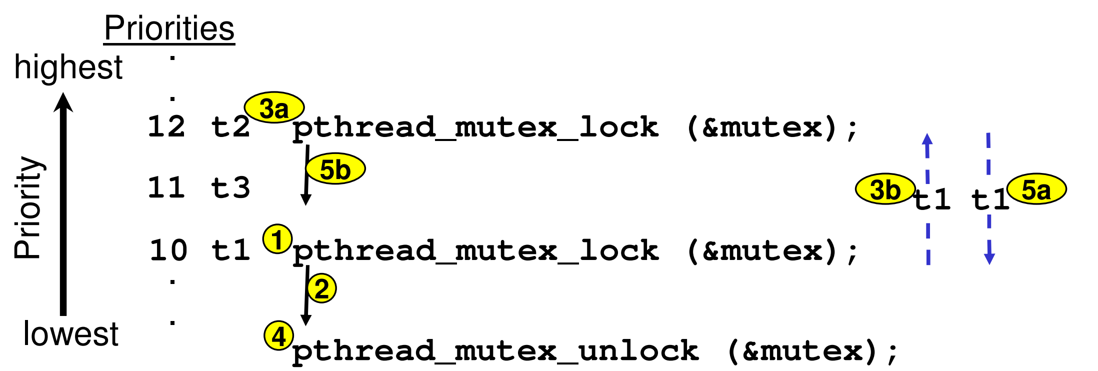
but what priority does this get? ...

NOTES:

Thread Operations - Priorities

Two of the priorities it gets are:

- **`param.sched_priority`** contains the *defined* priority - the priority assigned to the thread (e.g. at thread creation time)
- **`param.sched_curpriority`** contains the *current* priority



NOTES:

1. Thread t1, at priority 10, calls *pthread_mutex_lock()* to lock a mutex.
2. t1 successfully locks the mutex so the call returns.

3a. A little later, thread t2, running at priority 12, tries to lock the same mutex. Since t1 has it locked, t2 will block on the *pthread_mutex_lock()* call.

But at this point we would have t2, a high priority thread, blocked waiting for a mutex that is locked by t1, a lower priority thread. This would be bad so...

3b. During t2's call to pthread_mutex_lock(), the kernel bumps t1's priority up to that of t2 so that t1 continues running at t2's priority. Because of this t2 is not waiting for a lower priority thread!

4. t1, who is at priority 12, calls *pthread_mutex_unlock()* to unlock the mutex.

5a. During the *pthread_mutex_unlock()* call, the kernel drops t1 back to its original priority and…

5b. … since t2 was trying to lock the mutex and t2 is at a higher priority, it gets the mutex and therefore runs next.

t1's **defined priority** throughout this is 10. However, t1's **current priority** between 3a. and 4. is 12.

Note: receiving a message always changes *both* priorities.

Topics:

Processes and Threads

Processes

Threads

→ **Synchronization**

- mutexes
- condvars
- semaphores
- atomic operations

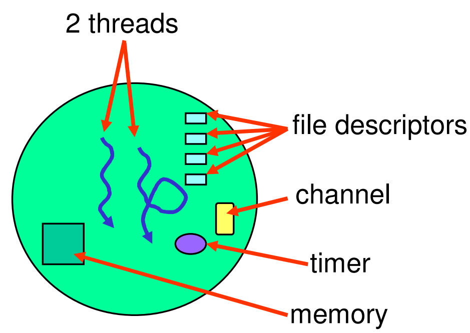
Conclusion

NOTES:

Synchronization - Resource Inheritance

Threads within a process share:

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



NOTES:

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...

NOTES:

These are problems with:

SYNCHRONIZATION

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

- mutexes,
- condvars,
- semaphores,
- atomic operations

NOTES:

Other synchronization tools we won't see:

- rwlocks:
  - 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

NOTES:

For rwlocks, look at the *pthread_rwlock*()* functions.

For once control, look at the *pthread_once*()* functions.

For thread local storage, look at the *pthread_key*()* functions.

