PTHREAD PROGRAMMING

What is Threading?

- Threading is the creation and management of multiple units of execution within a single process.
- Threading is a significant source of programming error, through the introduction of data races and deadlocks

Binaries, Processes, and Threads

- *Binaries* are dormant programs residing on a storage medium, ready to execute but not yet in motion.
- *Processes* are the operating system abstraction representing those binaries in action.
- *Threads* are the unit of execution within a process.
- If a process contains but one thread, there is only a single unit of execution in the process. We call such processes *single threaded*.
- If a process contains more than one thread, then there is more than one thing going on at once. We call such processes *multithreaded.*

Two Fundamental Virtualized Abstractions

- virtual memory and
- a virtualized processor
- Together, they give the illusion to each running process that it alone consumes the machine’s resources.
- Virtualized memory is associated with the process and not the thread.
- Conversely, a virtualized processor is associated with threads and not processes.

- A thread consists of the information necessary to represent an execution context within a process.
- This includes a thread ID that identifies the thread within a process, a set of register values, a stack, a scheduling priority and policy, a signal mask, an errno variable , and thread-specific data.
- The threads interfaces we're about to see are from POSIX.1-2001. The threads interfaces, also known as ''pthreads'' for ''POSIX threads,''.

Thread Identification

- Just as every process has a process ID, every thread has a thread ID.
- Unlike the process ID, which is unique in the system, the thread ID has significance only within the context of the process to which it belongs.
- Recall that a process ID, represented by the pid_t data type, is a non-negative integer.
- A thread ID is represented by the *pthread_t* data type.

The Functions

- Checking equality

```
#include <pthread.h>
int pthread_equal(pthread_t tid1 , pthread_t tid2 );
                                        Returns: nonzero if equal, 0 otherwise
```

- A thread can obtain its own thread ID by calling the pthread_self function.

```
#include <pthread.h>
pthread_t pthread_self(void);
                                        Returns: the thread ID of the calling thread
```

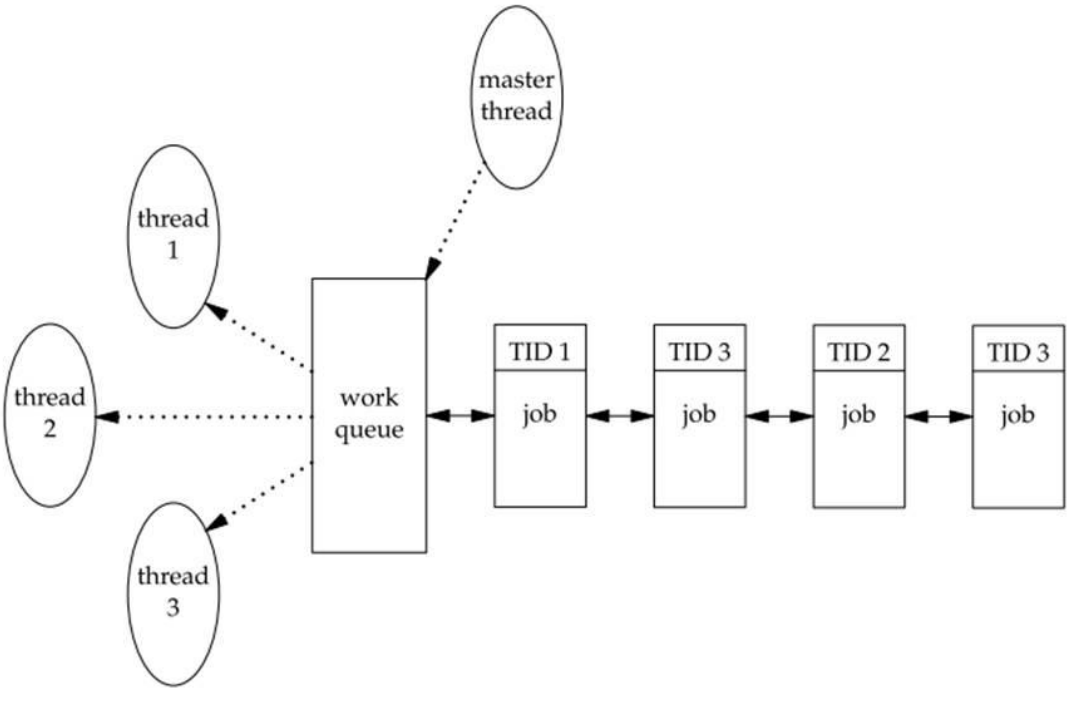
Thread Creation

- threads can be created by calling the pthread_create function.

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tidp ,
const pthread_attr_t *restrict attr , void *(* start_rtn )(void *), void *restrict arg );
                                                  Returns: 0 if OK, error number on failure
```

Attributes

- The memory location pointed to by *tidp* is set to the thread ID of the newly created thread when pthread_create returns successfully.
- The *attr* argument is used to customize various thread attributes.
- The newly created thread starts running at the address of the start_rtn function.
- This function takes a single argument, arg, which is a typeless pointer.



Example

```
#include <pthread.h>

pthread_t ntid;
void printids(const char *s)
{
pid_t      pid;
pthread_t  tid;
pid = getpid();
tid = pthread_self();
printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
(unsigned long)tid, (unsigned long)tid);
}
void * thr_fn(void *arg)
{
printids("new thread: ");

int main(void)
{
int    err;
err = pthread_create(&ntid, NULL, thr_fn,
    NULL);
if (err != 0)
err_exit(err, "can’t create thread");
printids("main thread:");
sleep(1);
exit(0);
}
```

Thread Termination

- If any thread within a process calls exit , _Exit , or _exit , then the entire process terminates.
- A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.
  - The thread can simply return from the start routine. The return value is thethread's exit code.
  - The thread can be canceled by another thread in the same process.

The pthread_exit( )

```
#include <pthread.h>
void pthread_exit(void * rval_ptr );
```

- The rval_ptr argument is a typeless pointer, similar to the single argument passed to the start routine.

Joining Threads

- This pointer is available to other threads in the process by calling the
- pthread_join function.

```
#include <pthread.h>
int pthread_join(pthread_t thread , void ** rval_ptr );
                                   Returns: 0 if OK, error number on failure
```

- The calling thread will block until the specified thread calls *pthread_exit* , returns from its start routine, or is canceled.
- If the thread simply returned from its start routine, rval_ptr will contain the return code.
- If the thread was canceled, the memory location specified

Demonstrating Exit code

```
#include <pthread.h>
void * thr_fn1(void *arg)
{
printf("thread 1 returning\n");
return((void *)1);
}
void * thr_fn2(void *arg)
{
printf("thread 2 exiting\n");
pthread_exit((void *)2);
}
```

```
int main(void)
{
int  err;
pthread_t   tid1, tid2;
void  *tret;
err = pthread_create(&tid1, NULL, thr_fn1, NULL);
if (err != 0)
printf( "can’t create thread 1");
err = pthread_create(&tid2, NULL, thr_fn2, NULL);
if (err != 0)
printf( "can’t create thread 2");
err = pthread_join(tid1, &tret);
if (err != 0)
printf( "can’t join with thread 1");
printf("thread 1 exit code %ld\n", (long)tret);
err = pthread_join(tid2, &tret);
if (err != 0)
printf("can’t join with thread 2");
printf("thread 2 exit code %ld\n", (long)tret);
exit(0);
```

Passing more than one value

- The typeless pointer passed to pthread_create and pthread_exit can be used to pass more than a single value.
- The pointer can be used to pass the address of a structure containing more complex information.
- Be careful that the memory used for the structure is still valid when the caller has completed.

Example

