Advanced C and System Programming

Anandkumar

Threads

- Threads are mechanisms to do more than one job at a time.

- Threads are finer-grained units of execution.

- Threads, unlike processes, share the same address space and other resources.

- POSIX standard thread API is not included in standard C library, they are in *libpthread.so.*

- In Linux, threads are handled by LWPs.

Threads

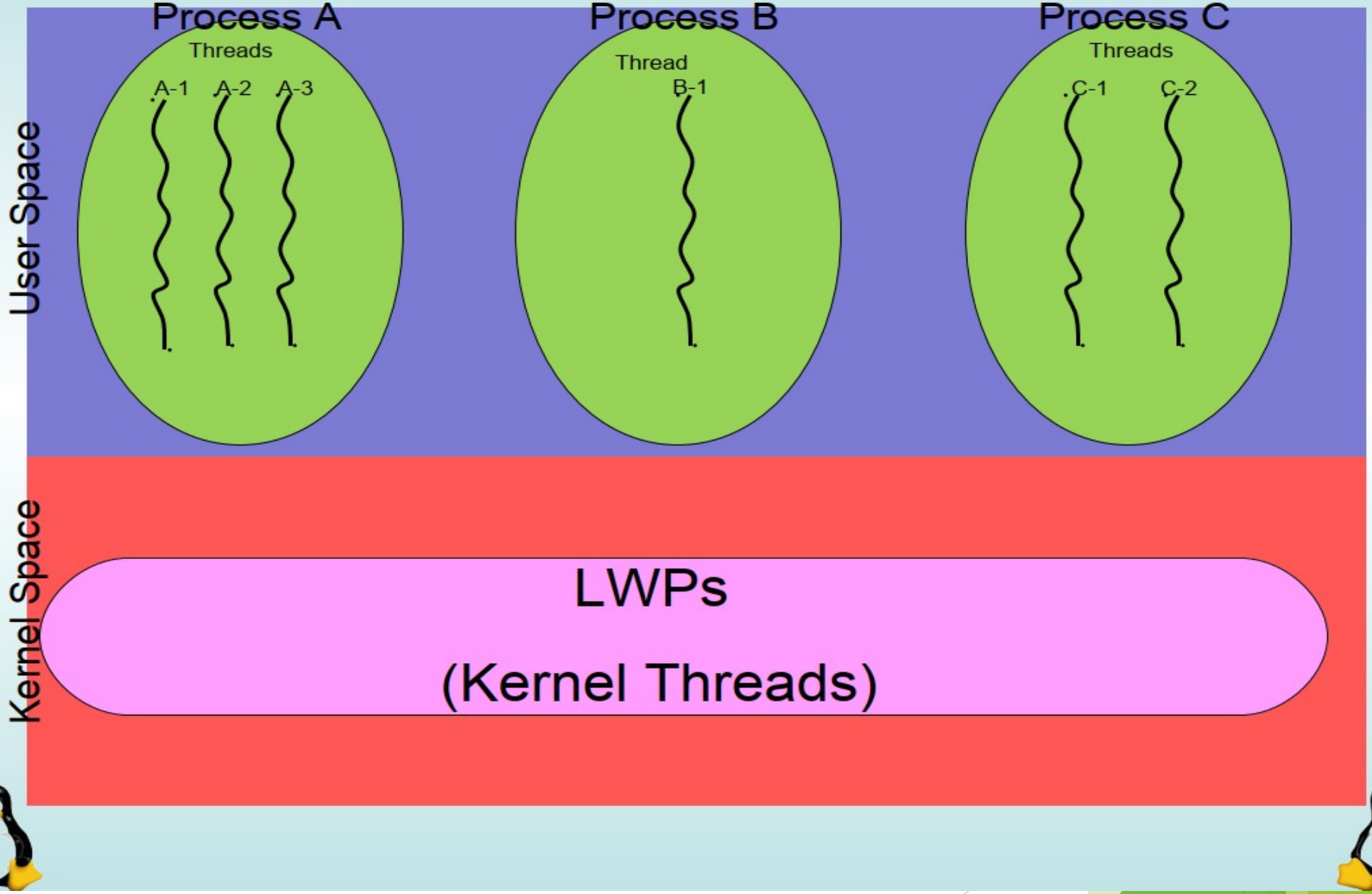
- A Thread is an independent stream of instructions that can be schedule to run as such by the OS.
- Think of a thread as a “procedure” that runs independently from its main program.
- Multi-threaded programs are where several procedures are able to be scheduled to run simultaneously and/or independently by the OS.

Threads (cont)

Threads only duplicate the essential resources it needs to be independently schedulable.

A thread will die if the parent process dies.

A thread is “lightweight” because most of the overhead has already been accomplished through the creation of the process.


Process A
Threads
A-1
A-2
A-3
Process B
Thread
B-1
Process C
Threads
C-1
C-2
User Space
Kernel Space
LWPs
(Kernel Threads)

POSIX Threads (PThreads)

For UNIX systems, implementations of threads that adhere to the IEEE POSIX 1003.1c standard are Pthreads.

Pthreads are C language programming types defined in the pthread.h header/include file.

Why Use Pthreads

The primary motivation behind Pthreads is improving program performance.

Can be created with much less OS overhead.

Needs fewer system resources to run.

View comparison of forking processes to using a pthreads_create subroutine.

Threads vs Forks

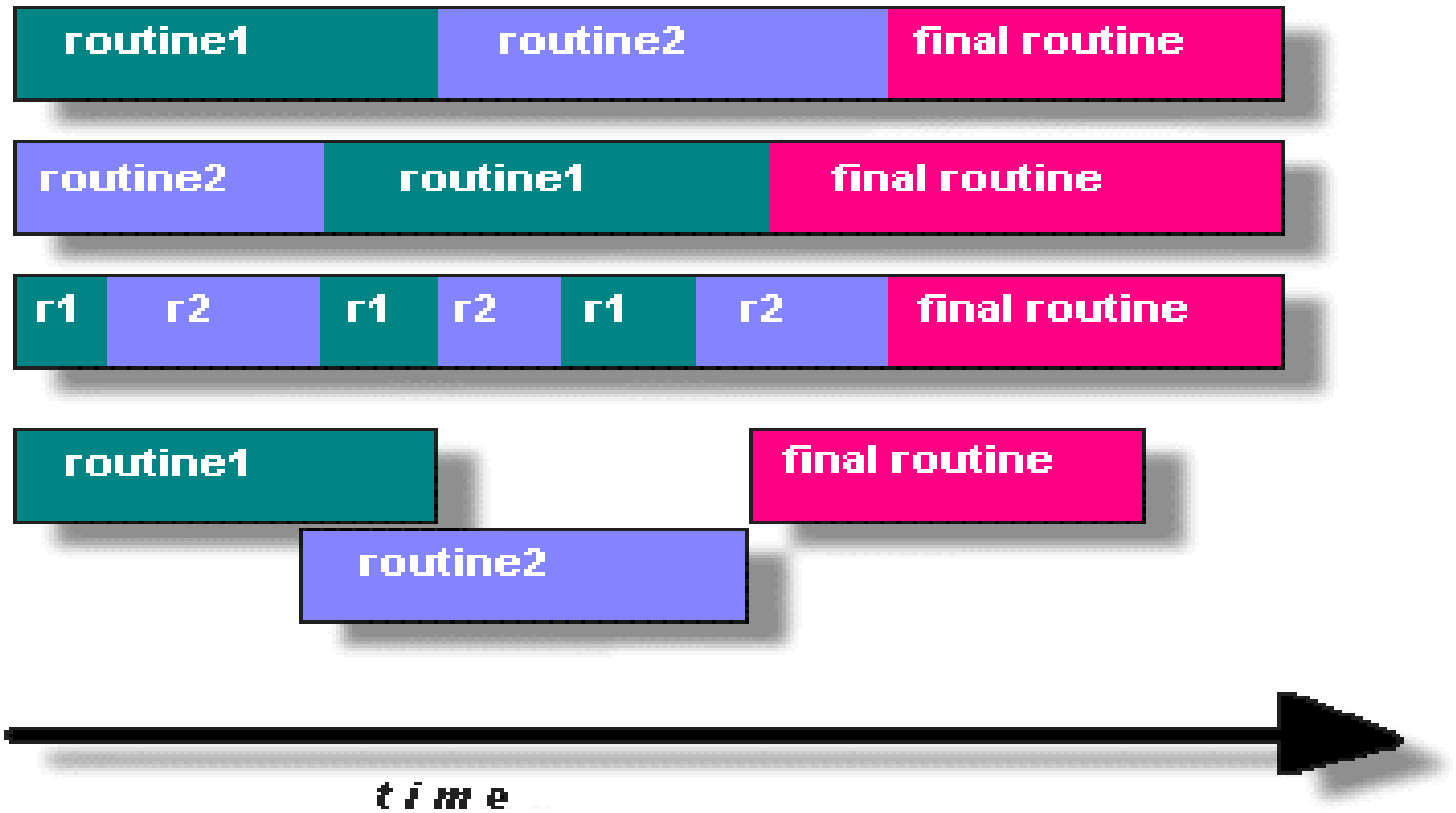
PLATFORM	fork()			pthread_create()		
	REAL	USER	SYSTEM	REAL	USER	SYSTEM
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Designing Pthreads Programs

Pthreads are best used with programs that can be organized into discrete, independent tasks which can execute concurrently.

Example: routine 1 and routine 2 can be interchanged, interleaved and/or overlapped in real time.

Candidates for Pthreads



Designing Pthreads (cont)

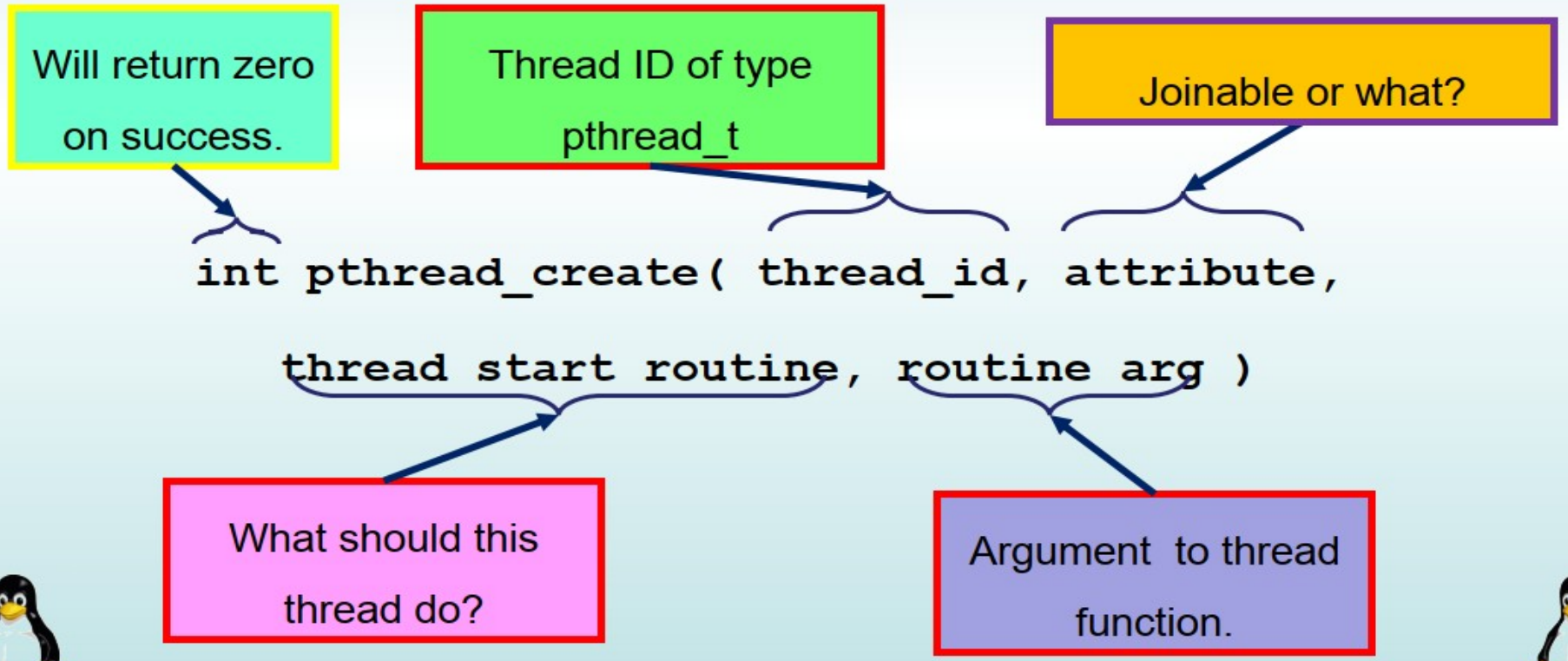
Common models for threaded programs:

- Manager/Worker: manager assigns work to other threads, the workers. Manager handles input and hands out the work to the other tasks.
- Pipeline: task is broken into a series of suboperations, each handled in series but concurrently, by a different thread.

Creating threads

- Like processes, each thread has its own Thread-ID of type *pthread_t.*

- You can create a thread bye calling the *pthread_create* function.



Creating threads

- *pthread_create* returns immediately and the specified thread will do its job separately.

- If one of the threads in a program, call *exec* the whole process image will be replaced.

- The argument passed to the thread routine is a *void* * .

- You can pass more data in a structure of type *void* *.

Pthread Management – Creating Threads

The main() method comprises a single, default thread.

pthread_create() creates a new thread and makes it executable.

The maximum number of threads that may be created by a process in implementation dependent.

Once created, threads are peers, and may create other threads.

Joining threads

- You can wait for a thread to finish its job using *pthread_join*.

- *pthead_join* is something similar to *wait* function in processes.

- Using *pthread_join*, you can also take the return value of a thread.

- A thread, can not call *pthread_join* to wait for itself, you can use *pthread_self* function to get the TID of running thread and deciding what to do.

Joining threads

- Like processes, you can wait for a thread to finish its job…

```
int pthread_join( pthread_t thread_id, void ** return_value )
```

Will return zero on success.

Thread ID which you want to wait for.

The return value of thread will be put here.

Pthread Management – Terminating Threads

Several ways to terminate a thread:

- The thread is complete and returns
- The pthread_exit() method is called
- The pthread_cancel() method is invoked
- The exit() method is called

The pthread_exit() routine is called after a thread has completed its work and it no longer is required to exist.

Terminating Threads (cont)

If the main program finishes before the thread(s) do, the other threads will continue to execute if a pthread_exit() method exists.

The pthread_exit() method does not close files; any files opened inside the thread will remain open, so cleanup must be kept in mind.

Pthread Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   int tid; tid = (int)threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL);
}
```

Pthread Example

```
int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0; t<NUM_THREADS; t++)
   {
      printf("In main: creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc)
     {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Pthread Example - Output

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
```

Thread attributes

- Second parameter in *pthread_create* is the thread attribute.

- Most useful attribute of a thread is *joinability*.

- If a thread is *joinable*, it is not automatically cleaned up.

- To clean up a *joinable* like a child process, you should call *pthread_join* .

- A *detached* thread, is automatically cleaned up.

- A joinable thread may be turned into a detached one, but can not be made joinable again.

- Using *pthread_detach* you can turn a joinable thread into detached.

Thread attributes

- If you do not clean up the joinable thread, it will become something like zombie.

- To assign an attribute to a thread, you should:
  - Create a *pthread_attr_t* object.
  - Call *pthread_attr_init* to initialize the attribute object.
  - Modify the attributes.
  - Pass a pointer to *pthread_create.*
  - Call *pthread_attr_destroy* to release the attribute object.

Thread cancelation

- A thread might be terminated by finishing its job or calling *pthread_exit* or by a request from another thread.

- The latter case is called “Thread Cancelation”.

- You can cancel a thread using *pthread_cancel*.

- If the canceled thread is not detached, you should join it after cancelation, otherwise it will become zombie.

- You can disable cancelation of a thread using *ptherad_setcancelstate()*.

Thread cancelation

- There are two cancel state:

- **PTHREAD_CANCEL_ASYNCHRONOUS:** Asynchronously cancelable (cancel at any point of execution)

- **PTHREAD_CANCEL_DEFERRED:** Synchronously cancelable (thread checks for cancellation requests)

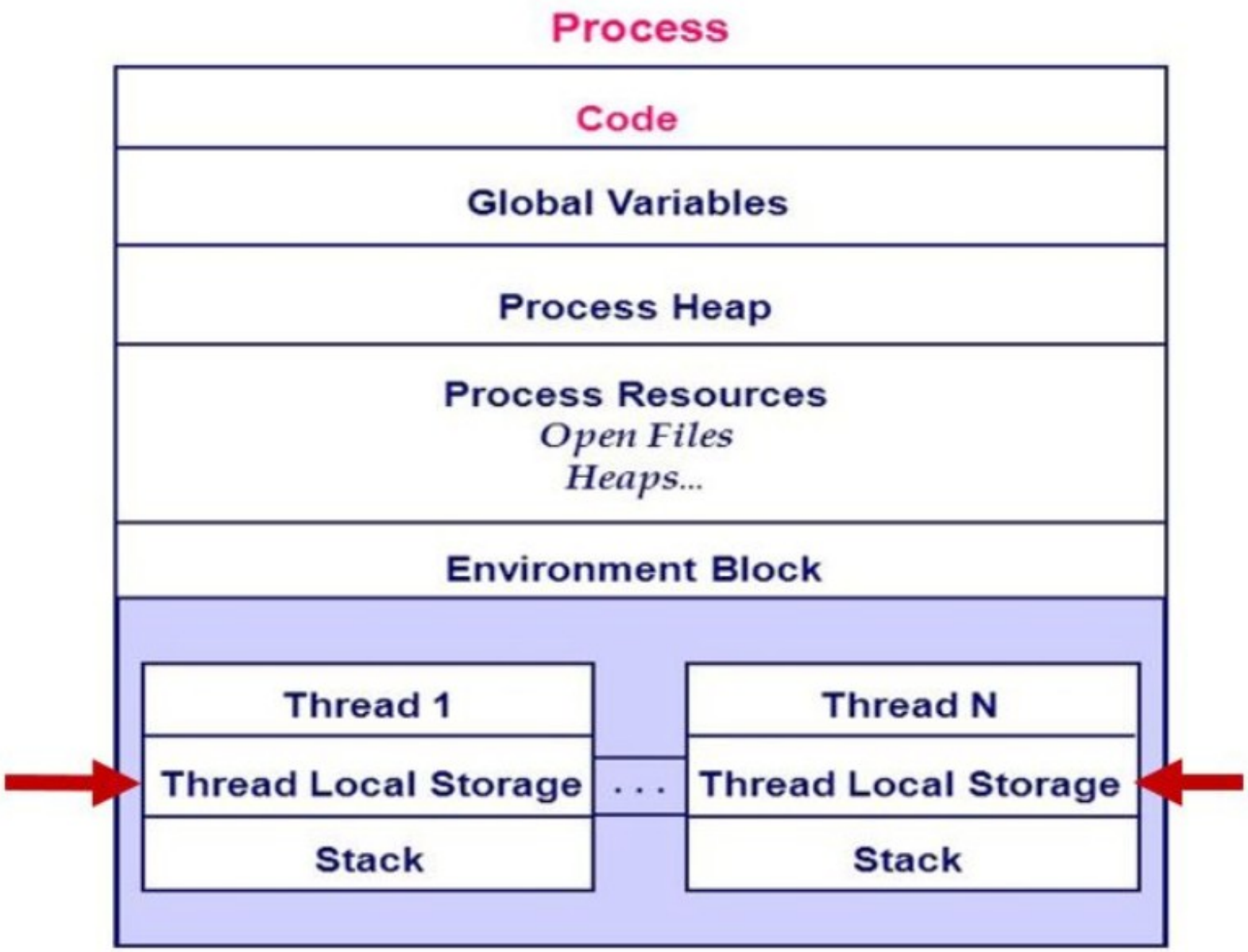
- There are two cancelation types:

- **PTHREAD_CANCEL_DISABLE** and **PTHREAD_CANCEL_ENABLE.**

- It's a good idea to set the state to *Uncancelable* when entering critical section…

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread


Process
Code
Global Variables
Process Heap
Process Resources
Open Files
Heaps...
Environment Block
Thread 1
Thread Local Storage
Stack
. . .
Thread N
Thread Local Storage
Stack

Critical Section

- The ultimate cause of most bugs involving threads is that they are accessing the same data at the same time.

- The section of code which is responsible to access the shared data, is called *Critical Section* .

- A critical section is part of code that should be executed completely or not at all (a thread should not be interrupted when it is in this section)

- If you do not protect the *Critical Section*, your program might crash because of *Race Condition.*

Race Condition

- Race Condition is a condition in which threads are racing each other to change the same data structure.

- Because there is no way to know when the system scheduler will interrupt one thread and execute the other one, the buggy program may crash once and finish regularly next time.

- To eliminate race conditions, you need a way to make operations *atomic* (uninterruptible).

Synchronization Primatives

- Counting Semaphores
  - Permit a limited number of threads to execute a section of the code
- Binary Semaphores - Mutexes
  - Permit only one thread to execute a section of the code
- Condition Variables
  - Communicate information about the state of shared data

POSIX Semaphores

- Named Semaphores
  - Provides synchronization between unrelated process and related process as well as between threads
  - Kernel persistence
  - System-wide and limited in number
  - Uses **`sem_open`**
- Unnamed Semaphores
  - Provides synchronization between threads and between related processes
  - Thread-shared or process-shared
  - Uses **`sem_init`**

POSIX Semaphores

Data type

- Semaphore is a variable of type **`sem_t`**

Include **`<semaphore.h>`**

Atomic Operations

```
int sem_init(sem_t *sem, int pshared,
  unsigned value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
```

Unnamed Semaphores

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

Initialize an unnamed semaph

You cannot make a copy of a semaphore variable!!!

Returns

0 on success

-1 on failure, sets **errno**

Parameters

sem:

Target semaphore

pshared:

0: only threads of the creating process can use the semaphore

Non-0: other processes can use the semaphore

value:

Initial value of the semaphore

Sharing Semaphores

Sharing semaphores between threads within a process is easy, use **`pshared==0`**

A non-zero **`pshared`** allows any process that can access the semaphore to use it

- Places the semaphore in the global (OS) environment
- Forking a process creates copies of any semaphore it has
  - Note: unnamed semaphores are not shared across

sem_init can fail

On failure

sem_init returns -1 and sets **errno**

errno	cause
EINVAL	**Value > sem_value_max**
ENOSPC	Resources exhausted
EPERM	Insufficient privileges

```
sem_t semA;

if (sem_init(&semA, 0, 1) == -1)
  perror("Failed to initialize semaphore
   semA");
```

Semaphore Operations

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

Destroy an semaphore

Returns

- 0 on success
- -1 on failure, sets **errno**

Parameters

- **sem**:

  Target semaphore

Notes

- Can destroy a **sem_t** only once
- Destroying a destroyed semaphore gives undefined results
- Destroying a semaphore on which a thread is blocked gives undefined results

Semaphore Operations

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

Unlock a semaphore - same as signal

Returns

0 on success

-1 on failure, sets **errno** (**== EINVAL** if semaphore doesn't exist)

Parameters

sem:

Target semaphore

sem > 0: no threads were blocked on this semaphore, the semaphore value is incremented

sem == 0: one blocked thread will be allowed to run

Semaphore Operations

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

Lock a semaphore

Blocks if semaphore value is zero

Returns

0 on success

-1 on failure, sets **errno** (**== EINTR** if interrupted by a signal)

Parameters

sem:

Target semaphore

sem > 0: thread acquires lock

sem == 0: thread blocks

Semaphore Operations

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

Test a semaphore's current condition

- Does not block

Returns

- 0 on success
- -1 on failure, sets **errno** (**== AGAIN** if semaphore already locked)

Parameters

- **sem**:
  - Target semaphore
  - sem > 0: thread acquires lock
  - sem == 0: thread returns

Pthread Mutex

States

- Locked
  - Some thread holds the mutex
- Unlocked
  - No thread holds the mutex

When several threads compete

- One wins
- The rest block
  - Queue of blocked threads

Mutex Variables

A typical sequence in the use of a mutex

1. Create and initialize **mutex**
2. Several threads attempt to lock **mutex**
3. Only one succeeds and now owns **mutex**
4. The owner performs some set of actions
5. The owner unlocks **mutex**
6. Another thread acquires **mutex** and repeats the process
7. Finally **mutex** is destroyed

Creating a mutex

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
   const pthread_mutexattr_t *attr);
```

Initialize a pthread mutex: the mutex is initially unlocked

Returns

0 on success

Error number on failure

`EAGAIN:` The system lacked the necessary resources; `ENOMEM:` Insufficient memory ; `EPERM:` Caller does not have privileges; `EBUSY:` An attempt to re-initialise a mutex; `EINVAL:` The value specified by attr is invalid

Parameters

`mutex`: Target mutex

`attr`:

NULL: the default mutex attributes are used

Non-NULL: initializes with specified attributes

Creating a mutex

Default attributes

- Use **`PTHREAD_MUTEX_INITIALIZER`**
  - Statically allocated
  - Equivalent to dynamic initialization by a call to **`pthread_mutex_init()`** with parameter **`attr`** specified as NULL
  - No error checks are performed

Destroying a mutex

```
#include <pthread.h>
int
   pthread_mutex_destroy(pthread_mutex_t
   *mutex);
```

Destroy a pthread mutex

Returns

0 on success

Error number on failure

`EBUSY:` An attempt to re-initialise a mutex; `EINVAL:` The value specified by attr is invalid

Parameters

`mutex`: Target mutex

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);
```

Returns

- 0 on success
- Error number on failure
  - EBUSY: already locked; EINVAL: Not an initialised mutex; EDEADLK: The current thread already owns the mutex; EPERM: The current thread does not own the mutex

Simple Example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

static pthread_mutex_t my_lock =
   PTHREAD_MUTEX_INITIALIZER;

void *mythread(void *ptr) {
   long int i,j;
   while (1) {

     pthread_mutex_lock (&my_lock);

     for (i=0; i<10; i++) {
       printf ("Thread %d\n", int) ptr);
       for (j=0; j<50000000; j++);
     }
```

```
int main (int argc, char *argv[]) {
   pthread_t thread[2];

   pthread_create(&thread[0], NULL,
   mythread, (void *)0);

   pthread_create(&thread[1], NULL,
   mythread, (void *)1);

   getchar();
}
```

Condition Variables

- Used to communicate information about the state of shared data
  - Execution of code depends on the state of
    - A data structure or
    - Another running thread
- Allows threads to synchronize based upon the actual value of data
- Without condition variables
  - Threads continually poll to check if the condition is met

Condition Variables

- Signaling, not mutual exclusion
  - A mutex is needed to synchronize access to the shared data
- Each condition variable is associated with a single mutex
  - Wait atomically unlocks the mutex and blocks the thread
  - Signal awakens a blocked thread

Creating a Condition Variable

Similar to pthread mutexes

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

int pthread_cond_destroy(pthread_cond_t
  *cond);


pthread_cond_t cond =
  PTHREAD_COND_INITIALIZER;
```

Using a Condition Variable

Waiting

- Block on a condition variable.
- Called with **mutex** locked by the calling thread
- Atomically release **mutex** and cause the calling thread to block on the condition variable
- On return, **mutex** is locked again

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

Using a Condition Variable

Signaling

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

unblocks at least one of the blocked threads

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

unblocks all of the blocked threads

Signals are not saved

Must have a thread waiting for the signal or it will be lost

Spinlock

Spin locks are a low-level synchronization mechanism suitable primarily for use on shared memory multiprocessors. When the calling thread requests a spin lock that is already held by another thread, the second thread spins in a loop to test if the lock has become available. When the lock is obtained, it should be held only for a short time, as the spinning wastes processor cycles. Callers should unlock spin locks before calling sleep operations to enable other threads to obtain the lock.

Spinlock

`pthread_spin_init()` Syntax

```
int  pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int pshared;
int ret;

/* initialize a spin lock */
ret = pthread_spin_init(&lock, pshared);
```

The *pshared* attribute has one of the following values:

`PTHREAD_PROCESS_SHARED`

Description: Permits a spin lock to be operated on by any thread that has access to the memory where the spin lock is allocated. Operation on the lock is permitted even if the lock is allocated in memory that is shared by multiple processes.

`PTHREAD_PROCESS_PRIVATE`

Description: Permits a spin lock to be operated upon only by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined. The default value of the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`.

Spinlock

pthread_spin_lock() Syntax

```
int  pthread_spin_lock(pthread_spinlock_t *lock);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_ spin_lock(&lock); /* lock the spinlock */
```

pthread_spin_unlock() Syntax

```
int  pthread_spin_unlock(pthread_spinlock_t *lock);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_spin_unlock(&lock); /* spinlock is unlocked */
```

pthread_spin_destroy() Syntax

```
int  pthread_spin_destroy(pthread_spinlock_t *lock);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_spin_destroy(&lock); /* spinlock is destroyed */
```

RW lock

Operation	Related Function Description
Initialize a read-write lock	`pthread_rwlock_init`
Read lock on read-write lock	`pthread_rwlock_rdlock`
Read lock with a nonblocking read-write lock	`pthread_rwlock_tryrdlock`
Write lock on read-write lock	`pthread_rwlock_wrlock`
Write lock with a nonblocking read-write lock	`pthread_rwlock_trywrlock`
Unlock a read-write lock	`pthread_rwlock_unlock`
Destroy a read-write lock	`pthread_rwlock_destroy`

RW lock

pthread_rwlock_init Syntax

```
#include <pthread.h>

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

pthread_rwlock_t  rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

pthread_rwlock_wrlock Syntax

```
#include <pthread.h>

int  pthread_rwlock_wrlock(pthread_rwlock_t *rwlock );
```

pthread_rwlock_destroy Syntax

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t **rwlock);
```

pthread_rwlock_rdlock Syntax

```
#include <pthread.h>

int  pthread_rwlock_rdlock(pthread_rwlock_t *rwlock );
```

pthread_rwlock_unlock Syntax

```
#include <pthread.h>

int pthread_rwlock_unlock (pthread_rwlock_t  *rwlock);
```

Thread priority

pthread_attr_setschedparam()

Set a thread's scheduling parameters attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedparam(
            pthread_attr_t * attr,
            const struct sched_param * param );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see *pthread_attr_init()*.

param

A pointer to a `sched_param` structure that defines the thread's scheduling parameters.

Thread priority

pthread_attr_getschedparam()

Get thread scheduling parameters attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedparam(
                    const pthread_attr_t * attr,
                    struct sched_param * param );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see *pthread_attr_init()*.

param

A pointer to a `sched_param` structure where the function can store the current scheduling parameters.

Thread Safe

```
...
...
...

char arr[10];
int index=0;

int func(char c)
{
     int i=0;
     if(index >= sizeof(arr))
     {
          printf("\n No storage\n");
          return -1;
     }
     arr[index] = c;
     index++;
     return index;
}

...
...
...
```

Thread Safe

```
char arr[10];
int index=0;

int func(char c)
{
     int i=0;
     if(index >= sizeof(arr))
     {
          printf("\n No storage\n");
          return -1;
     }

     /* ...
        Lock a mutex here
        ...
     */

     arr[index] = c;
     index++;

     /* ...
        unlock the mutex here
        ...
     */

     return index;
}
```

Producers Consumers Systems

One system produce items that will be used by other system

Examples

- shared printer, the printer here acts the consumer, and the computers that produce the documents to be printed are the consumers.
- Sensors network, where the sensors here the producers, and the base stations (sink) are the producers.

Producer Consumer Problem

The producer-consumer problem illustrates the need for synchronization in systems where many processes share a resource. In the problem, two processes share a fixed-size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently.

It is also called bounded buffer problem

Producer

```
While(Items_number ==buffer size)
        ;   //waiting since the buffer is full

      Buffer[i]=next_produced_item;
       i=(i+1)%Buffer_size;
      Items_number++;
```

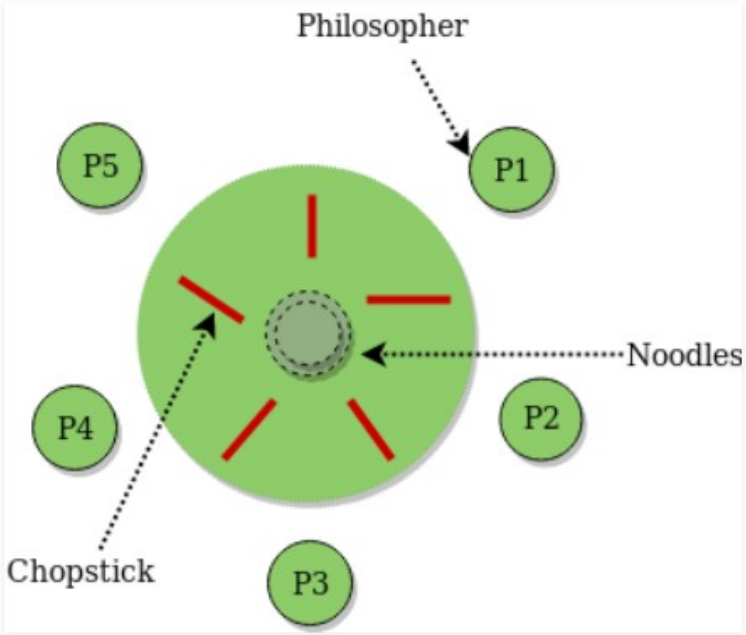
Consumer

```
while (Items_number == 0)
        ; // do nothing since the buffer is empty

            Consumed _item= buffer[j];
              j = (j + 1) % Buffer_size;
             Items_number--;
```

Dining Philosopher Problem

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
   {  THINK;
      PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
      EAT;
      PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
   }
```

There are three states of the philosopher: **THINKING, HUNGRY, and EATING**. Here there are two semaphores: Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

Priority Inversion Problem

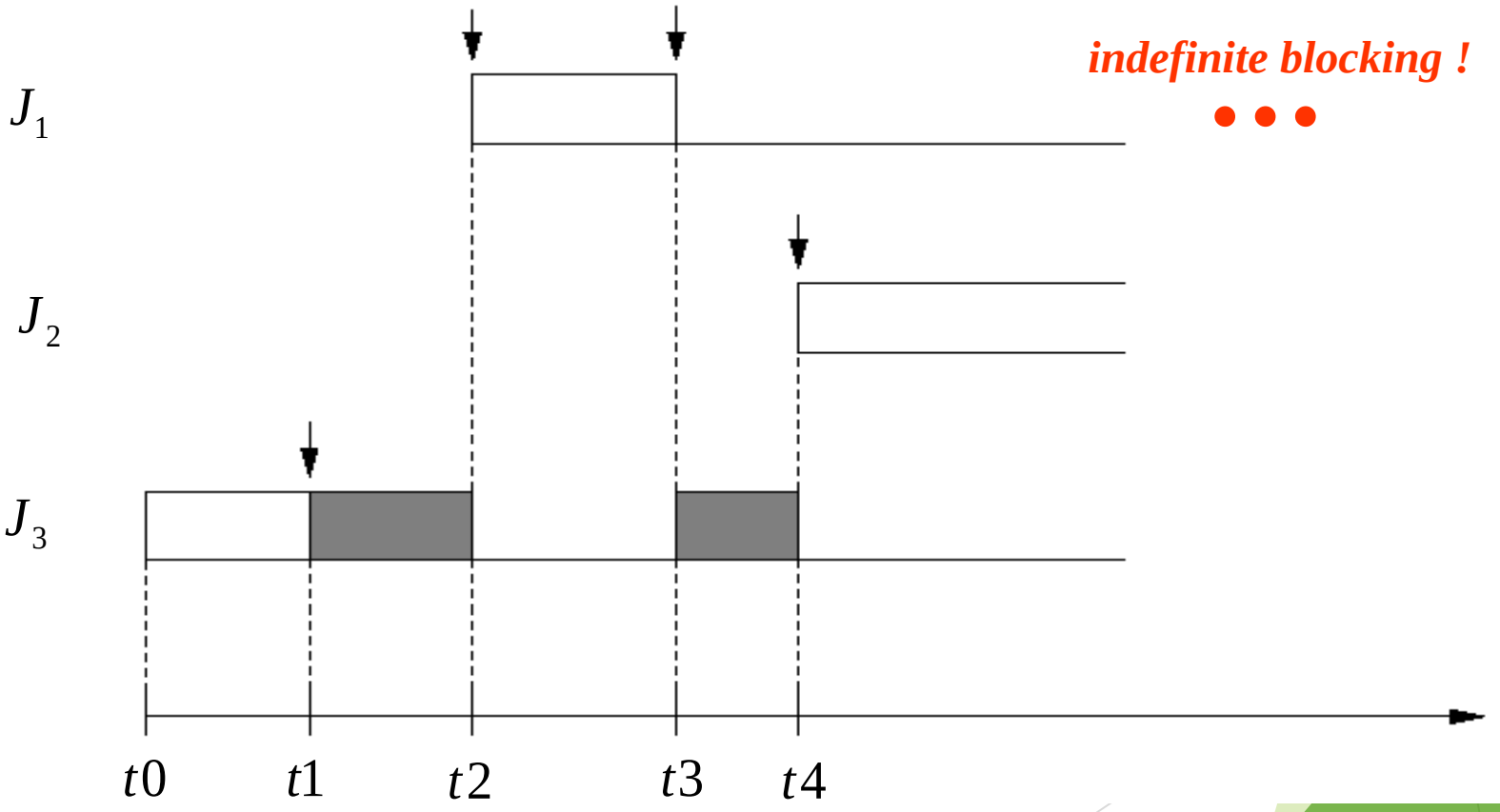
Priority inversion

- Phenomenon where a higher priority job is *block*ed by lower priority jobs

Indefinite priority inversion

- Occurs when a task of medium priority preempts a task of lower priority which is blocking a task of higher priority.

Indefinite Priority Inversion



indefinite blocking !

● ● ●

Process Scheduling

Although Linux is a preemptively multitasked operating system, it also provides a system call that allows processes to explicitly yield execution and instruct the scheduler to select a new process for execution:

```
#include <sched.h>

int sched_yield (void);
```

A call to `sched_yield( )` results in suspension of the currently running process, after which the process scheduler selects a new process to run, in the same manner as if the kernel had itself preempted the currently running process in favor of executing a new process. Note that if no other runnable process exists, which is often the case, the yielding process will immediately resume execution. Because of this uncertainty, coupled with the general belief that there are generally better choices, use of this system call is not common.

Process Scheduling

Linux provides several system calls for retrieving and setting a process' nice value. The simplest is `nice( )`:

```
#include <unistd.h>

int nice (int inc);
```

A successful call to `nice( )` increments a process' nice value by `inc`, and returns the newly updated value. Only a process with the `CAP_SYS_NICE` capability (effectively, processes owned by root) may provide a negative value for `inc`, decreasing its nice value, and thereby increasing its priority. Consequently, nonroot processes may only lower their priorities (by increasing their nice values).

On error, `nice( )` returns `-1`. However, because `nice( )` returns the new nice value, `-1` is also a successful return value. To differentiate between success and failure, you can zero out `errno` before invocation, and subsequently check its value. For example:

Process Scheduling

A preferable solution is to use the `getpriority( )` and `setpriority( )` system calls, which allow more control, but are more complex in operation:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

These calls operate on the process, process group, or user, as specified by `which` and `who`. The value of `which` must be one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, in which case `who` specifies a process ID, process group ID, or user ID, respectively. If `who` is `0`, the call operates on the current process ID, process group ID, or user ID, respectively.

A call to `getpriority( )` returns the highest priority (lowest numerical nice value) of any of the specified processes. A call to `setpriority( )` sets the priority of all specified processes to `prio`. As with `nice( )`, only a process possessing `CAP_SYS_NICE` may raise a process' priority (lower the numerical nice value). Further, only a process with this capability can raise or lower the priority of a process not owned by the invoking user.

Like `nice( )`, `getpriority( )` returns `-1` on error. As this is also a successful return value, programmers should clear `errno` before invocation if they want to handle error conditions. Calls to `setpriority( )` have no such problem; `setpriority( )` always returns `0` on success, and `-1` on error.

Process Scheduling

Processes can manipulate the Linux scheduling policy via `sched_getscheduler( )` and `sched_setscheduler( )`:

```
#include <sched.h>

struct sched_param {
        /* ... */
        int sched_priority;
        /* ... */


};

int sched_getscheduler (pid_t pid);

int sched_setscheduler (pid_t pid,
                        int policy,
                        const struct sched_param *sp);
```

A successful call to `sched_getscheduler( )` returns the scheduling policy of the process represented by `pid`. If `pid` is `0`, the call returns the invoking process' scheduling policy. An integer defined in `<sched.h>` represents the scheduling policy: the first in, first out policy is `SCHED_FIFO`; the round-robin policy is `SCHED_RR`; and the normal policy is `SCHED_OTHER`. On error, the call returns `-1` (which is never a valid scheduling policy), and `errno` is set as appropriate.

Temporary files

Quite often, we need temporary files in our programs. Some intermediate data needs to be stored and the file can be discarded when the process terminates. There are functions and command to create temporary files in Linux. The mkstemp function creates a temporary file and returns a file descriptor. The mkdtemp function creates a temporary directory. The tmpfile function creates a temporary file and returns a file pointer. The mktemp command is for creating a

mkstemp

```
#include <stdlib.h>

int mkstemp (char *template);
```

`mkstemp` creates a unique temporary file, opens it and returns a file descriptor to it. The file is created from the parameter, *template*, whose last six characters must be "XXXXXX". `mkstemp` modifies the six X's to get a unique filename. Since the *template* is modified, it should not be a constant string. It should be a null terminated character array. The file is created with 0600 permissions, which means read and write permissions for the owner and none for the group and others. The file is opened with O_EXCL flag, which guarantees that the caller process has created the file.

mkdtemp

```
#include <stdlib.h>

char *mkdtemp (char *template);
```

The `mkdtemp` function creates a temporary directory using the *template*. The last six characters of the template must be "XXXXXX". `mkdtemp` changes the X's in the *template* to generate a unique pathname. *template* is modified and must not be a constant string. It should be a null terminated character array. The directory is created with the permissions 0700. `mkdtemp` returns a pointer to the modified *template* on success and NULL on failure.

tmpfile

```
#include <stdio.h>

FILE *tmpfile (void);
```

The `tmpfile` function creates a temporary file. It returns a `FILE` pointer or `NULL` in case of error. The file is automatically opened for writing and is deleted when it is closed, or, when the calling process terminates. An example program using `tmpfile` is given below.

Timer

```
#include <signal.h>            /* Definition of SIGEV_* constants */
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *restrict sevp,
                 timer_t *restrict timerid);

Link with -lrt.
```

```
#include <time.h>

int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict new_value,
                  struct itimerspec *restrict old_value);
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);

Link with -lrt.
```

WatchDog Timer

Debugfs,Procfs,Sysfs

There are three commonly used pseudo file systems in the kernel: procfs, debugfs, and sysfs.

1.procfs — The proc filesystem is a pseudo-filesystem which provides an interface to kernel data structures.

2.sysfs — The filesystem for exporting kernel objects.

3.debugfs — Debugfs exists as a simple way for kernel developers to make information available to user space.

Debugfs,Procfs,Sysfs

They are used for data exchange between the Linux kernel and user space, but the applicable scenarios are different:

- The earliest history of procfs was originally used to interact with the kernel to obtain various information such as processors, memory, device drivers, and processes.

- Sysfs is closely tied to the kobject framework, and kobject exists for the device driver model, so sysfs is for device drivers.

- Debugfs is born from the name of the name, so it is more flexible.

Debugfs,Procfs,Sysfs

Their mounting methods are similar, let's do an experiment:

```
$ sudo mkdir /tmp/{proc,sys,debug}
$ sudo mount -t proc nondev /tmp/proc/
$ sudo mount -t sysfs nondev /tmp/sys/
$ sudo mount -t debugfs nondev /tmp/debug/
```

Debugfs,Procfs,Sysfs

The following is a brief introduction to the usage of these three file systems. Before you introduce, please write down their official documentation:

procfs — Documentation/filesystems/proc.txt

sysfs — Documentation/filesystems/sysfs.txt

debugfs — Documentation/filesystems/debugfs.txt

IOCTL

IOCTL is referred to as Input and Output Control, which is used to talking to device drivers. This system call, available in most driver categories. The major use of this is in case of handling some specific operations of a device for which the kernel does not have a system call by default.

Some real-time applications of ioctl are Ejecting the media from a “cd” drive, change the Baud Rate of Serial port, Adjust the Volume, Reading or Writing device registers, etc. We already have the write and read function in our device driver. But it is not enough for all cases.

Users and Groups

Linux understands Users and Groups

A user can belong to several groups

A file can belong to only one user and one group at a time

A particular user, the superuser *“root”* has extra privileges (uid = “0” in /etc/passwd)

Only root can change the ownership of a file

Users and Groups cont.

User information in `/etc/passwd`

Password info is in `/etc/shadow`

Group information is in `/etc/group`

`/etc/passwd` and `/etc/group` divide data fields using ":"

/etc/passwd:

```
joeuser:x:1000:1000:Joe User,,,:/home/joeuser:/bin/bash
```

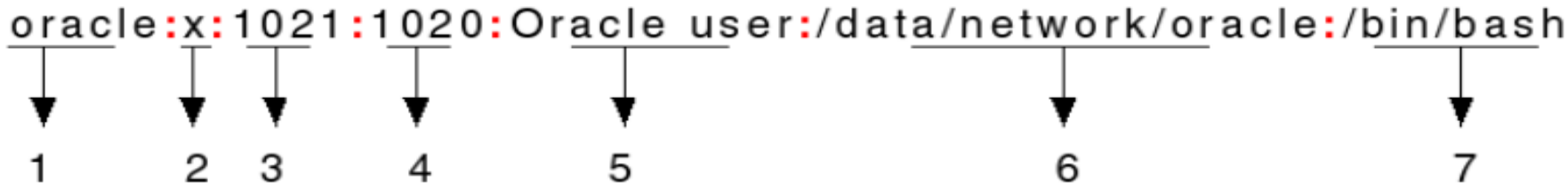
/etc/group:

```
joeuser:x:1000:
```

Users and Groups cont.

Understanding /etc/passwd file fields

The /etc/passwd contains one entry per line for each user (user account) of the system. All fields are separated by a colon (`:`) symbol. Total of seven fields as follows. Generally, /etc/passwd file entry looks as follows:



(Fig.01: /etc/passwd file format – click to enlarge)

/etc/passwd Format

From the above image:

1. **Username**: It is used when user logs in. It should be between 1 and 32 characters in length.

Users and Groups cont.

2. **Password**: An x character indicates that encrypted password is stored in /etc/shadow file. Please note that you need to use the passwd command to computes the hash of a password typed at the CLI or to store/update the hash of the password in /etc/shadow file.
3. **User ID (UID)**: Each user must be assigned a user ID (UID). UID 0 (zero) is reserved for root and UIDs 1-99 are reserved for other predefined accounts. Further UID 100-999 are reserved by system for administrative and system accounts/groups.
4. **Group ID (GID)**: The primary group ID (stored in /etc/group file)
5. **User ID Info (GECOS)**: The comment field. It allow you to add extra information about the users such as user's full name, phone number etc. This field use by finger command.
6. **Home directory**: The absolute path to the directory the user will be in when they log in. If this directory does not exists then users directory becomes /
7. **Command/shell**: The absolute path of a command or shell (/bin/bash). Typically, this is a shell. Please note that it does not have to be a shell. For example, sysadmin can use the nologin shell, which acts as a replacement shell for the user accounts. If shell set to `/sbin/nologin` and the user tries to log in to the Linux system directly, the /sbin/nologin shell closes the connection.

Users and Groups cont.

Understanding the /etc/group File

It stores group information or defines the user groups i.e. it defines the groups to which users belong. There is one entry per line, and each line has the following format (all fields are separated by a colon (:)

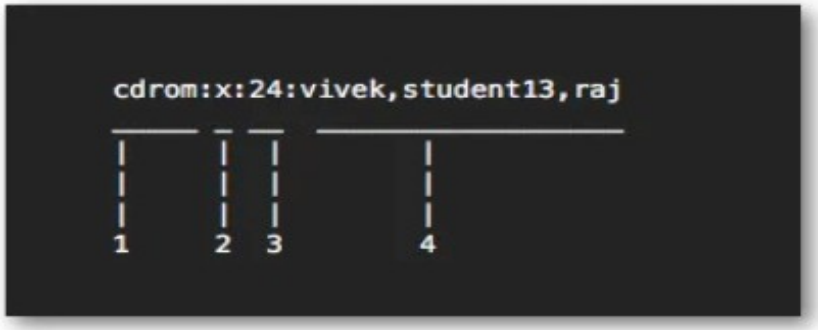


Fig.01: Sample entry in /etc/group file

Where,

1. **group_name**: It is the name of group. If you run ls -l command, you will see this name printed in the group field.

Users and Groups cont.

2. **Password**: Generally password is not used, hence it is empty/blank. It can store encrypted password. This is useful to implement privileged groups.
3. **Group ID (GID)**: Each user must be assigned a group ID. You can see this number in your /etc/passwd file.
4. **Group List**: It is a list of user names of users who are members of the group. The user names, must be separated by commas.

A program runs...

A program may be run by a user, when the system starts or by another process.

Before the program can execute the kernel inspects several things:

- Is the file containing the program accessible to the user or group of the process that wants to run it?
- Does the file containing the program permit execution by that user or group (or anybody)?
- In most cases, while executing, a program inherits the privileges of the user/process who started it.

A program in detail

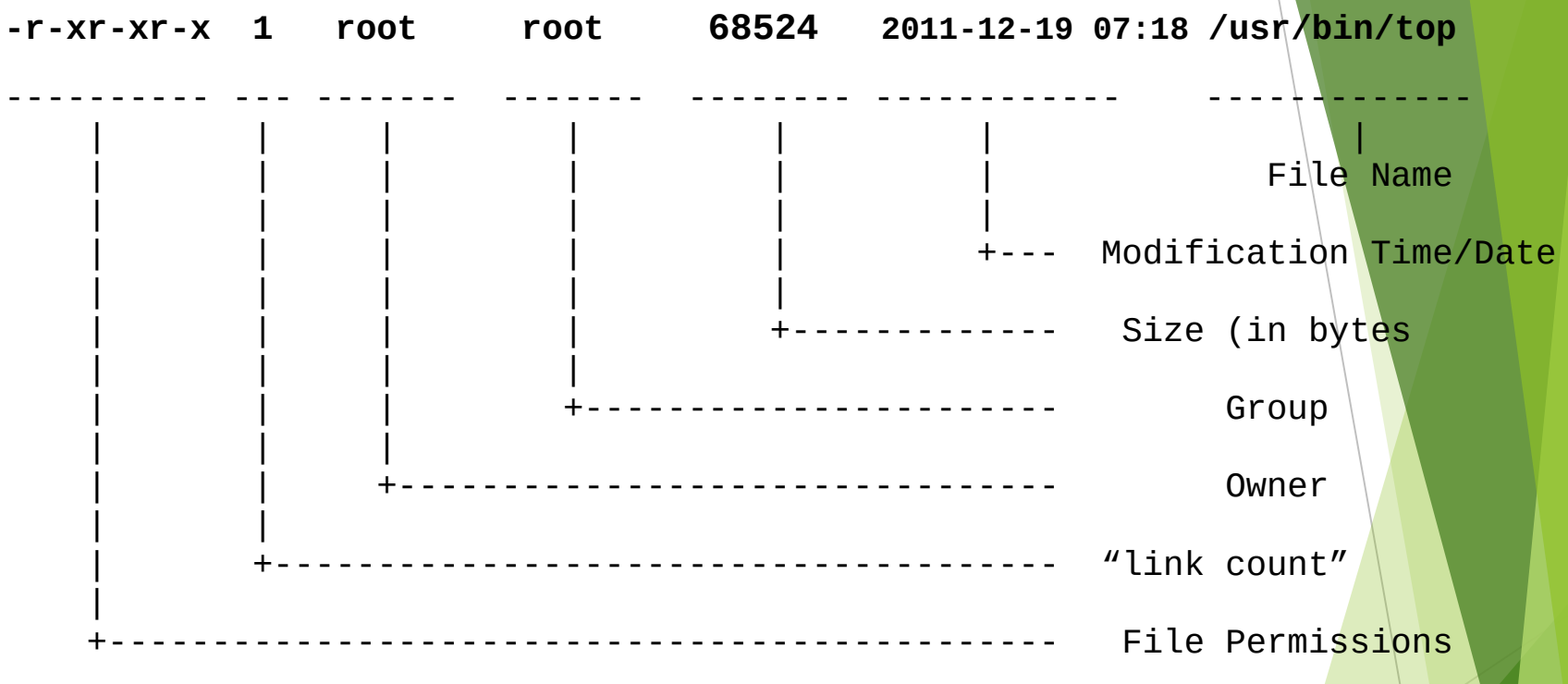
When we type:

```
ls -l /usr/bin/top
```

We'll see:

```
-rwxr-xr-x 1 root root 68524 2011-12-19 07:18 /usr/bin/top
```

What does all this mean?



Group

The name of the group that has permissions in addition to the file's owner.

Owner

The name of the user who owns the file.

File Permissions

The first character is the type of file. A "-" indicates a regular (ordinary) file. A "d” indicate a directory. Second set of 3 characters represent the read, write, and execution rights of the file's owner. Next 3 represent the rights of the file's group, and the final 3 represent the rights granted to everybody else.

(Example modified from **http://www.linuxcommand.org/lts0030.php**)

Access rights

Files are owned by a *user* and a *group* (ownership)

Files have permissions for the user, the group, and *other*

"*other*" permission is often referred to as "world"

The permissions are *Read, Write* and *Execute* (R, W, X)

The user who owns a file is always allowed to change its permissions

Some special cases

When looking at the output from "`ls -l`" in the first column you might see:

```
d = directory
- = regular file
l = symbolic link
s = Unix domain socket
p = named pipe
c = character device file
b = block device file
```

Some special cases cont

In the Owner, Group and other columns you might see:

`s = setuid`	[when in Owner column]
`s = setgid`	[when in Group column]
`t = sticky bit`	[when at end]

File permissions

There are two ways to set permissions when using the chmod command:

Symbolic mode:

testfile has permissions of `-r--r--r--`

```
                                U   G   O*
$ chmod g+x testfile      ==>  -r--r-xr--
$ chmod u+wx testfile     ==>  -rwxr-xr--
$ chmod ug-x testfile     ==>  -rw--r--r--
U=user, G=group, O=other (world)
```

File permissions cont.

Absolute mode:

We use octal (base eight) values represented like this:

Letter	Permission	Value
R	read	4
W	write	2
X	execute	1
-	none	0

For each column, User, Group or Other you can set values from 0 to 7. Here is what each means:

0= **---** 1= **--x** 2= **-w-** 3= **-wx**

4= **r--** 5= **r-x** 6= **rw-** 7= **rwx**

File permissions cont.

Numeric mode cont:

Example index.html file with typical permission values:

```
$ chmod 755 index.html
$ ls -l index.html
-rwxr-xr-x  1 root  wheel  0 May 24 06:20 index.html

$ chmod 644 index.html
$ ls -l index.html
-rw-r--r--  1 root  wheel  0 May 24 06:20 index.html
```

Inherited permissions

Two critical points:

1. The permissions of a directory affect whether someone can see its contents or add or remove files in it.
2. The permissions on a file determine what a user can do to the data in the file.

Example:

If you don't have write permission for a directory, then you can't delete a file in the directory. If you have write access to the file you can update the data in the file.

Working with users and Groups

How Linux User Accounts Work

Username

Password

- By default, all user home directories are created and maintained in the /home

  directory.

- However, the root user's home directory is /root

To view information about the user_name account on my Linux system, you would enter finger user_name

The following information about the user account:

- Login This is the username that is used to authenticate to the system.
- Name This is the user's full name.
- Directory This is the user's home directory.
- Shell This is the default shell that will be provided to the user.
- Last Login This displays the last time the user logged in and where from.

In addition to having a home directory and default shell assigned, each user account is also assigned a unique user ID (UID) number when they are created. **No** two user accounts on the system will have the same UID. To view the UID for a given user account, you can use the **id username** command from the shell prompt. For example, to view information about our vmk user account, we can enter

id vmk at the shell prompt

On a SUSE Linux system, the first regular user account created on the system is always assigned a UID of **1000**. The next user account will be assigned a UID of 1001...

Other distributions may use a different numbering scheme for the UID, however. For example, UIDs on a Fedora system start at **500** instead of 1000.

The root user account is always assigned a UID of 0 on most Linux distributions.

It’s this UID that the operating system actually uses to control access to files and directories in the file system.

Creating and Managing User Accounts from the Command Line

Using useradd

Using passwd

Using usermod

Using userdel

- Using useradd

Syntax: **useradd** options username

ex (options default): **useradd** ncth

ncth account is created using the default parameters contained in the following configuration files: /etc/default/useradd

/etc/login.defs This file contains values that can be used for the GID and UID parameters when creating an account with useradd. It also contains

defaults for creating passwords in /etc/shadow.

You can also view these default values by entering useradd –D at the shell prompt.

Options

–c Includes the user's full name.

–e Specifies the date when the user account will be disabled. Format the date as yyyy-mm-dd.

–f Specifies the number of days after password expiration before the account is disabled. Use a value of –1 to disable this functionality, e.g., useradd –f –1 jmcarthur.

–g Specifies the user's default group.

–G Specifies additional groups that the user is to be made a member of.

–M Specifies that the user account be created without a home directory.

–m Specifies the user’s home directory.

–n Used only on Red Hat or Fedora systems. By default, these systems create a new group with the same name as the user every time an account is created. Using this option will turn off this functionality.

–p Specifies the user’s password.

–r Specifies that the user being created is a system user.

–s Specifies the default shell for the user.

–u Manually specifies a UID for the user.

EX: **useradd** –c “Tommy” ncth1

useradd –c “Truong Khac Tung” –m –p “tung123” –s “/bin/bash” tktung

- Using passwd

The passwd utility is used to change an existing user's password

You can find out this information using the –S option with passwd. For example, we could enter passwd **–S** vmk at the shell promp

Syntax: passwd username

options

–l Locks the user’s account. This option invalidates the user’s password.

–u Unlocks a user’s account.

–d Removes a user’s password.

–n Sets the minimum number of days required before a password can be changed.

–x Sets the maximum number of days before a password must be changed.

–w Sets the number of days prior to password expiration when the user will be warned of the pending expiration.

–i Sets the number of days to wait after a password has expired to disable the account.

- Using usermod

From time to time, you will need to modify an existing user account. The syntax for usermod is very similar to that used by useradd.

Syntax:

usermod *options username*

options

–c Edits the user's full name.

–e Sets the date when the user account will be disabled. Format the date as yyyy-mm-dd.

–f Sets the number of days after password expiration before the account is disabled. Use a value of –1 to disable this functionality.

–g Sets the user's default group.

–G Specifies additional groups that the user is to be made a member of.

options

–l Changes the username.

–L Locks the user’s account. This option invalidates the user’s password.

–m Sets the user’s home directory.

–p Sets the user’s password.

–s Specifies the default shell for the user.

–u Sets the UID for the user.

–U Unlocks a user’s account that has been locked.

useradd –c “your_full_name” –m –p “your_password” –s “/bin/bash” your_username.

tail /etc/passwd

Create a user account using your system’s default settings by entering useradd abc

Passwd abc -> enter password

- Using userdel

Syntax:

userdel ***username***

ex: ***userdel*** ***ncth***

- Using userdel

It’s important to note that, by default, userdel will not remove the user’s home directory from the file system. If you do want to remove the home directory when you delete the user, you need to use the **–r** option in the command line. For example, entering **userdel –r ncth** will remove the account and delete her home directory.

Managing groups from the command line

Using groupadd

Using groupmod

Using groupdel

- Using groupadd

Syntax:

groupadd *options* *groupname*

Options:

–g Specifies a GID for the new group.

–p Specifies a password for the group.

–r Specifies that the group being created is a system group.

- Using groupmod

To modify a group, including adding users to the group membership, you use the groupmod utility.

Syntax:

groupmod *options* *group*

Options:

-g Changes the group's GID number.

–p Changes the group's password.

-A Adds a user account to the group.

–R Removes a user account from the group.

If we wanted to add ncth to the group, we would enter

groupmod –A “ncth” *student*

at the shell prompt.

- Using groupdel

Syntax:

groupdel *group_name*

ex: ***groupdel*** **student**

- **How ownership works**

  Anytime a user creates a new file or directory, his or her user account is assigned as that file or directory's "owner." For example, suppose the vmk user logs in to her Linux system and creates a file named linux_introduction.odt using OpenOffice.org in home directory. Because she created this file, ksanders is automatically assigned ownership of linux_introduction.odt. By right-clicking on this file in the system's graphical user interface and selecting Properties | Permissions, you can view who owns the file.

- **How ownership works**

You can also view file ownership from the command line using the **ls –l** command

Managing ownership from the command line

You can specify a different user and/or group as the owner of a given file or directory. To change the user who owns a file, you must be logged in as root. To change the group that owns a file, you must be logged in as root or as the user who currently owns the file.

- Using **chown**
- Using **chgrp**

- Using chown

The chown utility can be used to change the **user** or **group** that owns a file or **directory**.

Syntax **chown** user.group file or directory.

Ex: If I wanted to change the file's owner to the **ncth1** user, I would enter

chown ncth1 /tmp/myfile.txt

chown

If I wanted to change this to the users group, of which **users** is a member, I would enter

chown .users /tmp/myfile.txt

Notice that I used a period **(.)** before the group name to tell chown that the entity specified is a group, not a user account.

Ex: **chown** student.users /tmp/myfile.txt

Note: You can use the –R option with chown to change ownership on many files at once recursively.

- Using chgrp

In addition to chown, you can also use **chgrp** to change the group that owns a file or directory.

Syntax:

chgrp *group* *file* (or *directory*)

For example:

chgrp student /tmp/newfile.txt.

- How permissions work

TABLE 7-1 Linux Permissions

Permission	Symbol	Effect on Files	Effect on Directories
Read	r	Allows a user to open and view a file. Does not allow a file to be modified or saved.	Allows a user to list the contents of a directory.
Write	w	Allows a user to open, modify, and save a file.	Allows a user to add or delete files from the directory.
Execute	x	Allows a user to run an executable file.	Allows a user to enter a directory.

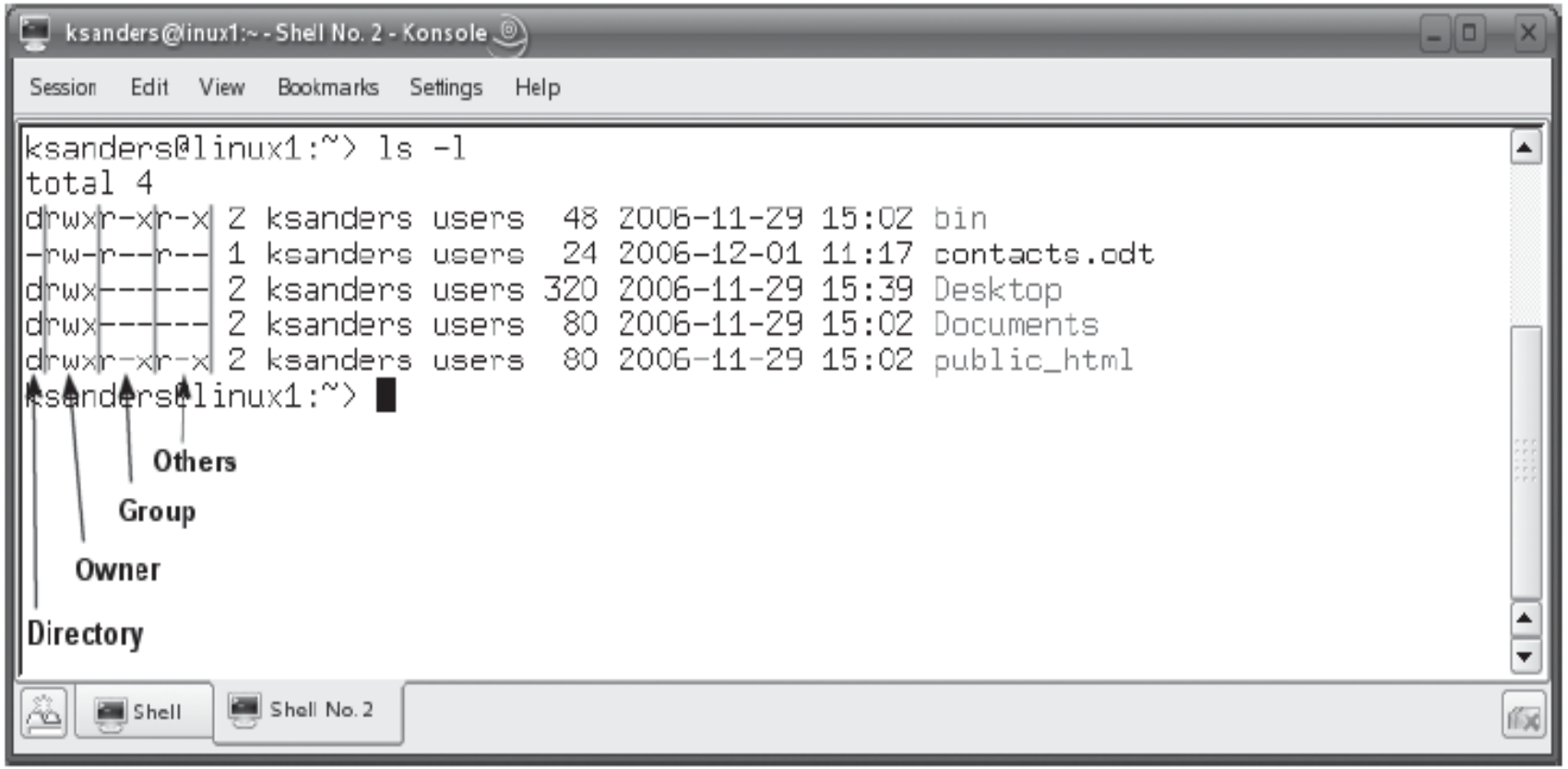
Each file or directory in your Linux file system stores the specific permissions assigned to it. These permissions together constitute the mode of the file. These permissions are assigned to each of three different entities for each file and directory in the file system:

- **Owner** This is the user account that has been assigned to be the file or directory's owner. Permissions assigned to the owner apply only to that user account.

- **Group** This is the group that has been assigned ownership of the file or directory. Permissions assigned to the group apply to all user accounts that are members of that group.
- **Others** This entity refers to all other users who have successful authenticated to the system. Permissions assigned to this entity apply to these user accounts.

ls -l

FIGURE 7-28 Viewing the file mode with ls –l



- Managing Permissions from the Command Line with chmod

chmod *entity=permissions* *filename*

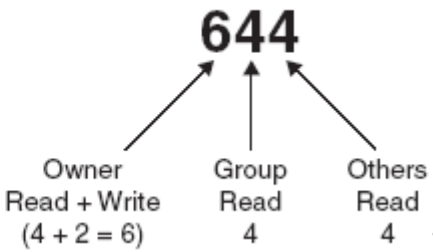
TABLE 7-2

Numeric Values Assigned to Permissions

Permission	Value
Read	4
Write	2
Execute	1

FIGURE 7-29

Representing permissions numerically



Owner, **g for Group**, and o for Others in the entity portion of the command. You substitute **r, w**, and/or **x** for the permissions portion of the command. For example, suppose I wanted to change the mode of contacts.odt to –rw–rw–r– –

chmod u=rw,g=rw,o=r contacts.odt

You can also use chmod to toggle a particular permission on or off using the + or – signs. For example, suppose I want to turn off the write permission I just gave to Group for the contacts.odt file. I could enter **chmod** g–w **contacts.odt**

at the shell prompt.

You can modify all three entities at once with only three characters. To do this, enter

chmod numeric_permission filename

ex: chmod 660 contacts.odt

Working with default permissions

By default, Linux assigns rw–rw–rw– (666) permissions to every file whenever it is created in the file system. It also assigns rwxrwxrwx permissions to every directory created in the file system. It also assigns rwxrwxrwx permissions to every directory created in the file system.

To increase the overall security of the system, Linux uses a variable called umask to automatically remove permissions from the default mode whenever a file or directory is created in the file system. The value of umask is a three-digit number

value of umask is 022. Each digit represents a numeric permission value to be removed. The first digit references—you guessed it—Owner, the second references Group, the last references Other.

If you only need to make a temporary change

to umask, you can enter **umask value** at the shell prompt. For example, if you wanted to remove the execute permission that is automatically assigned to Others whenever a new directory is created, you could enter umask 023

- Working with Special Permissions

SUID: 4

SGID: 2

Sticky Bit: 1

For example, suppose you wanted to apply the SUID and SGID permissions to a file named runme that should be readable and executable by Owner and Group. You would enter chmod 6554 runme at the shell prompt. This specifies that the file have SUID (4) and SGID (2) permissions assigned (for a total of 6 in the first digit). It also specifies that Owner and Group have read (4) and execute permissions (1) assigned (for a total of 5 in the second and third digits). It also specifies that Others be allowed to read (4) the file, but not be able to modify or run it (for a total of 4 in the last digit)

Suid,guid,stickybit

1. The `setuid` bit

This bit is present for files which have executable permissions. The `setuid` bit simply indicates that when running the executable, it will set its permissions to that of the user who created it (owner), instead of setting it to the user who launched it. Similarly, there is a `setgid` bit which does the same for the `gid`.

To locate the `setuid`, look for an 's' instead of an 'x' in the executable bit of the file permissions.

To set the `setuid` bit, use the following command.

```
chmod u+s
```

To remove the `setuid` bit, use the following command.

```
chmod u-s
```

Suid,guid,stickybit

2. The `setgid` bit

The `setgid` affects both files as well as directories. When used on a file, it executes with the privileges of the group of the user who owns it instead of executing with those of the group of the user who executed it.
When the bit is set for a directory, the set of files in that directory will have the same group as the group of the parent directory, and not that of the user who created those files. This is used for file sharing since they can be now modified by all the users who are part of the group of the parent directory.

To locate the `setgid` bit, look for an 's' in the group section of the file permissions, as shown in the example below.

```
-rwxrwsr-x root root 1427 Aug 2 2019 sample_file
```

To set the `setgid` bit, use the following command.

```
chmod g+s
```

To remove the `setgid` bit, use the following command.

```
chmod g-s
```

Suid,guid,stickybit

3. The sticky bit

The sticky bit was initially introduced to 'stick' an executable program's text segment in the swap space even after the program has completed execution, to speed up the subsequent runs of the same program. However, these days the sticky bit means something entirely different.

When a directory has the sticky bit set, its files can be deleted or renamed only by the file owner, directory owner and the root user. The command below shows how the sticky bit can be set.

```
chmod +t
```

User and Groups

- #include <sys/types.h>
- #include <unistd.h>
- int setuid(uid_t *uid*);
- int setgid(gid_t *gid*);
- If the process has superuser privileges
  - setuid() sets the real user ID, effective user ID, and saved set-user-ID to *uid*
- If the process does not have superuser privileges, but *uid* = the real user ID or the save set-user-ID
  - setuid sets the effective user ID to uid
- If neither is true, errno is set to EPERM and an error is returned

User and Groups

- Only a superuser process can change the real user ID
- The effective user ID is set by the exec functions, only if the setuid bit is set for the program file
  - Can call setuid any time to set the effective user ID to the real user ID or the saved set-user-ID
- The saved set-user-ID is copied from the effective user ID by exec

User and Groups

- int setreuid(uid_t *ruid*, uid_t *euid*);
- int setregid(gid_t *rgid*, gid_t *egid*);
- Swap real and effective user/group IDs
- int seteuid(uid_t *uid*);
- int setegid(gid_t *gid*);
- Set effective user/group ID

Socket Basics (2 of 2)

End point determined by two things:

- Host address: IP address is *Network Layer*
- Port number: is *Transport Layer*

Two end-points determine a connection: socket pair

- ex: `206.62.226.35,p21 + 198.69.10.2,p1500`
- ex: `206.62.226.35,p21 + 198.69.10.2,p1499`

Ports

Numbers (typical, since vary by OS):

- `0-1023` “reserved”, must be root
- `1024 - 5000` “ephemeral”
- Above `5000` for general use

  (`50,000` is specified max)

Well-known, reserved services (see `/etc/services` in Unix):

```
ftp             21/tcp
telnet      23/tcp
finger      79/tcp
snmp            161/udp
```

Transport Layer

UDP: User Datagram Protocol

- no acknowledgements
- no retransmissions
- out of order, duplicates possible
- connectionless

TCP: Transmission Control Protocol

- reliable (in order, all arrive, no duplicates)
- flow control
- Connection-based

While TCP ~95% of all flows and packets, much UDP traffic is games!

Addresses and Sockets

Structure to hold address information

Functions pass address from user to OS

- `bind()`
- `connect()`
- `sendto()`

Functions pass address from OS to user

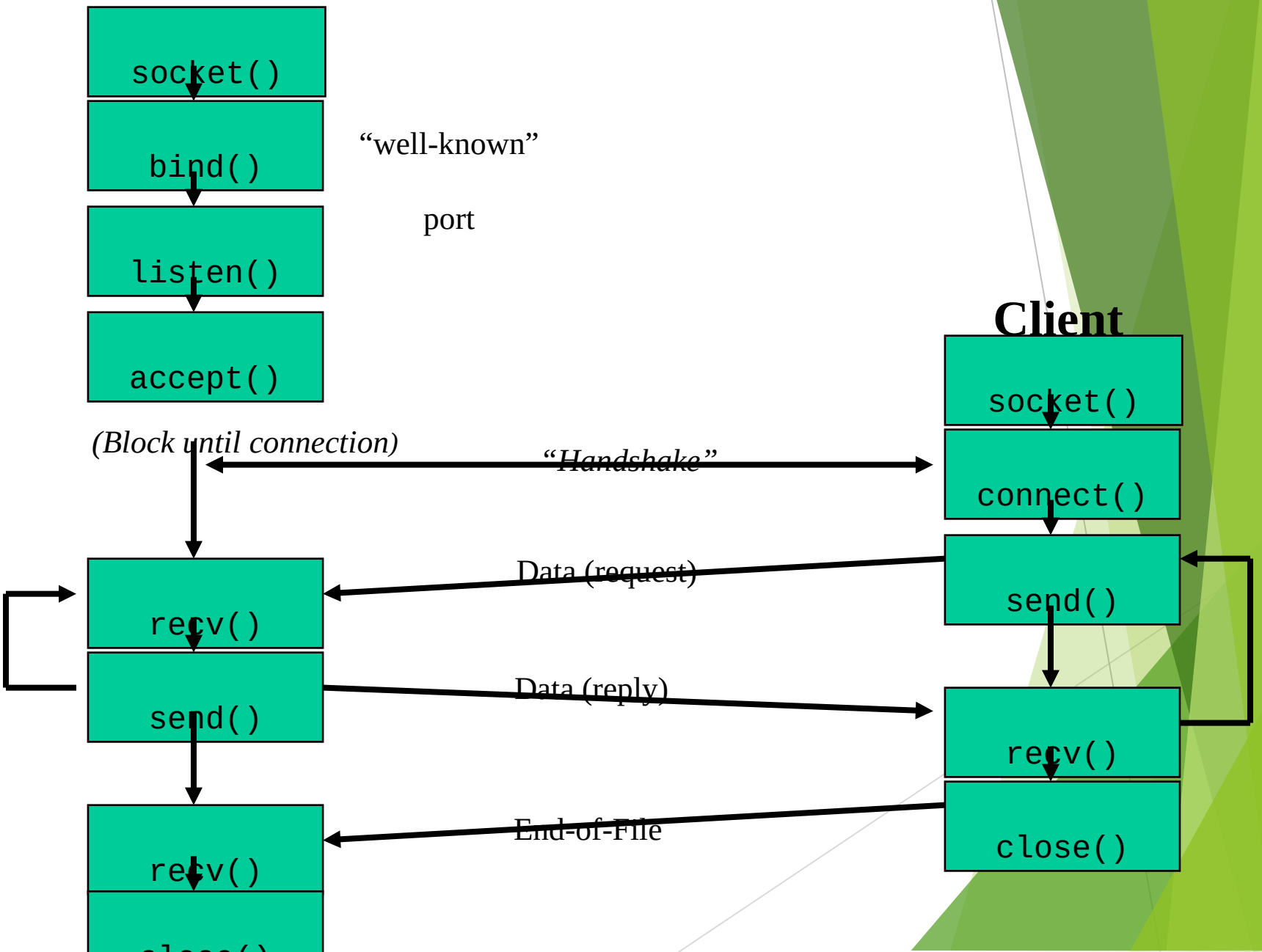
- `accept()`
- `recvfrom()`

Socket Address Structure

```
struct in_addr {
  in_addr_t s_addr;          /* 32-bit IPv4 addresses */
};
struct sockaddr_in {
  unit8_t       sin_len;     /* length of structure */
  sa_family_t   sin_family;  /* AF_INET */
  in_port_t     sin_port;    /* TCP/UDP Port num */
  struct in_addr sin_addr;   /* IPv4 address (above) */
  char sin_zero[8];          /* unused */
}
```

Are also “generic” and “IPv6” socket structures

TCP Client-Server



socket()

`int socket(int family, int type, int protocol);`

Create a socket, giving access to transport layer service.

family is one of

- AF_INET (IPv4), AF_INET6 (IPv6), AF_LOCAL (local Unix),
- AF_ROUTE (access to routing tables), AF_KEY (new, for encryption)

type is one of

- SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
- SOCK_RAW (for special IP packets, PING, etc. Must be root)
  - setuid bit `(-rws--x--x root 1997 /sbin/ping*)`

protocol is 0 (used for some raw socket options)

upon success returns socket descriptor

- Integer, like file descriptor
- Return -1 if failure

bind()

```
int bind(int sockfd, const struct sockaddr *myaddr,
    socklen_t addrlen);
```

Assign a local protocol address (“name”) to a socket.

sockfd is socket descriptor from `socket()`

myaddr is a pointer to address struct with:

- *port number* and *IP address*
- if port is 0, then host will pick ephemeral port
  - not usually for server (exception RPC port-map)
- IP address != INADDR_ANY (unless multiple nics)

addrlen is length of structure

returns 0 if ok, -1 on error

- EADDRINUSE (“Address already in use”)

listen()

```
int listen(int sockfd, int backlog);
```

Change socket state for TCP

sockfd is socket descriptor from `socket()`

backlog is maximum number of *incomplete* connections

- historically 5
- rarely above 15 on a even moderate Web server!

Sockets default to active (for a client)

- change to passive so OS will accept connection

accept()

```
int accept(int sockfd, struct sockaddr
cliaddr, socklen_t *addrlen);
```

Return next completed connection

sockfd is socket descriptor from `socket()`

cliaddr and *addrlen* return protocol address from client

returns brand new descriptor, created by OS

note, if create new process or thread, can create concurrent server

close()

```
int close(int sockfd);
```

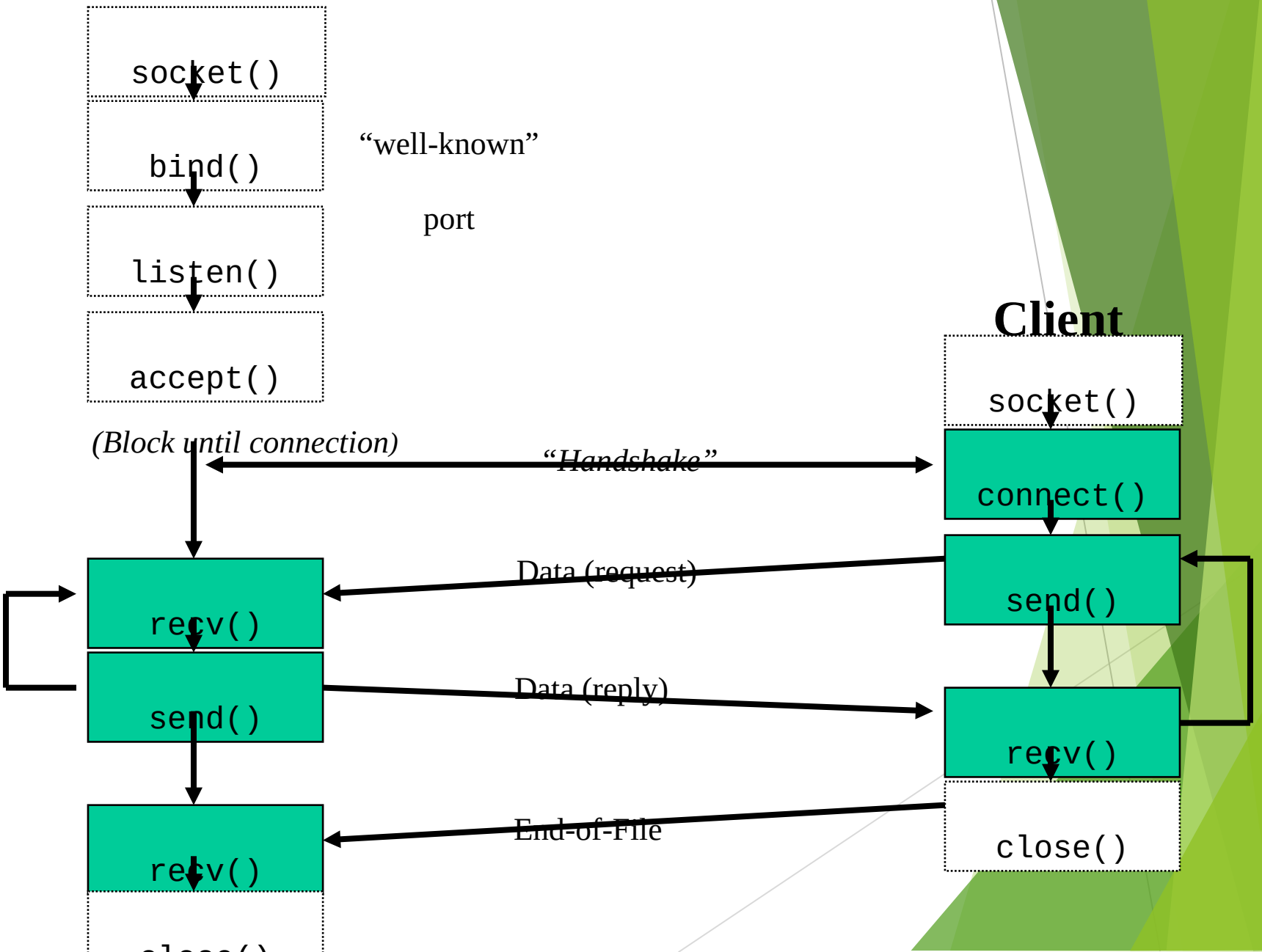
Close socket for use.

sockfd is socket descriptor from `socket()`

closes socket for reading/writing

- returns (doesn't block)
- attempts to send any unsent data
- socket option SO_LINGER
  - block until data sent
  - or discard any remaining data
- returns -1 if error

TCP Client-Server



connect()

```
int connect(int sockfd, const struct sockaddr
*servaddr, socklen_t addrlen);
```

sockfd is socket descriptor from `socket()`

Connect to server.

servaddr is a pointer to a structure with:

- *port number* and *IP address*
- must be specified (unlike `bind()`)

addrlen is length of structure

client doesn't need `bind()`

- OS will pick ephemeral port

returns socket descriptor if ok, -1 on error

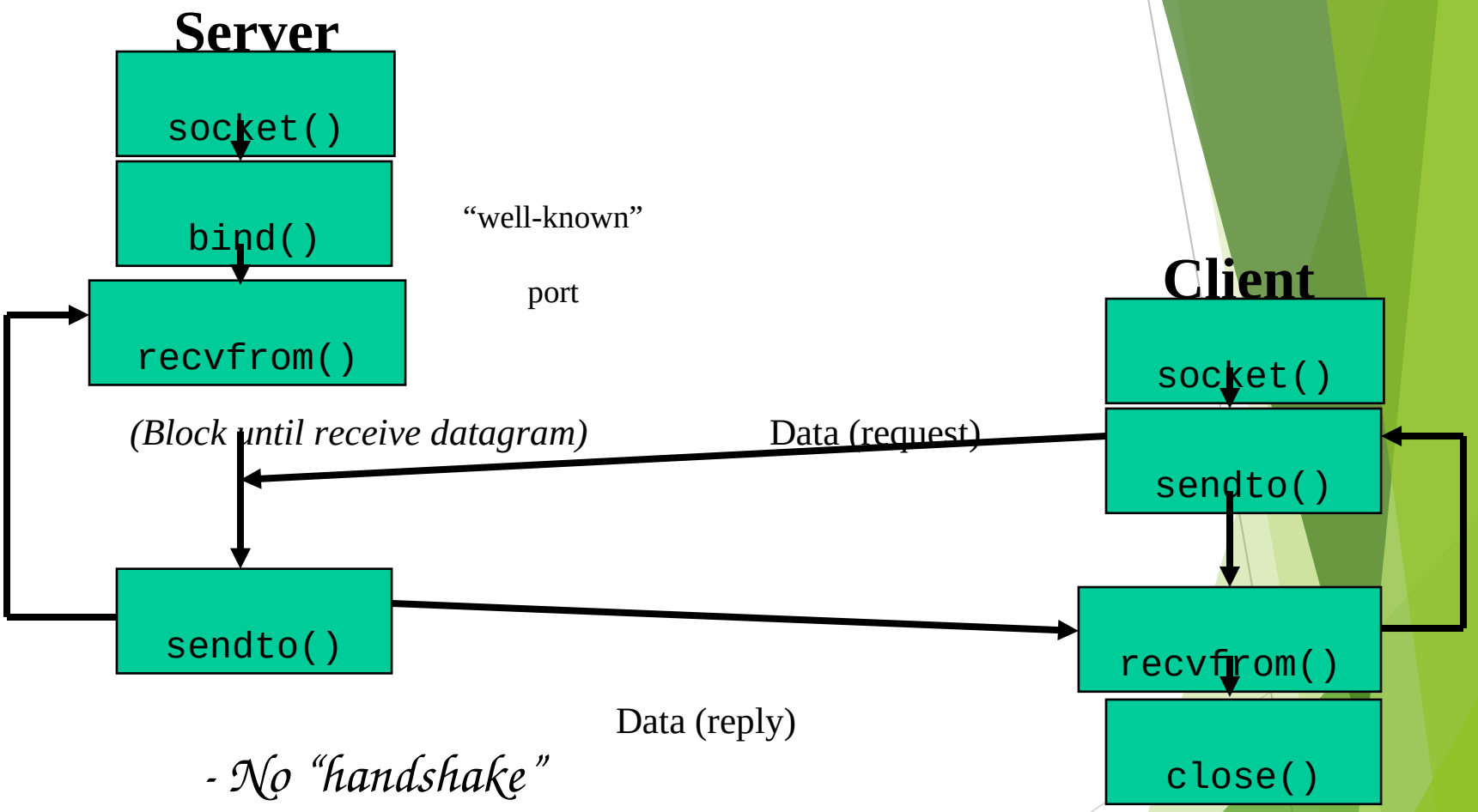
Sending and Receiving

```
int recv(int sockfd, void *buff, size_t
  mbytes, int flags);
int send(int sockfd, void *buff, size_t
  mbytes, int flags);
```

Same as `read()` and `write()` but for *flags*

- MSG_DONTWAIT (this send non-blocking)
- MSG_OOB (out of band data, 1 byte sent ahead)
- MSG_PEEK (look, but don't remove)
- MSG_WAITALL (don't give me less than max)
- MSG_DONTROUTE (bypass routing table)

UDP Client-Server



- *No "handshake"*
- *No simultaneous close*
- *No* `fork()/spawn()` *for concurrent servers!*

Sending and Receiving

```
int recvfrom(int sockfd, void *buff, size_t mbytes,
    int flags, struct sockaddr *from, socklen_t
    *addrlen);

int sendto(int sockfd, void *buff, size_t mbytes, int
    flags, const struct sockaddr *to, socklen_t
    addrlen);
```

Same as `recv()` and `send()` but for *addr*

- `recvfrom` fills in address of where packet came from
- `sendto` requires address of where sending packet to

connect() with UDP

Record address and port of peer

- datagrams to/from others are not allowed
- does not do three way handshake, or connection
- "connect" a misnomer, here. Should be `setpeername()`

Use send() instead of `sendto()`

Use `recv()` instead of `recvfrom()`

Can change connect or unconnect by repeating `connect()` call

(Can do similar with `bind()` on receiver)

Why use connected UDP?

- Send two datagrams unconnected:
  - connect the socket
  - output first dgram
  - unconnect the socket
  - connect the socket
  - ouput second dgram
  - unconnect the socket
- Send two datagrams connected:
  - connect the socket
  - output first dgram
  - ouput second dgram

Socket Options

`setsockopt(), getsockopt()`

SO_LINGER

- upon close, discard data or block until sent

SO_RCVBUF, SO_SNDBUF

- change buffer sizes
- for TCP is “pipeline”, for UDP is “discard”

SO_RCVLOWAT, SO_SNDLOWAT

- how much data before “readable” via `select()`

SO_RCVTIMEO, SO_SNDTIMEO

- timeouts

Socket Options (TCP)

TCP_KEEPALIVE

- idle time before close (2 hours, default)

TCP_MAXRT

- set timeout value

TCP_NODELAY

- disable Nagle Algorithm
- won’t buffer data for larger chunk, but sends immediately

fcntl()

‘File control’ but used for sockets, too

Signal driven sockets

Set socket owner

Get socket owner

Set socket non-blocking

```
flags = fcntl(sockfd, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(sockfd, F_SETFL, flags);
```

Beware not getting flags before setting!

Concurrent Servers

Text segment

```
sock = socket()
/* setup socket */
while (1) {
   newsock = accept(sock)
   fork()
   if child
      read(newsock)
      until exit

}
```

Parent

```
int sock;

int newsock;
```

Child

```
int sock;

int newsock;
```

Close sock in child, newsock in parent

Reference count for socket descriptor