Synchronization - Volatile

Any variables that are being shared:

- should be declared as **`volatile`**

```
volatile unsigned flags;
...
atomic_clr (&flags, A_FLAG);
```

- **`volatile`** is an ANSI C keyword that tells the compiler not to optimize the variable

Example:

The compiler may optimize the code by having it store the value in a register, and refer to the register all the time instead of going back to the memory. Meanwhile, the another thread's code would be accessing the memory!

NOTES:

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- semaphores
- atomic operations

Conclusion

NOTES:

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

NOTES:

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_trylock (pthread_mutex_t *);
pthread_mutex_unlock (pthread_mutex_t *);
```


NOTES:

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_destroy (&myMutex);
}
```

use default attributes

NOTES:

The function *pthread_mutex_destroy()* will most usually be called in designs which use structures that contain the mutex within the structure, for example:

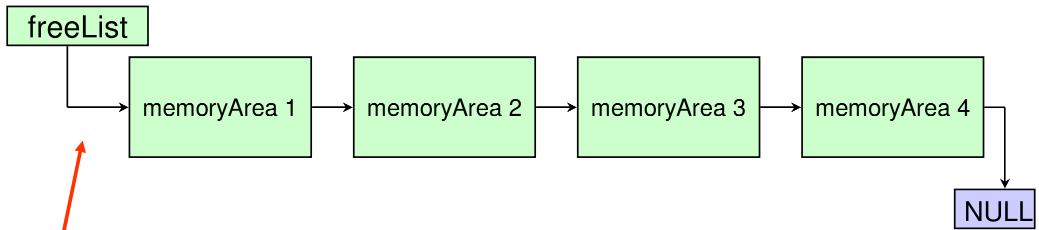
```
typedef struct {
    pthread_mutex_t     mutex;
    char                dataArea [64];
    int                 flags;
}   LockingDataStructure_t;
```

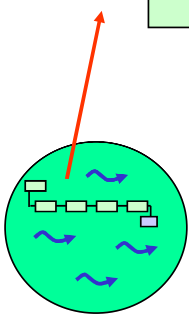
If the structure is one that is dynamically allocated and freed, just before it is freed the mutex should be destroyed.

Note that when a process dies, any mutexes locked by that process are automatically unlocked and destroyed.

Mutual Exclusion

Consider malloc():





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

NOTES:

Simplified malloc() source looks something like this:

```
void *
malloc (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);
   }
   …
}
```


NOTES:

The reason we said it was simplified is because the real library *malloc()* routine has to deal with all kinds of other issues, such as:

If it can’t find a block of the correct size, it has to try and split a bigger block into two pieces; one of the correct size, and the rest of the block, which is returned to the free pool,

If there is no more memory available to the process/thread, more can be allocated from the operating system. This memory then has to be added to the free pool.

However, for the purposes of our discussion here, the important code is reasonably similar to that shown.

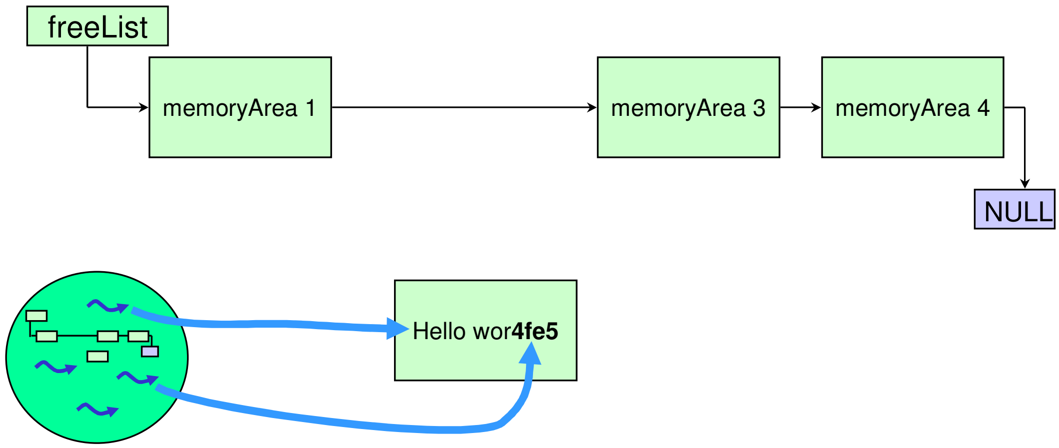
QNX SOFTWARE SYSTEMS

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

```
thread1 ()
{
    char    *data;

    data = malloc (64);
}

thread2 ()
{
    char    *other_data;

    other_data = malloc (64);
}
```


NOTES:

Something bad happens!