```
#include<pthread.h>
struct foo {
int a, b, c, d;
};
Void printfoo(const char *s, const struct foo *fp)
{
printf("%s", s);
printf("  structure at 0x%lx\n", (unsigned long)fp);
printf("  foo.a = %d\n", fp->a);
printf("  foo.b = %d\n", fp->b);
printf("  foo.c = %d\n", fp->c);
printf("  foo.d = %d\n", fp->d);
}
void * thr_fn1(void *arg)
{
struct foo  foo = {1, 2, 3, 4};
printfoo("thread 1:\n", &foo);
pthread_exit((void *)&foo);
}
void * thr_fn2(void *arg)
{
```

```
int main(void)
{
int  err;
pthread_t   tid1, tid2;
struct foo   *fp;
err = pthread_create(&tid1, NULL, thr_fn1, NULL);
if (err != 0)
printf("can't create thread 1");
err = pthread_join(tid1, (void *)&fp);
if (err != 0)
printf("can't join with thread 1");
sleep(1);
printf("parent starting second thread\n");
err = pthread_create(&tid2, NULL, thr_fn2, NULL);
if (err != 0)
printf("can't create thread 2");
sleep(1);
printfoo("parent:\n", fp);
```

Cancelling Threads

- One thread can request that another in the same process be canceled by calling the pthread_cancel function.

```
#include <pthread.h>
int pthread_cancel(pthread_t tid );
                                        Returns: 0 if OK, error number on failure
```

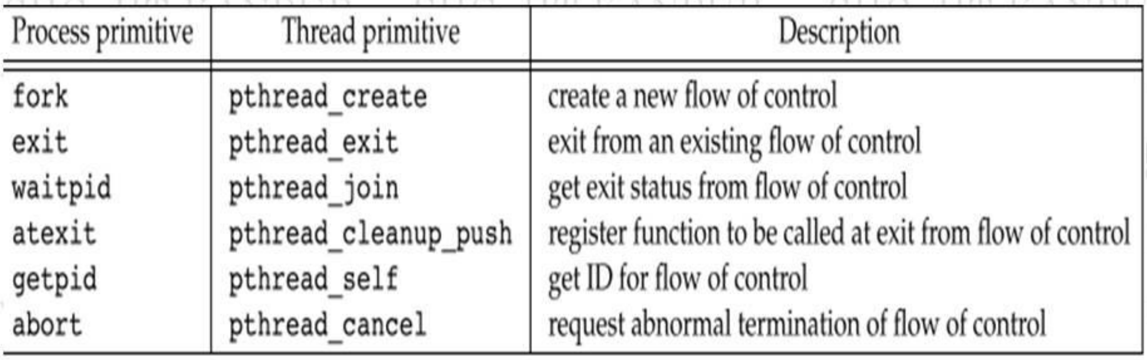
Thread cleanup handlers.

- A thread can arrange for functions to be called when it exits, similar to the way that the atexit function can be used by a process to arrange that functions are to be called when the process exits.
- The functions are known as *thread cleanup handlers*.
- More than one cleanup handler can be established for a thread.
- *The handlers are recorded in a stack*, which

```
#include <pthread.h>
void pthread_cleanup_push(void (* rtn )(void *), void * arg );
void pthread_cleanup_pop(int execute );
```

- The pthread_cleanup_push function schedules the cleanup function, rtn, to be called with the single argument, arg, when the thread performs one of the following actions:
    - Makes a call to pthread_exit
    - Responds to a cancellation request
    - Makes a call to pthread_cleanup_pop with a nonzero execute argument

Similarities between thread and Process functions



Process primitive	Thread primitive	Description
fork	pthread_create	create a new flow of control
exit	pthread_exit	exit from an existing flow of control
waitpid	pthread_join	get exit status from flow of control
atexit	pthread_cleanup_push	register function to be called at exit from flow of control
getpid	pthread_self	get ID for flow of control
abort	pthread_cancel	request abnormal termination of flow of control