50004 - Operating Systems - Lecture 3

Oliver Killane

21/10/21

Lecture Recording

Lecture recording is available here

Threads

Threads are execution streams that share the same address space. When multi-threading is used, each process can have more than one thread.

Process	**Thread**
Address Space	Program Counter
Global Variables	Registers
Open files	Stack
Child Processes	
Signals	

Applications may want to run many activities in parallel, with access shared memory. They also need to be capable of synchronisation, through semaphores, locks & monitors or blocking to prevent race conditions on shared data.

Processes have several disadvantages:

- **Communication**
  Difficult to communicate between different address spaces, requiring system calls (which have overhead if a context switch is required).

  If we split our program into several processes, each has a different address space, so all communication (internally in our program) has to use expensive system calls.
- **Blocking**
  Blocking activities may block the whole process, rather than one thread of a process.
- **Context Switches**
  Transfering between processes requires expensive context switches.
- **Create & Kill**
  Creation and desruction costs are higher.

Threads are a more lightweight abstraction for concurrency.

However threads access shared data, which means race conditions & memory corruption can occur.

PThreads

```
1 #include <pthread.h>
2 #include <sys/types.h>
3
4 pthread_t thread;              /* thread handle */
5 pthread_attr_t thread_attrs;    /* thread attributes */
```

Defined by **IEEE** standard 1003.1c and implemented on most **UNIX** systems. A library that internally makes systems calls to enable threading.

```c
#include <pthread.h>
#include <sys/types.h>

pthread_t thread;               /* thread handle */
pthread_attr_t thread_attrs;    /* thread attributes */
```

Creating Threads

```c
int pthread_create (pthread_t *thread , const pthread_attr_t *attr ,void *(*
    ↪ start_routine )(void*), void *arg)
```

Creates a new thread stored in **thread**, returning 0 for a successful creation, or an error code for failure.

Arguments:

- **attr**
  specifies thread attributes, can be **NULL** for default (e.g minimum stack stack size, guard size, detached/joinable etc)
- **start_routine**
  C function the thread will start executing from, returning some pointer, and taking some pointer as arguments.
- **arg**
  Arguments to be passed to start routine, can be **NULL** if no arguments are to be passed.

Terminating Threads

```c
void pthread_exit (void *value_ptr)
```

Terminates the thread and makes **value_ptr** available to any successful join with the terminating thread.

Is called implicitly when the thread's start routine returns.

- Not called for the initial thread (that started main).
- If **main** terminates before other threads without calling **pthread_exit**, the entire process is terminated.
- If **pthread_exit** is called in **main** the process continues executing until the last thread terminates (or **exit** is called.).

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

#define NUM_THREADS 10;
```

```c

/* takes a void pointer, which contains the id*/
void *thread_func(void *id) {
    printf("This is thread: %d!\n", (int)id);
}

int main (int argc, char** argv) {
    /* array containing threads */
    pthread_t threads[NUM_THREADS];

    /* create each thread and start */
    for (int id = 0; id < NUM_THREADS; id++)
        pthread_create(&threads[id], NULL, thread_func, (void *)id);

    /* process continues until all threads have terminated */
    pthread_exit(NULL);
}
```

Yielding Thread

```c
int pthread_yield (void)
```

Releases the **CPU** to let another thread run. Returns 0 on success. In linux it always succeeds.

Joining Threads

```c
int pthread_join (pthread_t thread, void **value_ptr)
```

Blocks until the thread terminates. The value passed to **pthread_exit** is available on location referenced by **value_ptr** (can be **NULL**) for example an error value may be propagated.

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

void *thread_1_work(void *id) {
    printf("Thread 1 is working");

    /* code for task goes here */

    printf("Thread 1 is done!");
}

void *thread_2_work(void *id) {
    printf("Thread 2 is working");

    /* code for task goes here */

    printf("Thread 2 is done!");
}

int main (int argc, char** argv) {
    /* array containing threads */
    pthread_t thread_1, thread_2;

    printf("Both threads ordered to work!");

    pthread_create(&thread_1, NULL, thread_1_work, NULL);
```

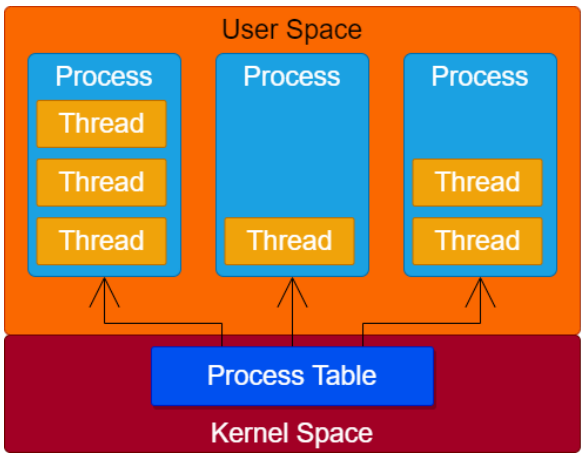
```
27      pthread_create(&thread_2, NULL, thread_1_work, NULL);
28
29      /* Join both threads to current thread. */
30      pthread_join(&thread_1, NULL);
31      pthread_join(&thread_2, NULL);
32
33      /* both have completed their work! */
34      printf("Both threads done working!");
35  }
```

OS Thread Implementation

The two main approaches are **User-Level Threads** and **Kernel-Level Threads**. Both have tradeoffs and hybrid approaches are possible.

User Level Threads

OS Kernel is manages processes only. Threads are implemented through a library, with running processes maintaining their own thread table to manage their threads.



Advantages

- **Better Performance**
  As the kernel is not involved in thread creation, termination, switching or synchronisation. They can be much faster.

  Fewer expensive context switches as less kernel involvement.

- **Optimisation**
  Each application can have its own scheduling algorithm, which can be optimised for the specific context of that application. Which aides performance.

Disadvantages

- **Blocking Calls** A blocking system call blocks the process (and all threads inside), which denies a key motivation for using threads.

While non-blocking I/O can be used (e.g **select**) they are more complex & inelegant.

- **Exceptions**
  Exceptions can block all threads, even if some are runnable (e.g page-fault).

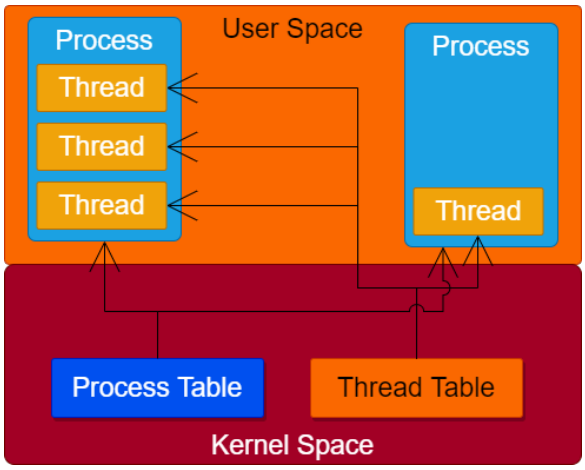
Select

select and **pselect** allow a program to monitor multiple file descriptors, waiting for some to become ready for some I/O before doing it.

```c
/* Example from tutorialspoint.com */
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    fd_set rfds;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input. */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;
    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");
    return 0;
}
```

Kernel Level Threads

The OS kernel manages the threads of processes directly.



Advantages

- **Blocking Calls**
  Blocking calls & exceptions (e.g page fault) do not block whole process, if one thread in a process is blocked, the kernel can schedule another runnable thread from that process.

Disadvantages

- **Thread Creation more expensive**
  Requires systemn calls (though is still cheaper than creating a new process).

  A mitigation stragetgy is to use thread pools (recycling threads).

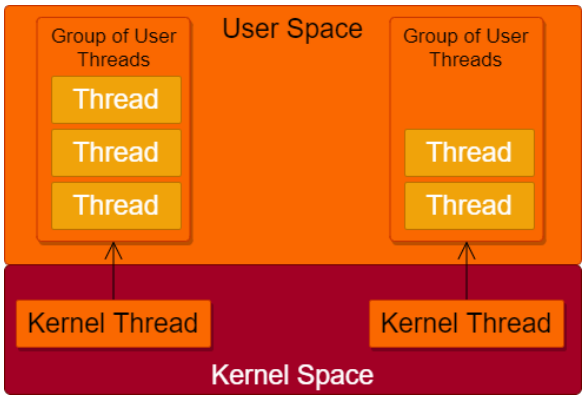
- **Synchronisation**
  Requires blocking system calls, which are expensive.

- **Switching more expensive**
  Requires a system call, kernel the schedule. Kernel scheduler be optimal for all applications (however still cheaper than switching processes).

- **No application-specific Schedulers!**

Hybrid Approaches

A suggested hybrid approach is to map user level threads, to a lower number of kernel threads. This has some of the advantages of both approaches, however requires some overhead in creating the mapping, and updating it.



The user level threads do not have 'real' concurrency in this approach, only kernel threads.