NOTES:

The problem is that one thread got the memory block from *malloc()*, but was then pre-empted by another thread, which retrieved the SAME memory block.

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...

NOTES:

Let's fix the **`malloc`** routine:

```
pthread_mutex_t                 malloc_mutex;

void *
malloc (int nbytes)
{
   …
   pthread_mutex_lock (&malloc_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 (&malloc_mutex);
     return (block);
   }
   pthread_mutex_unlock (&malloc_mutex);
   …
}
```

critical section

NOTES:

A critical section is defined as a section of code that, once processing has started, cannot be re-entered.

In this example, we have marked everything from the time that the *pthread_mutex_lock()* is called to the time that *pthread_mutex_unlock()* is called as a critical section. (We haven't yet shown how to initialize a mutex, we'll see that in a few slides).

Mutex Initialization

To explicitly initialize the mutex:

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

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

NOTES:

Mutex Initialization

A simple method for mutex initialization:

```
// static initialization of Mutex

pthread_mutex_t malloc_mutex =
  PTHREAD_MUTEX_INITIALIZER;

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

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

NOTES:

Assigning **`PTHREAD_MUTEX_INITIALIZER`** to the mutex doesn’t *really* initialize the mutex. It just marks it as “not in use” and “to be initialized by the kernel when someone first tries to use it”.

The one problem here is that you may get an **`EAGAIN`** return code from the FIRST lock attempt if the kernel is running low on resources. Therefore, you should explicitly test against this, and retry the operation.

In reality, though, if you do get the **`EAGAIN`** return code, then the system is running so low on resources that your application will most likely die due to unavailability of other resources…

Sharing Mutexes between Processes

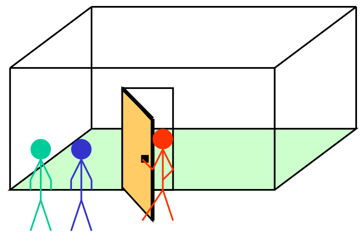
By default, mutexes cannot be shared between processes

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

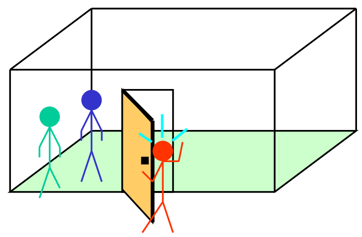
```
pthread_mutexattr_t mutex_attr;
pthread_mutex_t *mutex;
pthread_mutexattr_init( &mutex_attr );
pthread_mutexattr_setpshared( &mutex_attr,
  PTHREAD_PROCESS_SHARED);
mutex = (pthread_mutex_t *)shmem_ptr;
pthread_mutex_init( mutex, &mutex_attr );
```


NOTES:

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

NOTES:

You can visualize a mutex as a lock on a door. Several people want to get into the room "simultaneously". Only one person will get to the key that unlocks the door first. When that person gets the key, they unlock the room, go inside, and lock the door again, taking the key with them -- that way no one else can disturb them. When that person is done in the room (i.e. has executed the code in the critical section), that person leaves the room, and puts the key back into the lock, so that the next person can do the same thing. Since there is only one key, and the person holding the key locks the door as soon as they get into the room, they will have exclusive access to the resources in the room.

However, the room has no walls! There is nothing stopping someone from going around the door. This is the case when you have a critical section in your code but you forget to lock a mutex around that critical section.

Keep mutexes locked for short periods

- keep a mutex locked for as short a time as possible
- while one thread has the mutex locked, other threads that want the resource protected by the mutex have to wait
- 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...

NOTES:

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!

NOTES:

If the mutex can be acquired, this function executes in just a few instructions, avoiding a kernel call!

Note that on processors that implement a compare and exchange instruction, we use it. On ones that don't, we emulate the compare and exchange instruction. Even with emulation, uncontested mutex access is still very fast.

Since you often don't end up doing the kernel call, this means that some of your calls to *pthread_mutex_lock()* will not show up if you are doing tracing with the instrumented kernel and some of your calls will show up (those that do the kernel call).

Exercise:

- in your **`thread`** project:
- look at nomutex.c
  - run the program
- modify mutex.c to fix the problem
  - add a mutex to control access to the critical section
  - how does this affect performance?

NOTES:

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- semaphores
- atomic operations

Conclusion

NOTES:

Condvars

Consider a simple case where:

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

