National University
of Computer & Emerging Sciences

Operating System

POSIX Threads

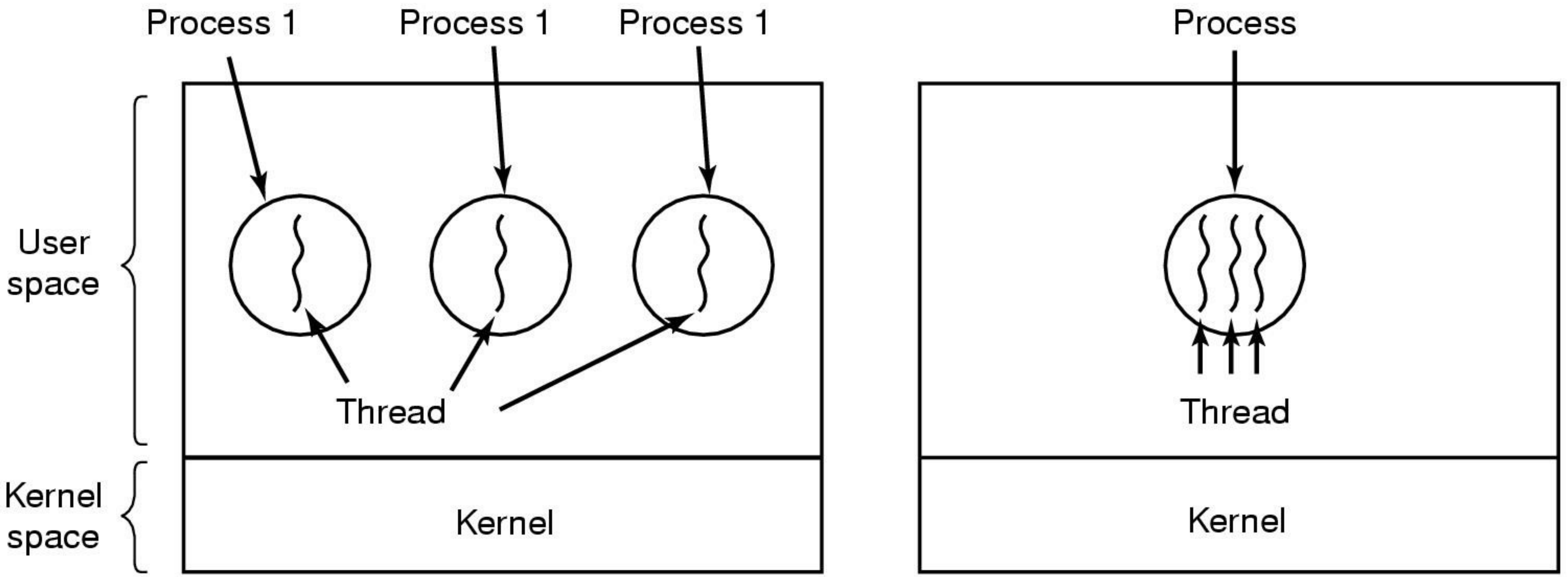
Threads

- Light weight processes
- Allow multiple execution paths in the same process environment
- The first thread starts execution with
  `int main(int argc, char *argv[])`
- The threads appear to the Scheduling part of an OS just like any other process

Thread state

- Each thread has its own stack and local variables
- Globals are shared.
- File descriptors are shared. If one thread closes a file, all other threads can't use
- The file I/O operations block the calling thread.
- For example, the exit() function operates terminates the entire and all associated threads.

Process Vs. Threads



(a) Three threads, each running in a separate address space

(b) Three threads, sharing the same address space

Its better to distinguish between the two concepts

Heavy weight process

Address space/Global Variables
Open files
Child processes
Accounting info
Signal handlers
Program counter
Registers
Stack
State

Split

In case of multiple thr per process

Light weight processes

Unit of Resource

Unit of Dispatc

Address space/Global Variables
Open files
Child processes
Accounting info
Signal handlers

Share

Program
cou
Reg
Sta
Sta

Program
counter
R
S
S

Program
counter
Registers
Stack
State

S. N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Thread Implementation

- POSIX is a standard API supported
- Portable across most UNIX platforms.
- PTHREAD library contains implementation of POSIX standard
- To link this library to your program use – ***lpthread***
  - `gcc  MyThreads.c  -o MyThreadExecutable - lpthread`

Thread Creation

- `pthread_create( pthread_t *threadid ,const pthread_attr_t *attr, void *(*start_routine)(void *),void *arg);`
- This routine creates a new thread and makes it executable.
- Thread stack is allocated and thread control block is created
- Once created, a thread may create other threads.
- Note that an "initial thread" exists by default and is the thread which runs main.
- Returns zero, if ok
- Returns Non-zero if error
- **threadid**
  - The routine returns the new thread ID via the **threadid**
  - The caller can use this thread ID to perform various operations
  - This ID should be checked to ensure that the thread was successfully created.

Thread Creation

- ```
  pthread_create( pthread_t *threadid
  ,const pthread_attr_t *attr, void
  *(*start_routine)(void *),void *arg);
  ```
- **attr**
  - used to set thread attributes.
  - NULL for the default values.
- **start_routine**
  - The C routine that the thread will execute once it is created.
- **arg**
  - Arguments are passed to *start_routine* via *arg*.
  - Arguments must be passed by reference as pointers
  - These pointers must be cast as pointers of type void.
  ```

Thread Termination

- `pthread_exit(status)`
- Several ways of termination:
  - The thread makes a call to the pthread_exit subroutine.
  - The thread is canceled by another thread via the pthread_cancel routine.
  - The entire process is terminated due to a call to the exit subroutines.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <iostream>
using namespace std;
void *PrintHello(void *threadid)
{
        cout<<" Hello World!\n" <<pthread_self()<<endl;
        pthread_exit(NULL);
}
int main(){
        pthread_t threads[5];
        int rc, t;
        for(t=0;t < 5;t++)
        {
                cout<<"Creating thread "<< t;
                rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
                if (rc)
                {
                cout<<"ERROR; return code from pthread_create() is "<< rc<<endl;
                exit(-1);
                }
        }
        pthread_exit(NULL);
        return 0;
}
```

Passing Arguments To Threads

- The pthread_create() routine permits the programmer to pass one argument to the thread start routine.
- What if we want to pass multiple arguments.
- Create a structure which contains all of the arguments
- Pass a pointer to the structure in the pthread_create() routine.
- Argument must be passed by reference and cast to (void *).
- Important:
  - Threads initially access their data structures in the parent thread's memory space.
  - That data structure must not be corrupted/modified until the thread has finished accessing it.

Correct pthread_create() argument passing

```
int *task_ids[NUM_THREADS];
for(t=0;t < NUM_THREADS;t++)
{
  task_ids[t] = new int;
     *task_ids[t] = t;
     printf("Creating thread %d\n", t);
     rc = pthread_create(&threads[t],
  NULL,   PrintHello, (void *)
  task_ids[t]);
     ...
}
```

Passing a structure as an argument

```
struct thread_data
{ int thread_id;
  int sum;
};
thread_data thread_data_array[NUM_THREADS];
void *PrintHello(void *threadarg)
{ thread_data *my_data;
  ...
  my_data = (struct thread_data *) threadarg;
  taskid = my_data->thread_id;
  sum = my_data->sum;
  ...
}
```

Passing a structure as an argument

```
int main()
{ ...
  thread_data_array[t].thread_id = t;
  thread_data_array[t].sum = sum;
  rc =
  pthread_create(&threads[t], NULL,
  PrintHello,
  (void *) &thread_data_array[t]);
  ...
}
```

Thread ID

- `pthread_self()`
- Returns the unique thread ID of the calling thread.
- `pthread_equal(threadid1,threadid2)`
- Compares two thread IDs:
- If the two IDs are different 0 is returned, otherwise a non-zero value is returned.

Thread Suspension and Termination

- Similar to UNIX processes, threads have the equivalent of the wait() and exit() system calls
- pthread_join()
  - Used to block threads
- To instruct a thread to block and wait for a thread to complete, use the **pthread_join()** function.
- This is also the mechanism used to get a return value from a thread
- Any thread can call join on (and hence wait for) any other thread.

Joining thread

- **Joinable:** on thread termination the thread ID and exit status are saved by the OS.
- Joining a thread means waiting for a thread

pthread_join(threadid, status)

- "Joining" is one way to accomplish synchronization between threads.
- The pthread_join() subroutine blocks the calling thread until the specified *threadid* thread terminates.
- The programmer is able to obtain the target thread's termination return status (if specified) in the *status* parameter.

Joining thread

- Multiple threads cannot wait for the same thread to terminate.
- If they try to, one thread returns successfully
- The others fail with an error of ESRCH
- After pthread_join() returns, any stack storage associated with the thread can be reclaimed by the application.
- Their resources cannot be fully recovered.

pthread_detach()

- By default threads are created joinable.
- Instead of waiting for a child, a parent thread can specify that
  - it does not require a return value
  - or any explicit synchronization with that thread.
- To do this, the parent thread uses the **pthread_detach()** function.
- After this call, there is no thread waiting for the child – it executes independently until termination.
- To avoid memory leaks a thread should either be *joined* Or detached by a call to **pthread_detach()**
- int pthread_detach(pthread_t tid);
- Returns 0 on OK, nonzero on error
- Threads can detach themselves by calling *pthread_detach* with an argument of *pthread_self*