OSLab: Threads

Sourangshu Bhattacharya

What are threads ?

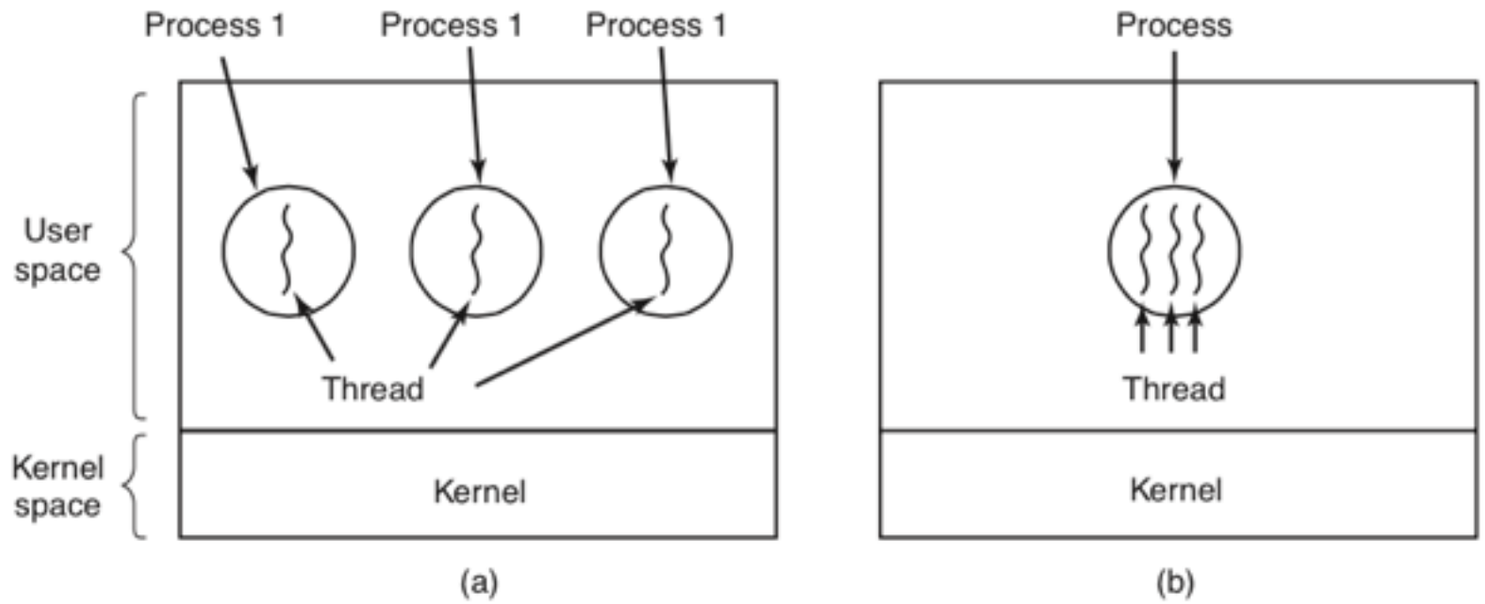
- **Wikipedia:** A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

- “Processes within processes”

- “Lightweight processes”

Process vs Thread

a single-threaded process = resource + execution
a multi-threaded process = resource + executions



- A process =a unit of resource ownership, used to group resources together

- A thread = a unit of scheduling, scheduled for execution on the CPU.

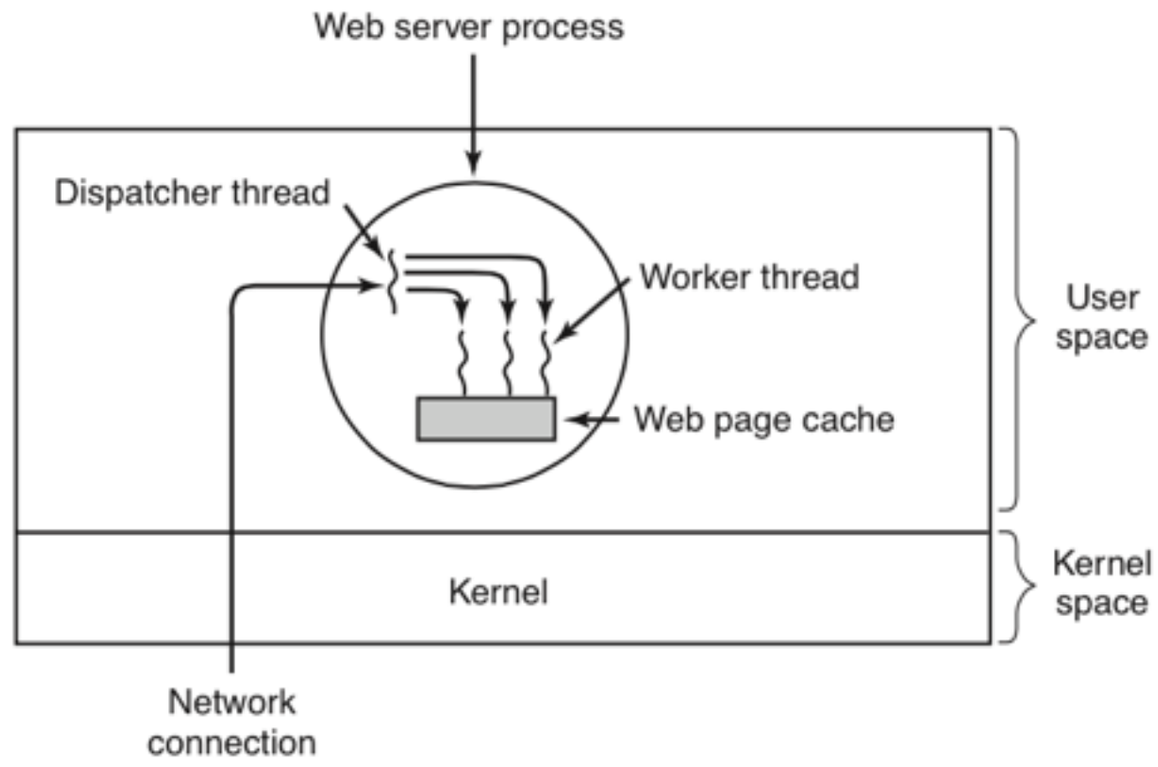
Process vs Thread

Threads share resources	Processes share devices
Memory space	CPU, disk,
File pointers	memory, printers
…	…

Threads own	Processes Own
Program counter	Threads +
Registers	Memory space
Stack	File pointers
…	…

- All threads of a process have same user. Hence no protection among threads.

Multi-threaded web server



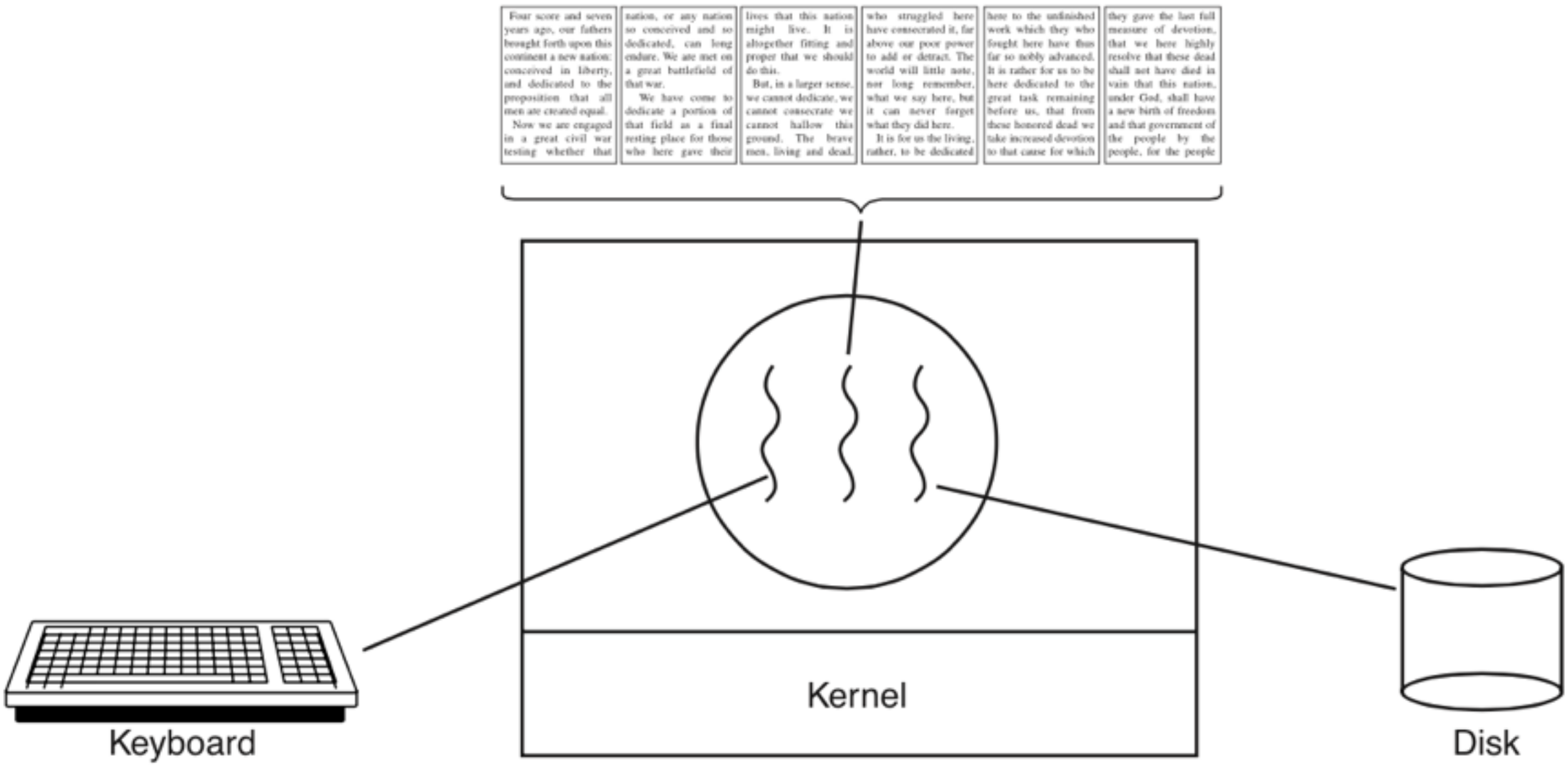

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Multi-threaded editor

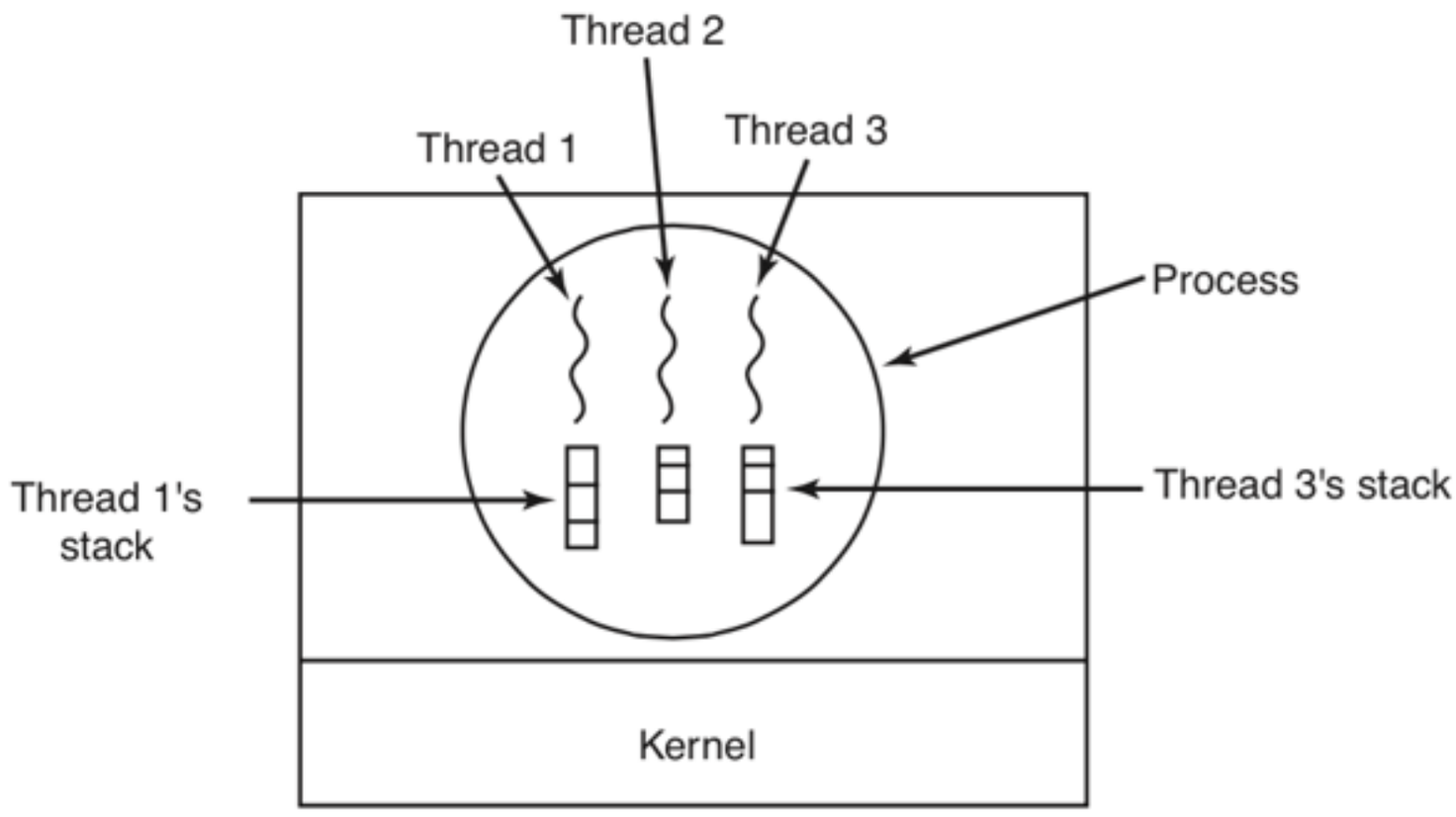


Advantages of multi-threading

- Parallelisation: Use multiple cores / cpus. e.g. multithreaded matrix multiplication.
- Responsiveness: Longer running tasks can be run in a worker thread. The main thread remains responsive e.g. editor.
- Cheaper: Less resource intensive than processes both memory and time.
- Simpler sharing: IPC harder and more time consuming.
- Better system utilisation: jobs finish faster.

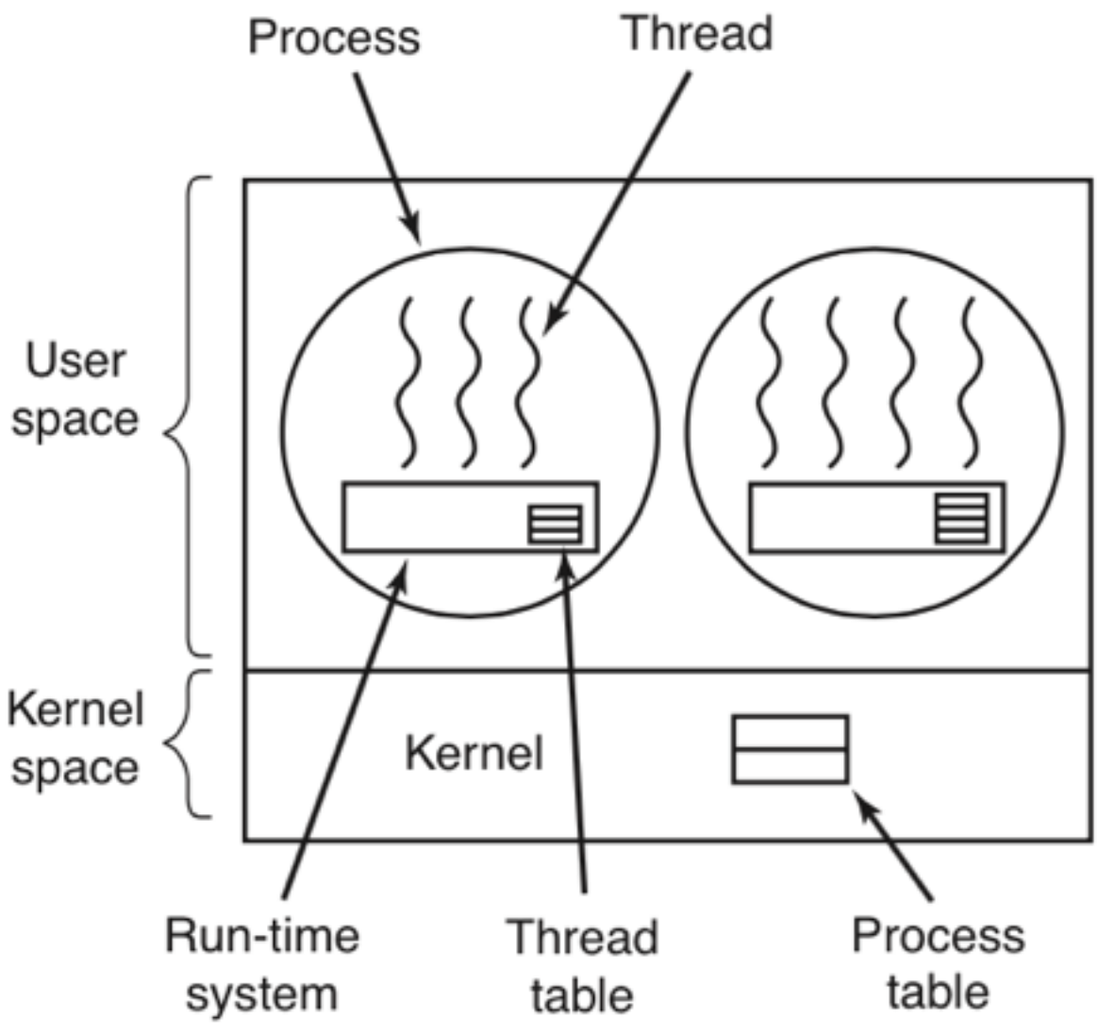
Each thread has own stack

- Stores data local to function. Can take advantage of functions, recursion, etc.
- Stack is destroyed when the thread exits.

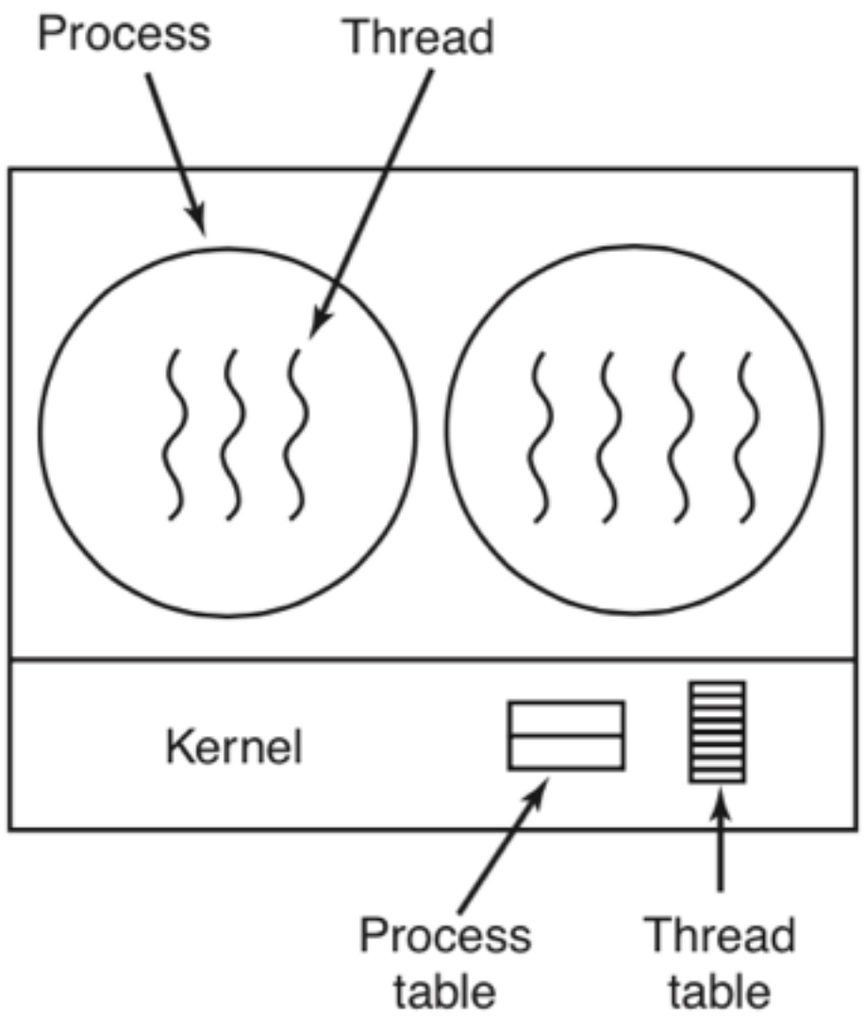


Thread implementation

User-level threads



Kernel-level threads



User-level threads

Advantages:

- ★ No dependency on OS - uniform behaviour.
- ★ Application specific thread scheduling.
- ★ Simple and fast - creation, switching, etc.

Disadvantages:

- ★ Entire process gets one time schedule.
- ★ Entire process gets blocked if one thread is blocked - requires non-blocking system calls.
- ★ Page fault in one thread can cause blocking, even though data for other threads are in memory.

Kernel-level threads

- Advantage: Kernel schedules threads independently - all above disadvantages are gone.

Disadvantages:

★ Overhead: more information per thread needs to be stored. Context switch is also slower.

★ Complexity: Kernel becomes more complex. Needs to handle thread scheduling, etc.

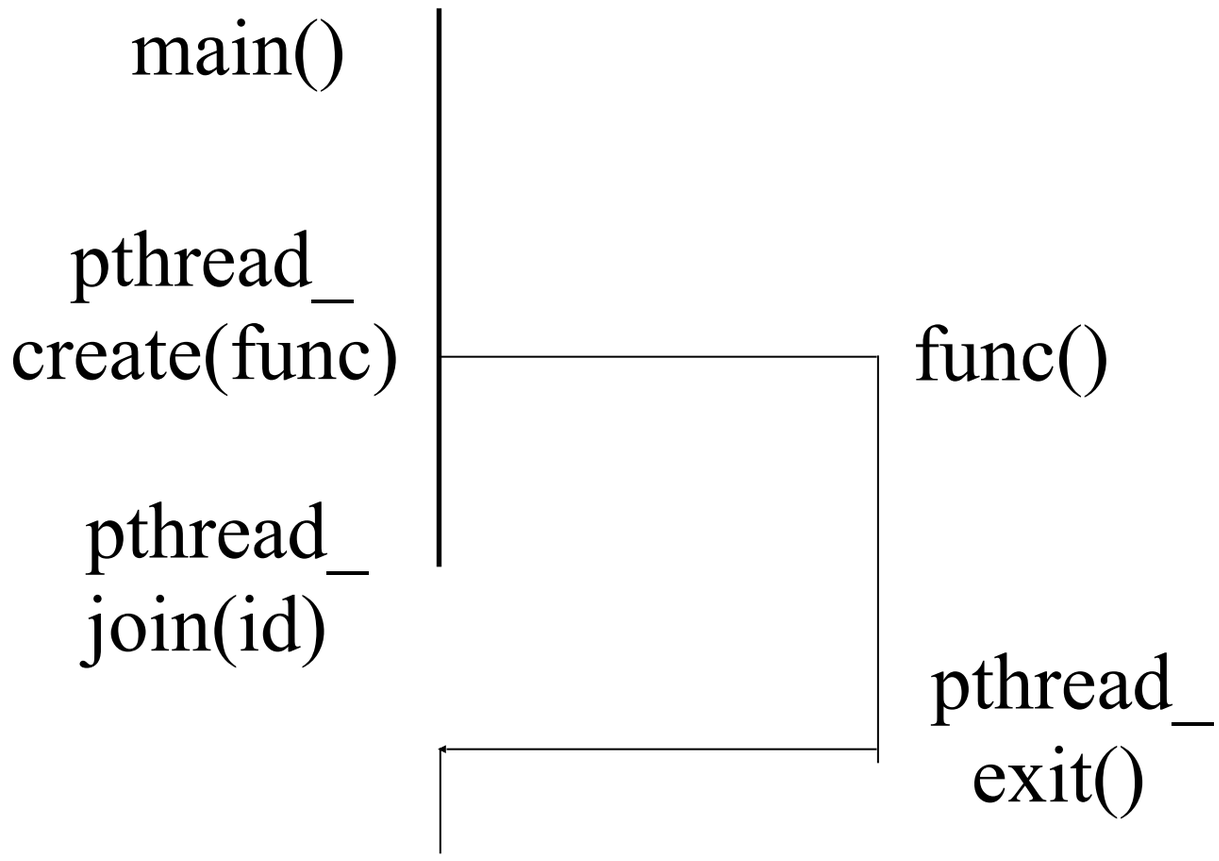
Hybrid implementations are possible !!

POSIX Threads

- IEEE 1003.1 c: The standard for writing portable threaded programs. The threads package it defines is called Pthreads, including over 60 function calls, supported by most UNIX systems.
- Some functions:

Thread call	**Description**
pthread_create	Create a new thread
pthread_exit	Terminate the calling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Release the CPU to let another thread run
pthread_attr_init	Create and initialize a thread's attribute structure
pthread_attr_destroy	Remove a thread's attribute structure

Typical structure



POSIX Threads Example

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

void *thread_function(void *arg){
  int i;
  for( i=0; i<20; i++ ){
    printf("Thread says hi!\n");
    sleep(1);
  }
  return NULL;
}

int main(void){
  pthread_t mythread;
  if(pthread_create(&mythread, NULL, thread_function, NULL)){
    printf("error creating thread.");
    abort();
  }

  if(pthread_join(mythread, NULL)){
    printf("error joining thread.");
    abort();
  }
  exit(0);
}
```

Pthread Mutex

- Access shared data in a thread safe manner.
- Typical sequence:
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed

Pthread Sync Example

- Preamble

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int      count = 0;
int      thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

Pthread Sync Example

```
void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %ld, count = %d  Threshold reached.\n",
             my_id, count);
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
           my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some "work" so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}
```

Pthread Sync Example

```
void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait
  routine will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.\n", my_id);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

Pthread Sync Example

```
int main (int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);

}
```