```
int state;

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

– condvars provide a mechanism for doing this

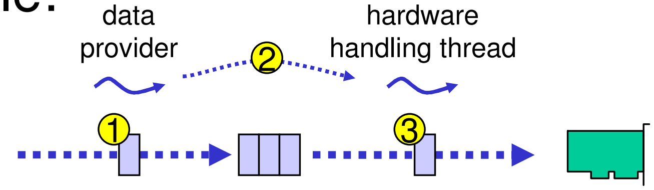
NOTES:

The condvar calls:

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


NOTES:

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

NOTES:

Condvars - Example

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);
}
```


NOTES:

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
```


NOTES:

Condvars - Under the covers

Let's zoom in on the "wait":

`pthread_cond_wait (&condvar, &mutex);` → **unlock**, **wait**, becomes **READY**, **lock**

unlock

- allows other threads to get access

wait

- this is the actual "waiting" part

becomes READY

- 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

NOTES:

This four step approach is required, so that as far as the user of the wait function is concerned, the mutex is always locked. It just so happens that it gets unlocked while the wait function is waiting so that other threads can gain access to the mutex controlling the variable.

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**)

NOTES:

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

NOTES:

Getting back to our door and lock analogy that we used for the mutex, we can easily incorporate condvars.

A number of people are waiting outside the room. This time, they want to know when something has changed inside the room. One person goes into the room (via the mutex), and makes a change. When that person leaves the room, he selects <u>one</u> person from the group waiting and tells her, "I have made a change" (via the function call *pthread_cond_signal()*). She then goes and tries to get into the room (via mutex) to see what the change was.

Note that *pthread_cond_signal()* signals only **<u>ONE</u>** thread, either the highest priority thread waiting on the condvar, or, if there are multiple threads at the high priority, the one that has been waiting the longest. *pthread_cond_broadcast()* (next slide) can be used to signal **<u>ALL</u>** waiting threads.

Condvars - Signalling vs broadcasting

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 READY, but only one can lock the mutex at a time so they will end up taking turns.

NOTES:

In this case, the person who made the change in the room announces to everyone that a change has been made. Everyone then tries to simultaneously get into the room to see what the change was. Because of the mutex, only one person will be able to get into the room at a time, again based on their priority and the length of time they have been waiting.

Condvars - Signalling vs broadcasting

QNX SOFTWARE SYSTEMS

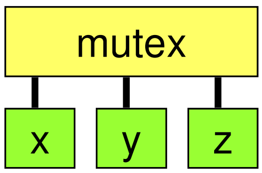
We use one over the other?

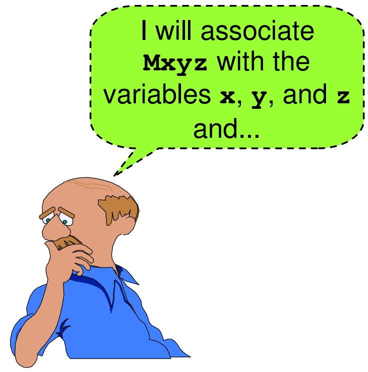
- use a signal if:
  - you have only one waiting thread, or
  - you need only one thread to do the work and you don't care which one
- use a broadcast if you have multiple threads and:
  - they all need to do something, or
  - they don't all need to do something but you don't know which one(s) to wake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

NOTES:

Let's examine condvars in a little more detail:

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





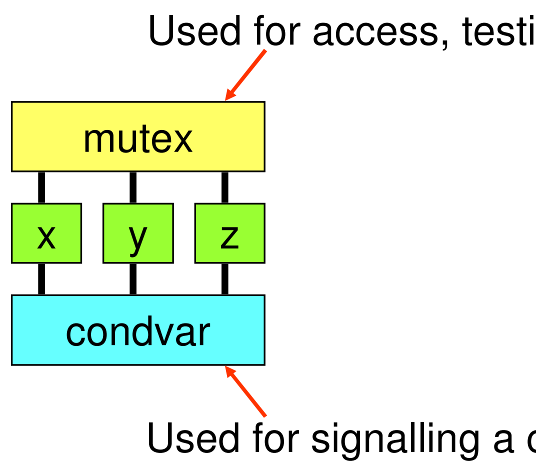
NOTES:

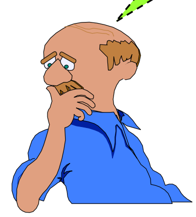
At design time, the association has to be made between the mutex that will be used, and the variables that it will be used with. This is so that various threads in the process can agree on which mutex to lock to test the associated variables.

Condvars

QNX SOFTWARE SYSTEMS

And, the programmer associates a condition variable as well:





NOTES:

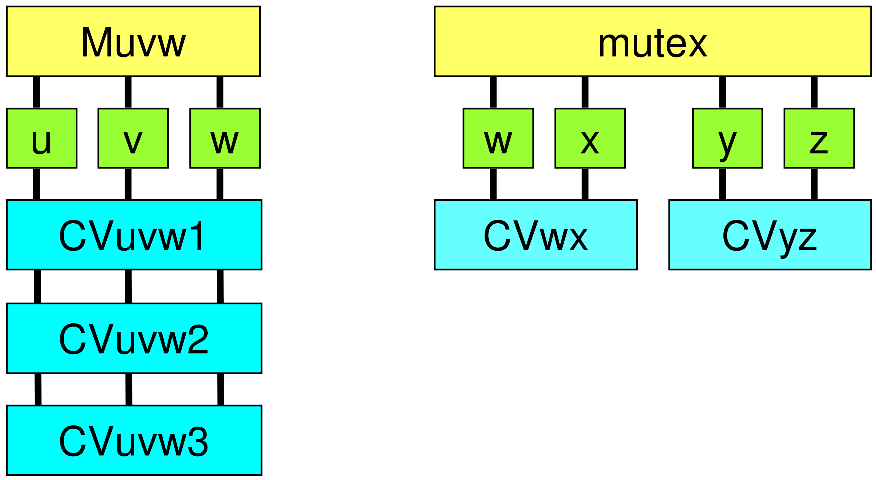
The mutex acts as a **door** to allow access to the variables. Only when the mutex has been acquired can we be assured that we have exclusive access to the variables; only when we have exclusive access can we reliably **test** these variables against whatever complex conditions we may have.

The condvar, on the other hand, functions as a **signaling interface**. We can wait for a condvar to be signalled, and we can cause a condvar to be signalled.

It is therefore the **combination** of mutex and condvar that allows us to **detect** changes (via the condvar) and safely **test** against our criteria (via the mutex).

Condvars

The association need not be one-to-one:



NOTES:

In the first example, different condvars are used to notify of different values of the same data fields, while in the 2nd example different condvars are used to flag changes in different data areas.

Sharing condvars between processes

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 );
```


NOTES:

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);
}
```


NOTES:

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);
}
```

Processes, Threads & Synchronization 2010/06/22 R11

NOTES:

Note that the producer / consumer example used *pthread_cond_signal()* to tell the other thread that it should check its condition variable. *pthread_cond_signal()* only signals ONE thread -- the highest priority thread. In the exercise coming up, we will need to signal ALL of the threads to get them to check their variables. This is done via the *pthread_cond_broadcast()* function, which takes the same parameter -- the condition variable.

This is the prodcons.c sample from the threads project (modified to fit the slide).

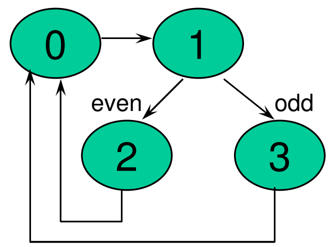
EXERCISE

Exercise:

– in your **threads** 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



NOTES:

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- semaphores
- atomic operations

Conclusion

NOTES:

Semaphores for access control:

administration

unnamed semaphores

```
sem_init (sem_t *semaphore, int pshared, unsigned int val);
sem_destroy (sem_t *semaphore);

sem_t *sem_open (char *name, int oflag, [int sharing,
                 unsigned int val]);
sem_close (sem_t *semaphore);
sem_unlink (char *name);
```

named semaphores

usage:

```
sem_post (sem_t *semaphore);
sem_trywait (sem_t *semaphore);
sem_wait (sem_t *semaphore);
sem_getvalue (sem_t *semaphore, int *value);
```


NOTES:

QNX Neutrino provides counting semaphores. When the semaphore is created (via the *sem_open()* or *sem_init()* call) the **`val`** parameter specifies a count. 0 means “none are available”. When *sem_wait()* is called, it checks this count value and if greater than zero, decrements the count and returns success, otherwise the call blocks. A *sem_post()* increments this count value, possibly unblocking one waiting thread.

There are two basic variants of semaphores:

> Named semaphores are created with *sem_open()*, closed with *sem_close()*, and destroyed with *sem_unlink()*. These allow processes to access a common semaphore by agreeing on a name; no other communication has to happen between the processes.
>
> Unnamed semaphores are created with *sem_init()*, and destroyed with *sem_destroy()*. These are useful either between threads of a process, where the semaphore structure (type **`sem_t`**) is available via common memory, or between processes where the semaphore structure is mapped into a shared memory segment.

Regardless of the type of semaphore in use, the *sem_post()*, *sem_trywait()*, *sem_wait()*, and *sem_getvalue()* functions are applicable.

Unnamed vs named semaphores

- with unnamed, *sem_post()* and *sem_wait()* call the semaphore kernel calls directly whereas...
- with named semaphores, *sem_post()* and *sem_wait()* send messages to **`procnto`**
- so unnamed semaphores are faster than named semaphores
- if you are using semaphores within a multithreaded process, unnamed semaphores are easy since the semaphore can simply be a global variable

NOTES:

A sample semaphore program called semex.c is in your thread directory.

Using Semaphores

Semaphores can be used in two ways:

– as a broken mutex

thread 1:

```
sem_wait()
// access data
sem_post()
```

thread 2:

```
sem_wait()
// access data
sem_post()
```

– priority inversions are possible because semaphores don't have ownership

– DON'T DO THIS

- use mutexes instead
- if you have existing code like this, replace it with mutexes

NOTES:

Using Semaphores

Semaphores can be used in two ways:

- as a dispatch mechanism:

thread 1:
```
while(1)
  //prepare data
  sem_post()
```

thread 2:
```
while (1)
  sem_wait()
  //use data
```

- this overlaps with what condvars do
- some problems are a bit easier to solve with semaphores than condvars
- condvars are usually more efficient than semaphores

NOTES:

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- semaphores
- atomic operations

Conclusion

NOTES:

Atomic Operations

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

NOTES:

"Correctly" means that if two code paths both do an atomic operation on the same variable, then one operation will be done as a whole (start to finish) before the 2nd one is done.

Topics:

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

NOTES:

Conclusion

QNX SOFTWARE SYSTEMS

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

NOTES:

Reference

David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997, ISBN 0-201-63392-2

Kay A. Robbins and Steven Robbins, *Practical Unix Programming*, Prentice Hall, 1996, ISBN 0-13-443706-3

Bill O. Gallmeister, *POSIX.4 - Programming for the Real World*, O'Reilly & Associates, 1995

Rob Krten, *Getting Started with QNX Neutrino 2*, Parse Software Devices, 1999, ISBN 0-9682501-1-4

NOTES: