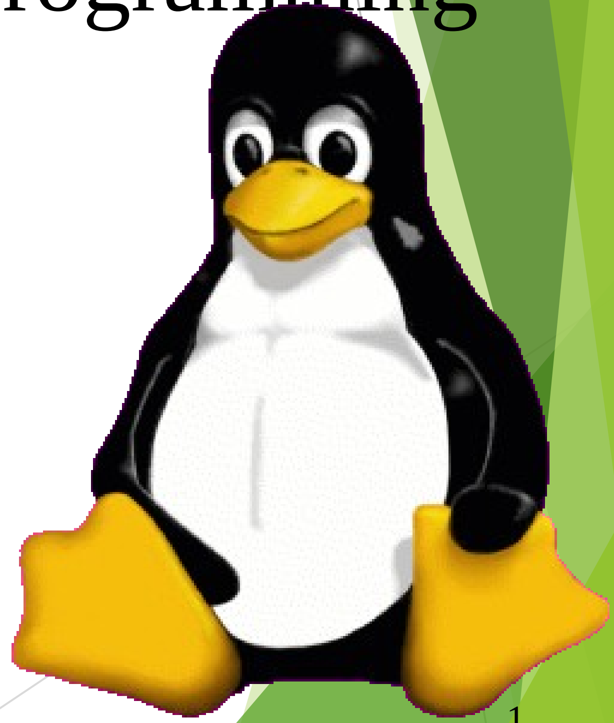
Advanced C and System Programming

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

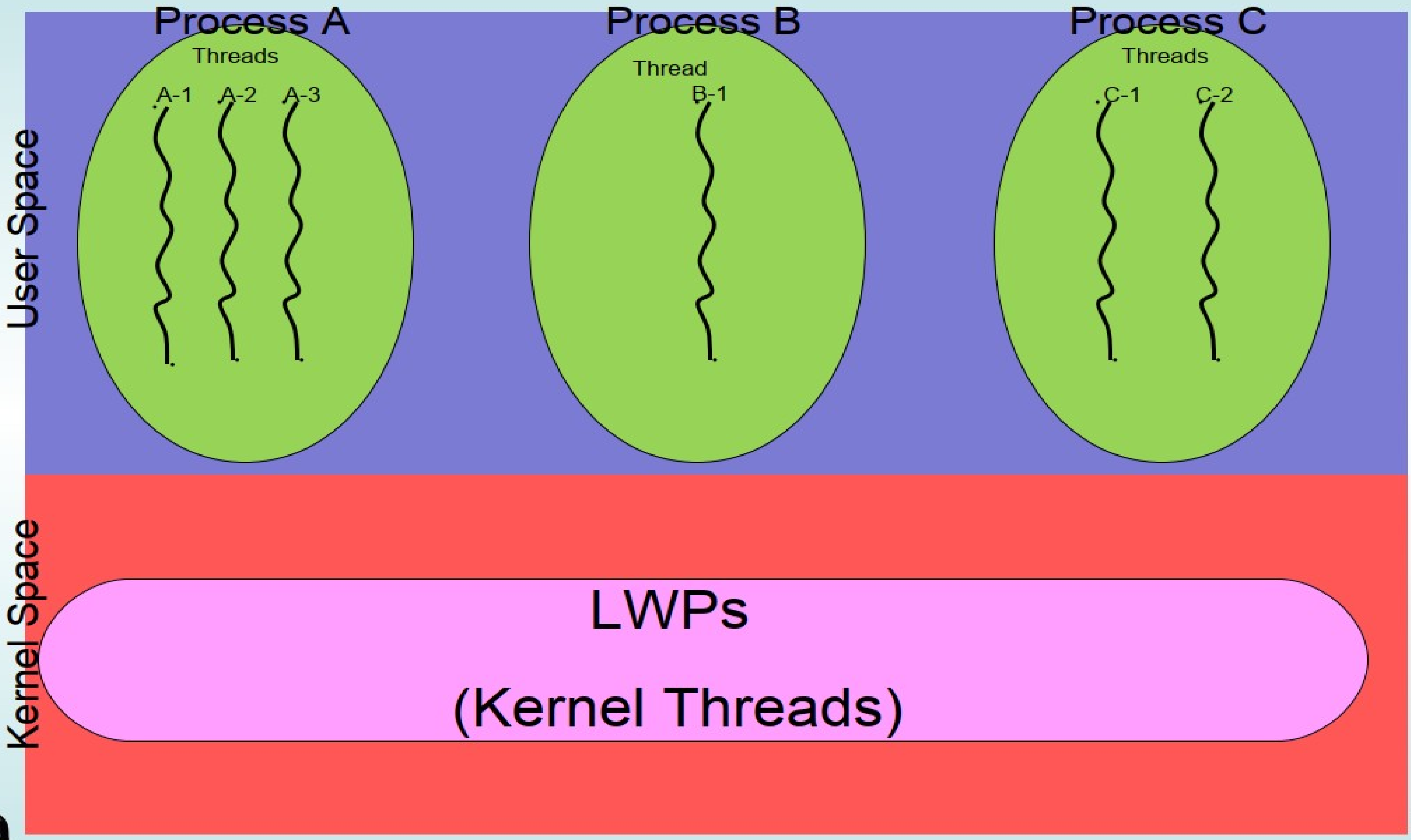
- A Thread is an independent stream of instructions that can be schedule to run as such by the OS.
- Think of a thread as a “procedure” that runs independently from its main program.
- Multi-threaded programs are where several procedures are able to be scheduled to run simultaneously and/or independently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

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. Timings reflect 50,000 processes/thread creations.

Threads vs Forks

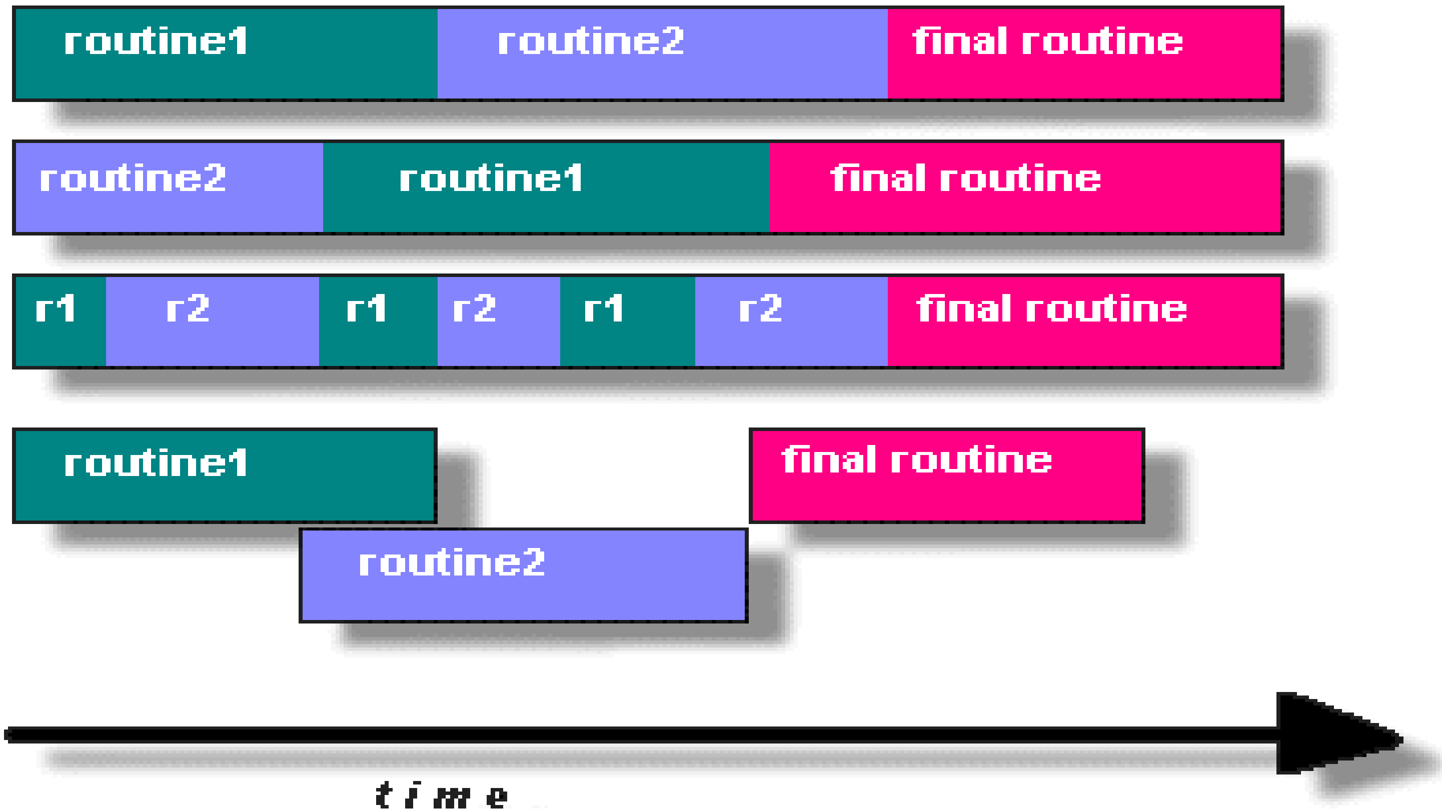
PLATFORM	fork()			pthread_create()		
	REAL	USER	SYSTEM	REAL	USER	SYSTEM
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	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.

Will return zero on success.

Thread ID of type pthread_t

Joinable or what?

```
int pthread_create( thread_id, attribute,
    thread start routine, routine arg )
```

What should this thread do?

Argument to thread 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*.

- *pthread_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!

Hello World! It's me, thread #4!

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)

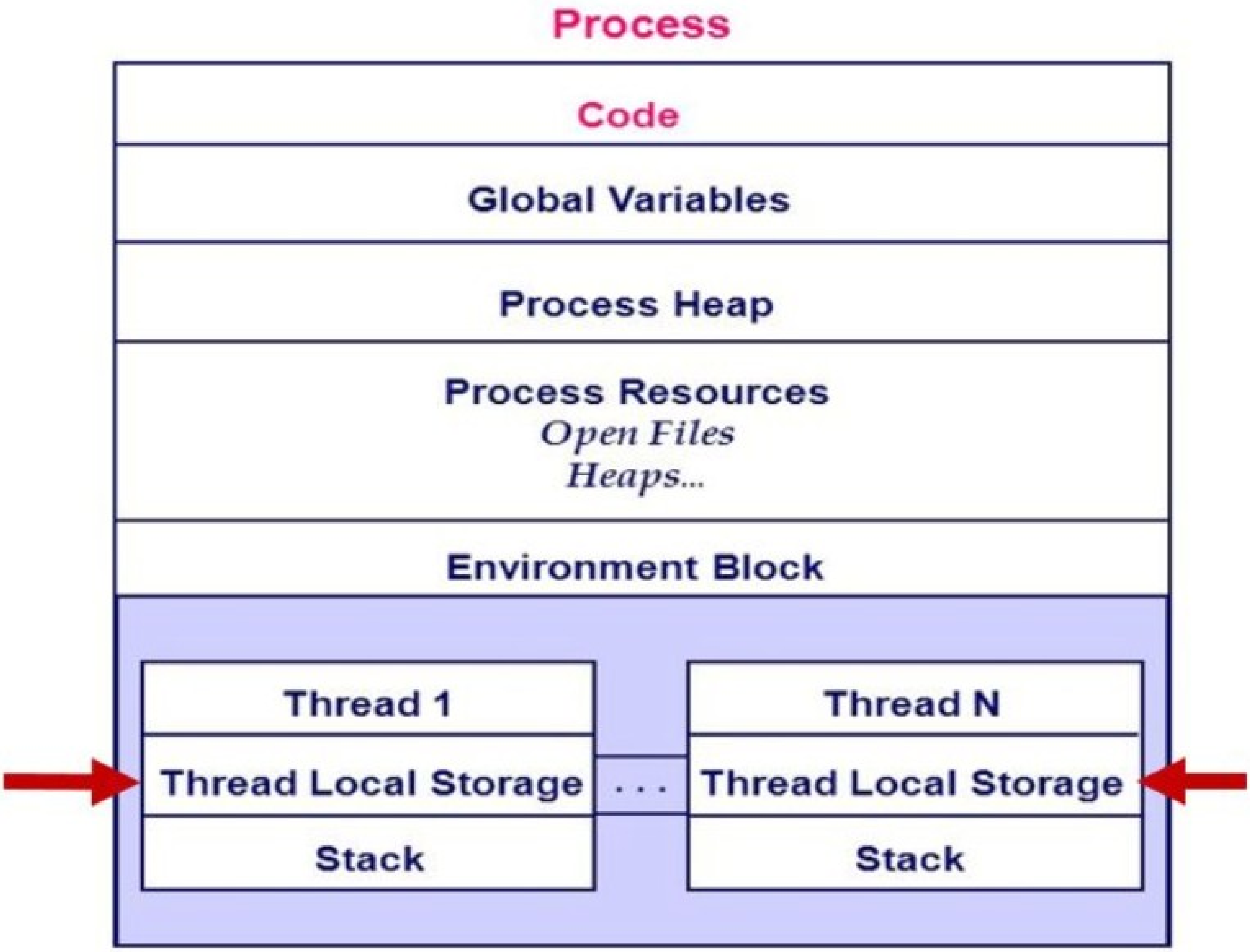
- There are two cancelation types:

- **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);
int sem_wait(sem_t *sem);
```

Unnamed Semaphores

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

Initialize an unnamed semaphore

Returns

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

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

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 unrelated processes

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

```c
#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++);
     }

     pthread_mutex_unlock (&my_lock);
     for (j=0; j<50000000; j++);
   }
}
```

```c
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

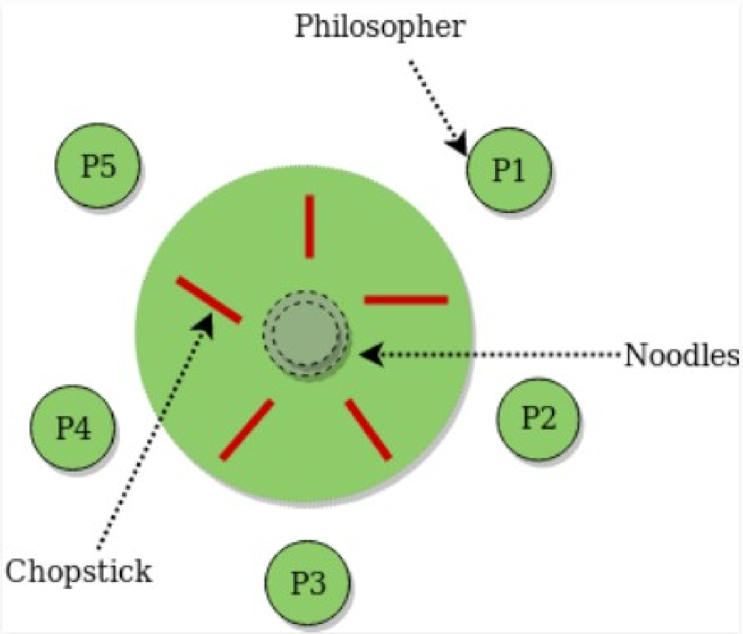
Consumed _item= buffer[j];

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:

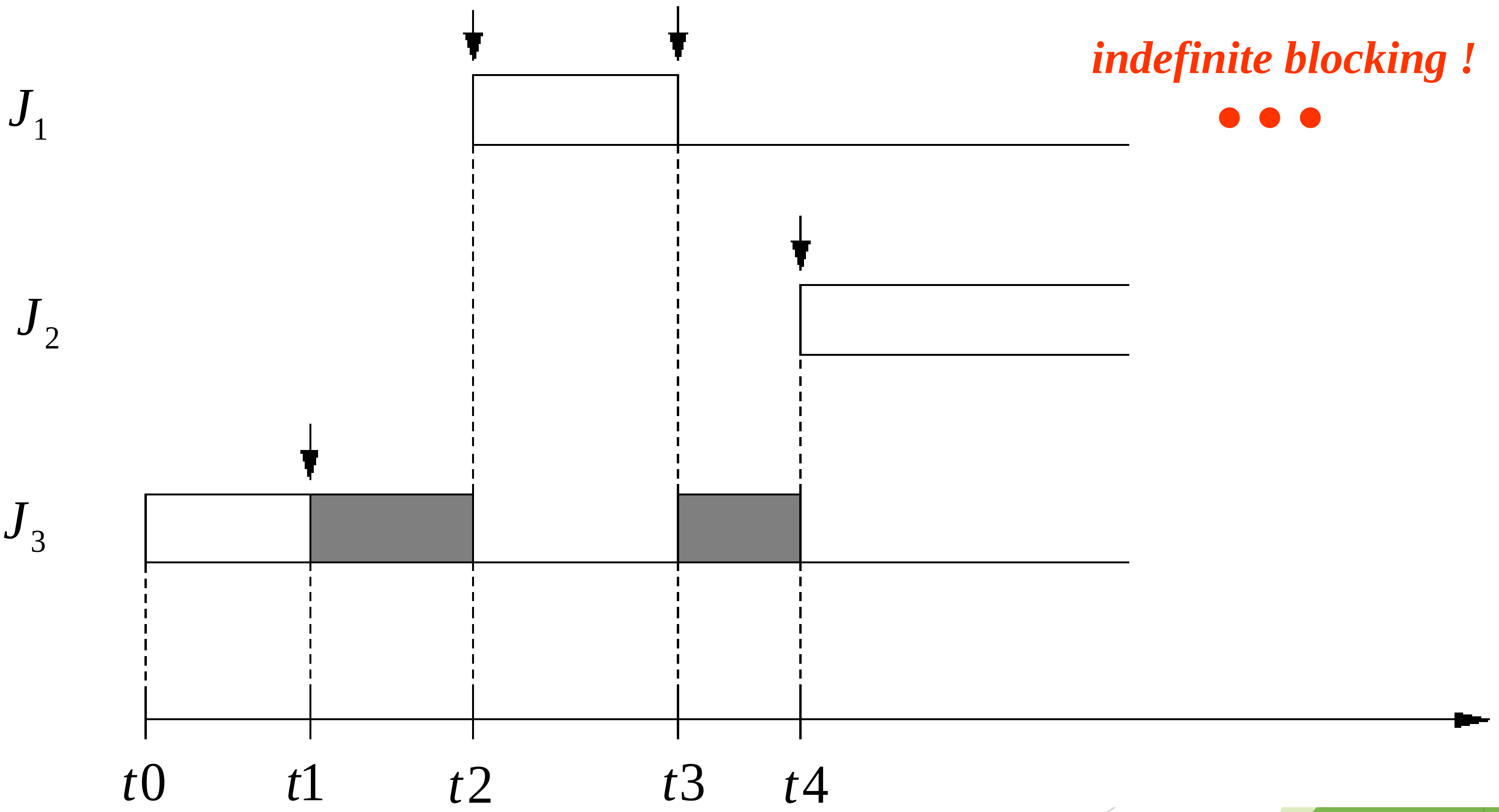
```
process P[i]
 while true do
   {  THINK;
      PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
      EAT;
      PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
   }
```

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



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: