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.

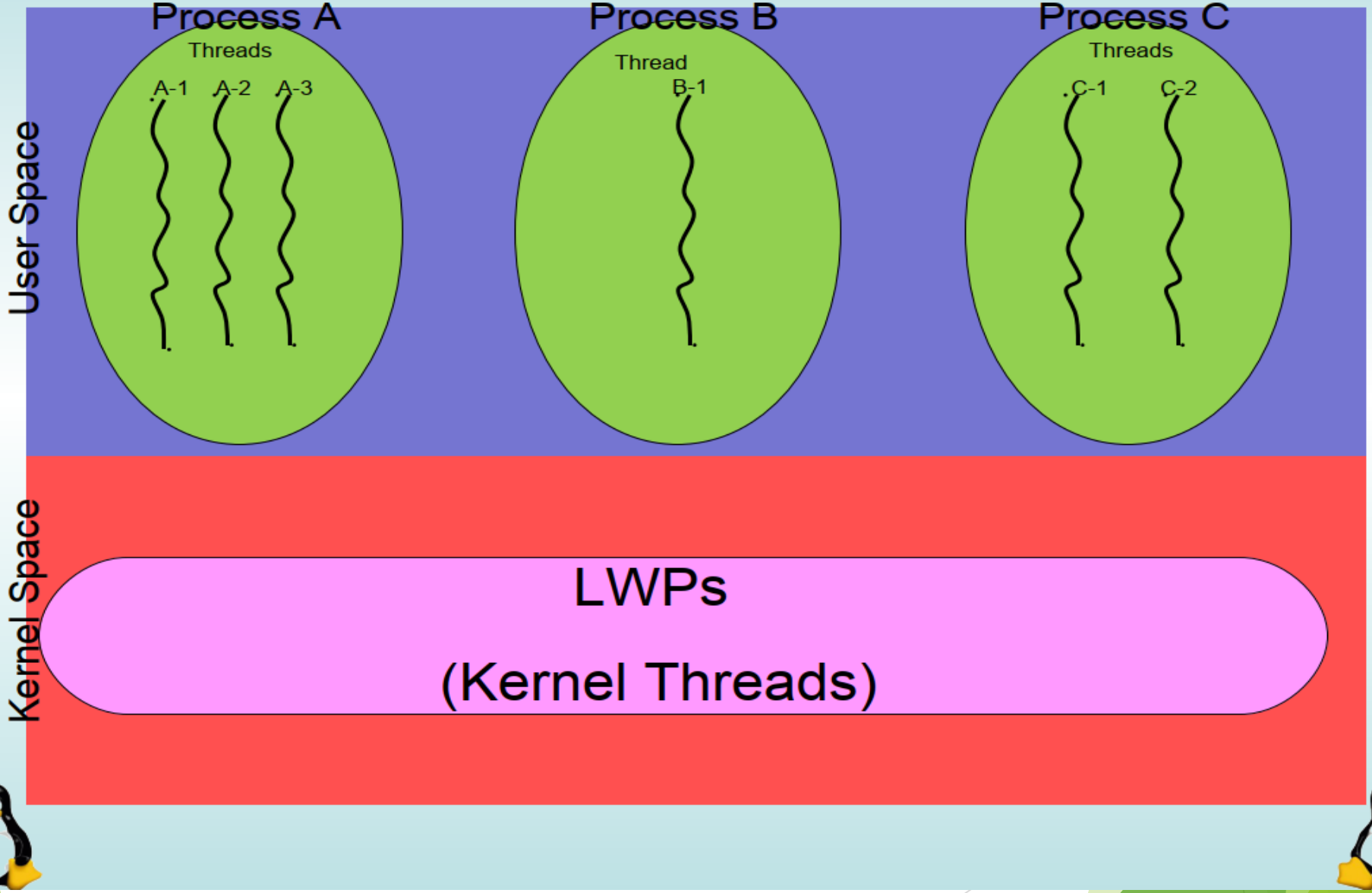
A Thread exists within a process and uses the process resources.

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

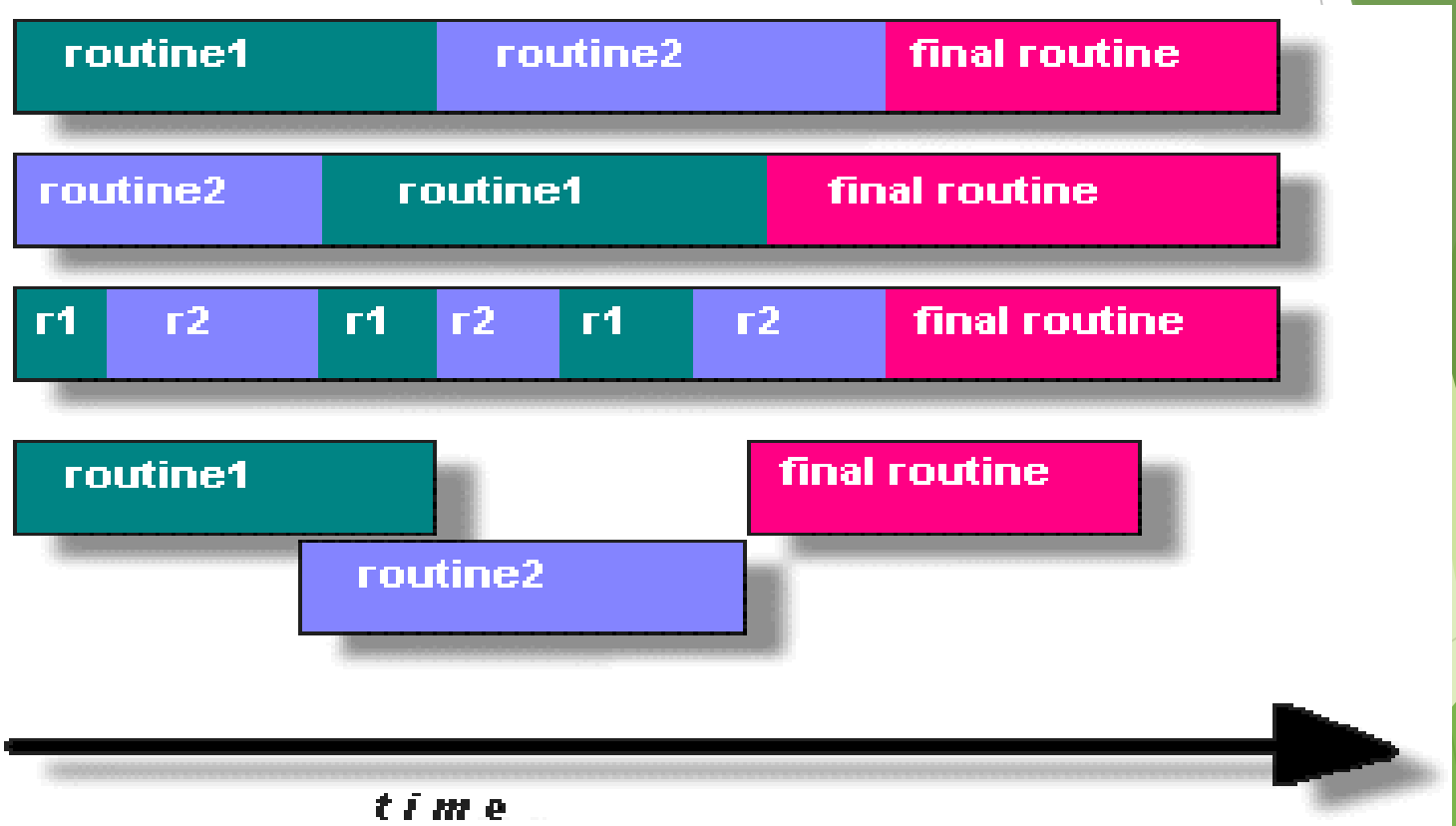
PLATFORM	fork()			pthread_create()		
	REAL	USER	SYSTEM	REAL	USER	SYSTEM
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)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

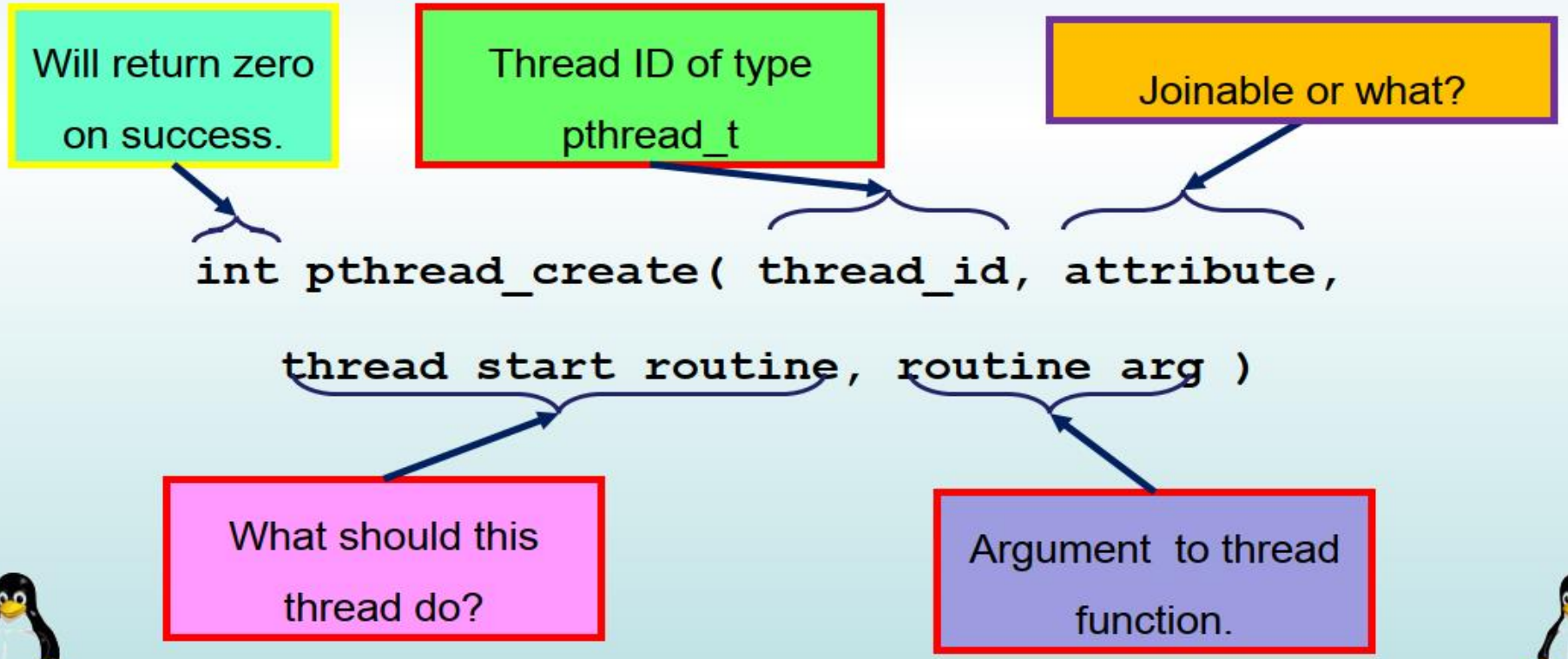
Common models for threaded programs:

- 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.



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…

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).