Computing Lab 1: Threads

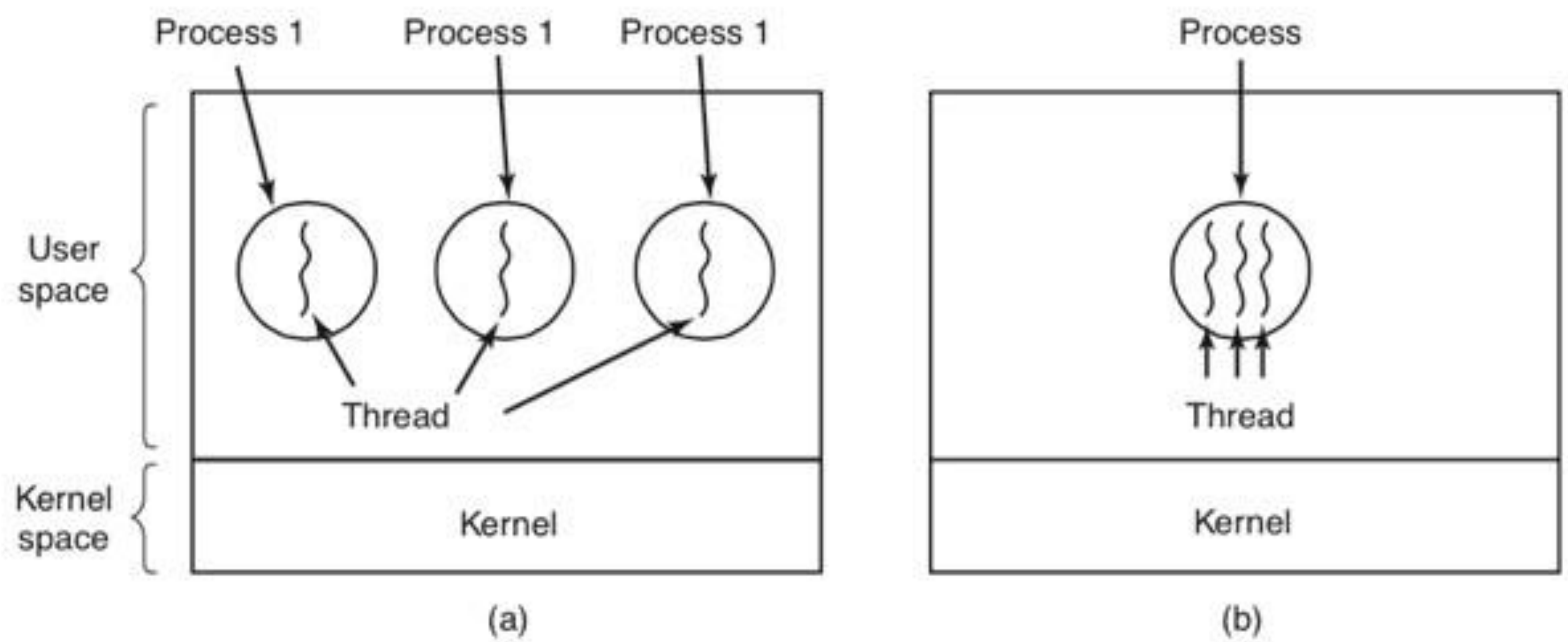
Sourangshu Bhattacharya

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.

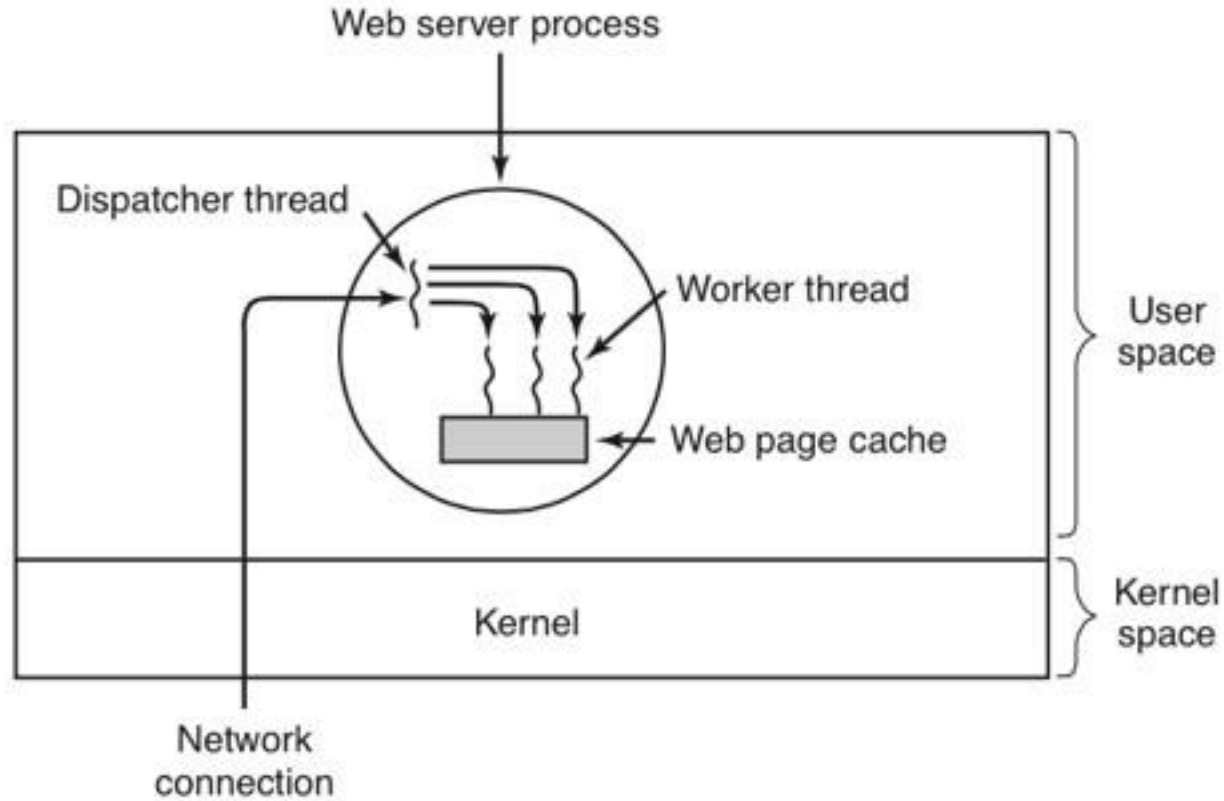
Process vs Thread

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

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

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

Multi-threaded web server



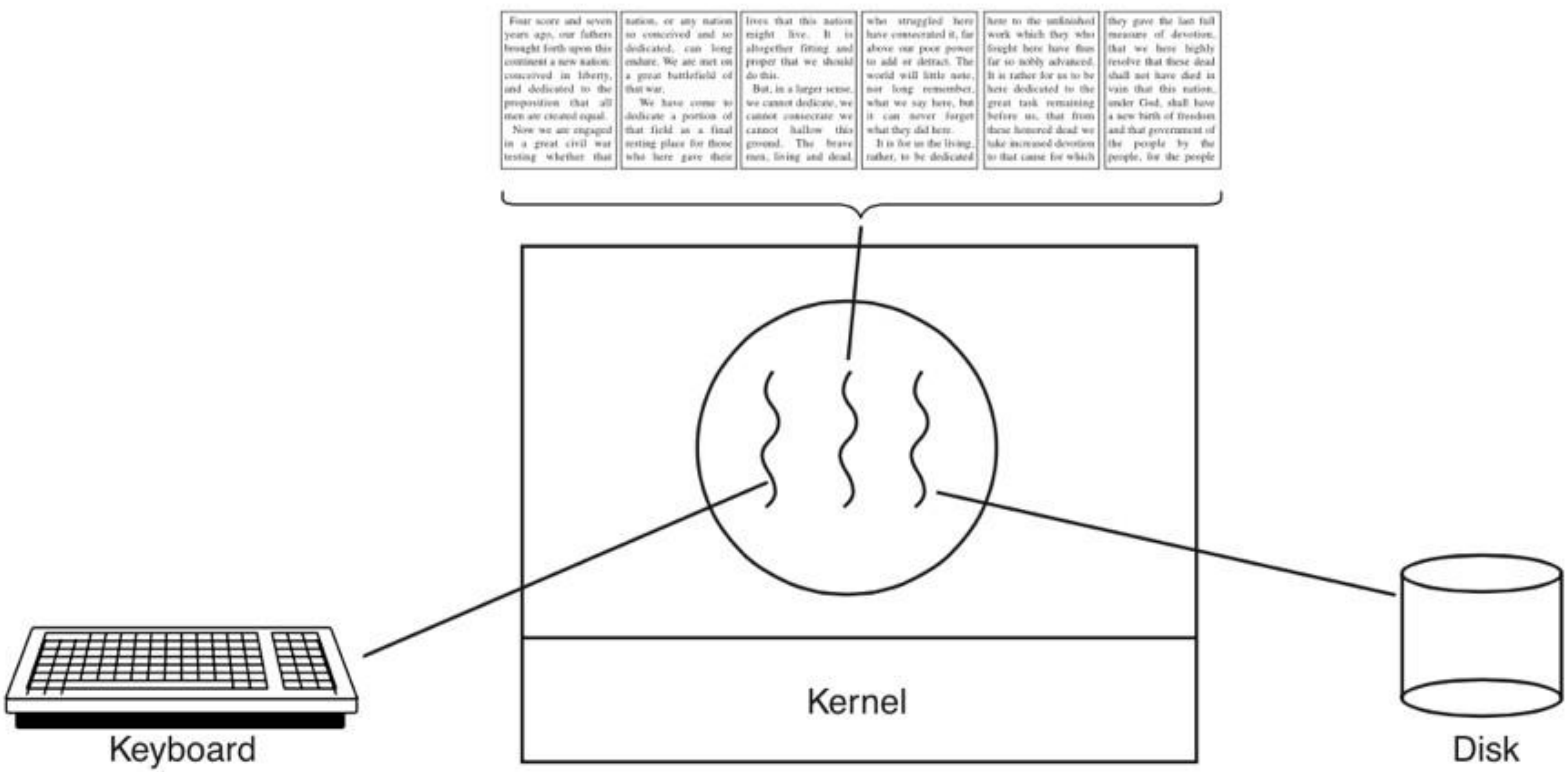

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(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

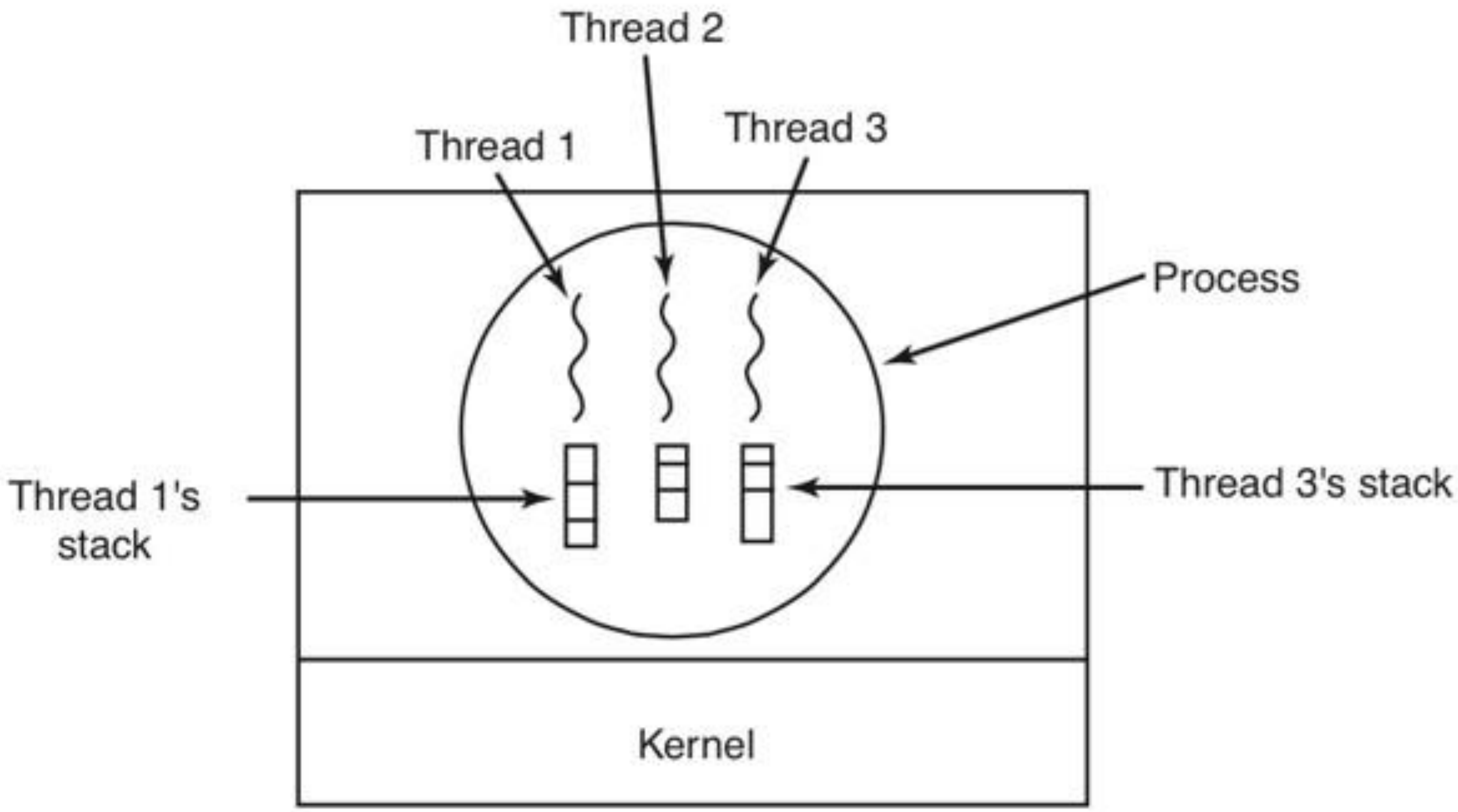


Advantages of multi-threading

- 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.

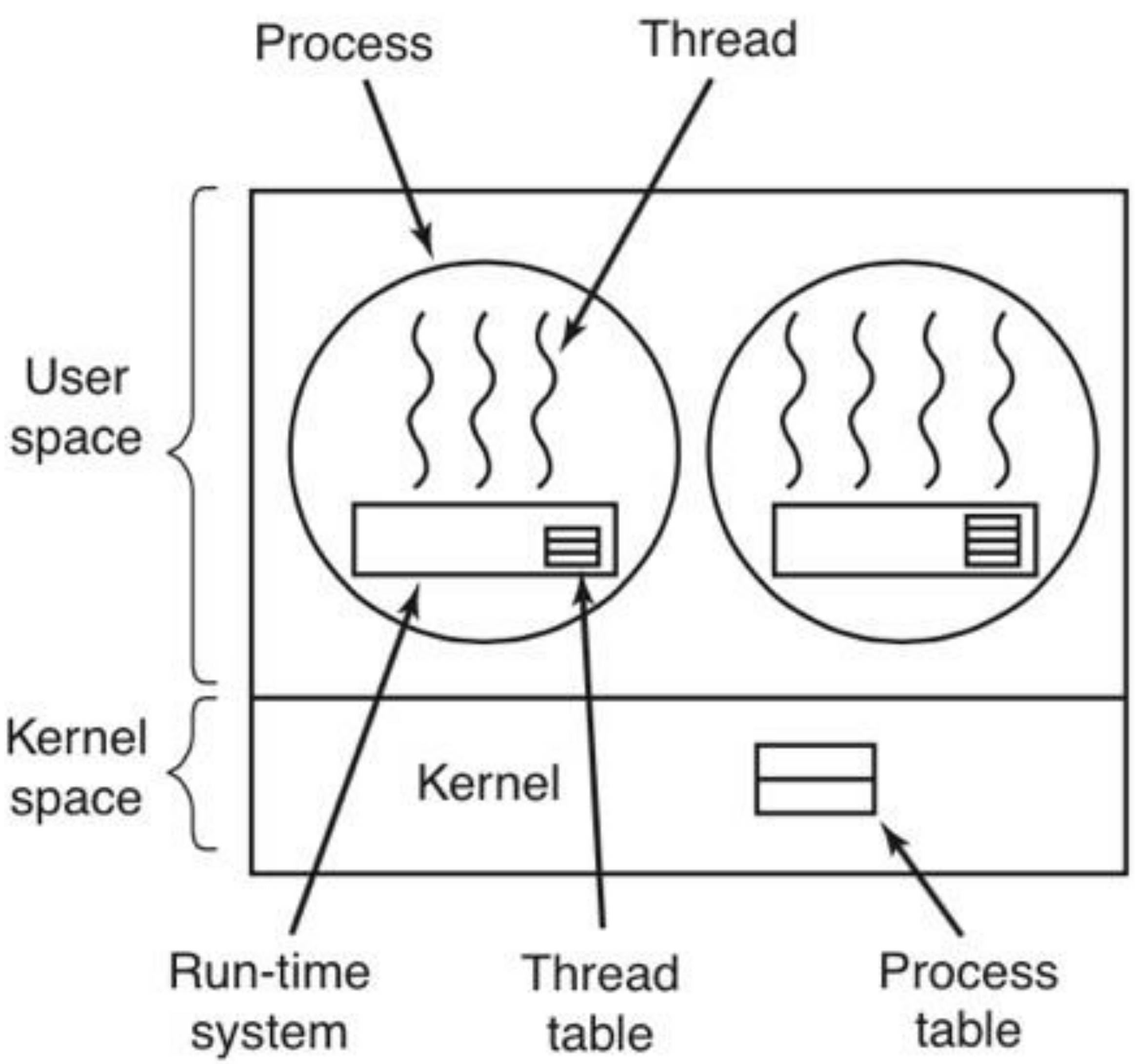
Each thread has own stack

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

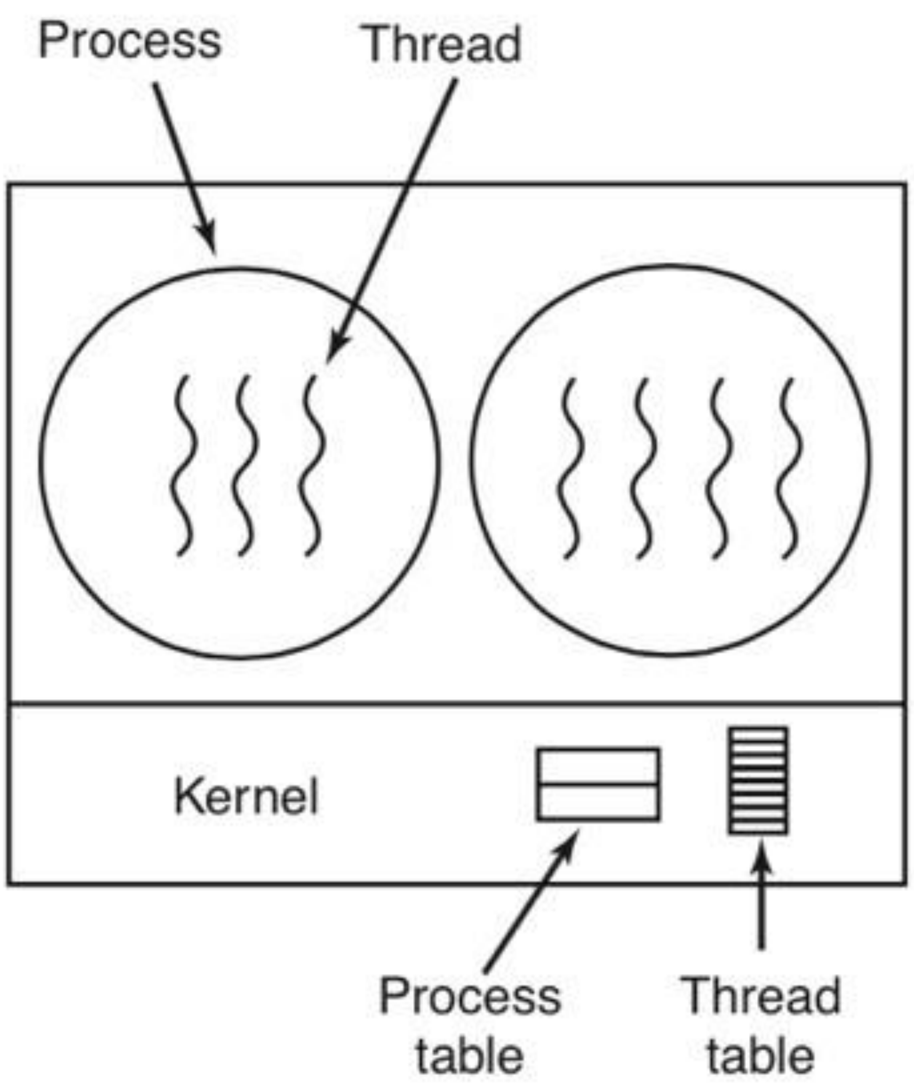


Thread implementation

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.

Examples: POSIX Threads, Java threads, etc.

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.

Examples: Solaris, Windows NT.

Hybrid implementations are possible !!

Linux

- For a set of user threads created in a user process, there is a set of corresponding LWPs in the kernel

```
os@os:~/os2018fall/code/4_thread/lwp1$ ./lwp1
LWP id is 20420
POSIX thread id is 0
```

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

int main()
{
    pthread_t tid = pthread_self();
    int sid = syscall(SYS_gettid);
    printf("LWP id is %dn", sid);
    printf("POSIX thread id is %dn", tid);
    return 0;
}
```

```
os@os:~$ ps -efL
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root         1     0     1  0    1 Oct13 ?        00:00:05 /sbin/init text
root         2     0     2  0    1 Oct13 ?        00:00:00 [kthreadd]
root         4     2     4  0    1 Oct13 ?        00:00:00 [kworker/0:0H]
root         6     2     6  0    1 Oct13 ?        00:00:00 [mm_percpu_wq]
root         7     2     7  0    1 Oct13 ?        00:00:00 [ksoftirqd/0]
root         8     2     8  0    1 Oct13 ?        00:00:02 [rcu_sched]
root         9     2     9  0    1 Oct13 ?        00:00:00 [rcu_bh]
root        10     2    10  0    1 Oct13 ?        00:00:00 [migration/0]
root        11     2    11  0    1 Oct13 ?        00:00:00 [watchdog/0]
root        12     2    12  0    1 Oct13 ?        00:00:00 [cpuhp/0]
root        13     2    13  0    1 Oct13 ?        00:00:00 [cpuhp/1]
root        14     2    14  0    1 Oct13 ?        00:00:00 [watchdog/1]
root        15     2    15  0    1 Oct13 ?        00:00:00 [migration/1]
root        16     2    16  0    1 Oct13 ?        00:00:00 [ksoftirqd/1]
root        18     2    18  0    1 Oct13 ?        00:00:00 [kworker/1:0H]

root       761     1   761  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   806  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   807  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   808  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   822  0    8 Oct13 ?        00:00:01 /usr/lib/snapd/snapd
root       761     1   823  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   824  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1  4293  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
```

Linux

NAME top

clone, __clone2 - create a child process

SYNOPSIS top

```
/* Prototype for the glibc wrapper function */

#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */ );

/* For the prototype of the raw system call, see NOTES */
```

DESCRIPTION top

clone() creates a new process, in a manner similar to fork(2).

This page describes both the glibc **clone**() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.

Unlike fork(2), **clone**() allows the child process to share parts of its execution context with the calling process, such as the virtual address space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of **CLONE_PARENT** below.)

One use of **clone**() is to implement threads: multiple flows of control in a program that run concurrently in a shared address space.

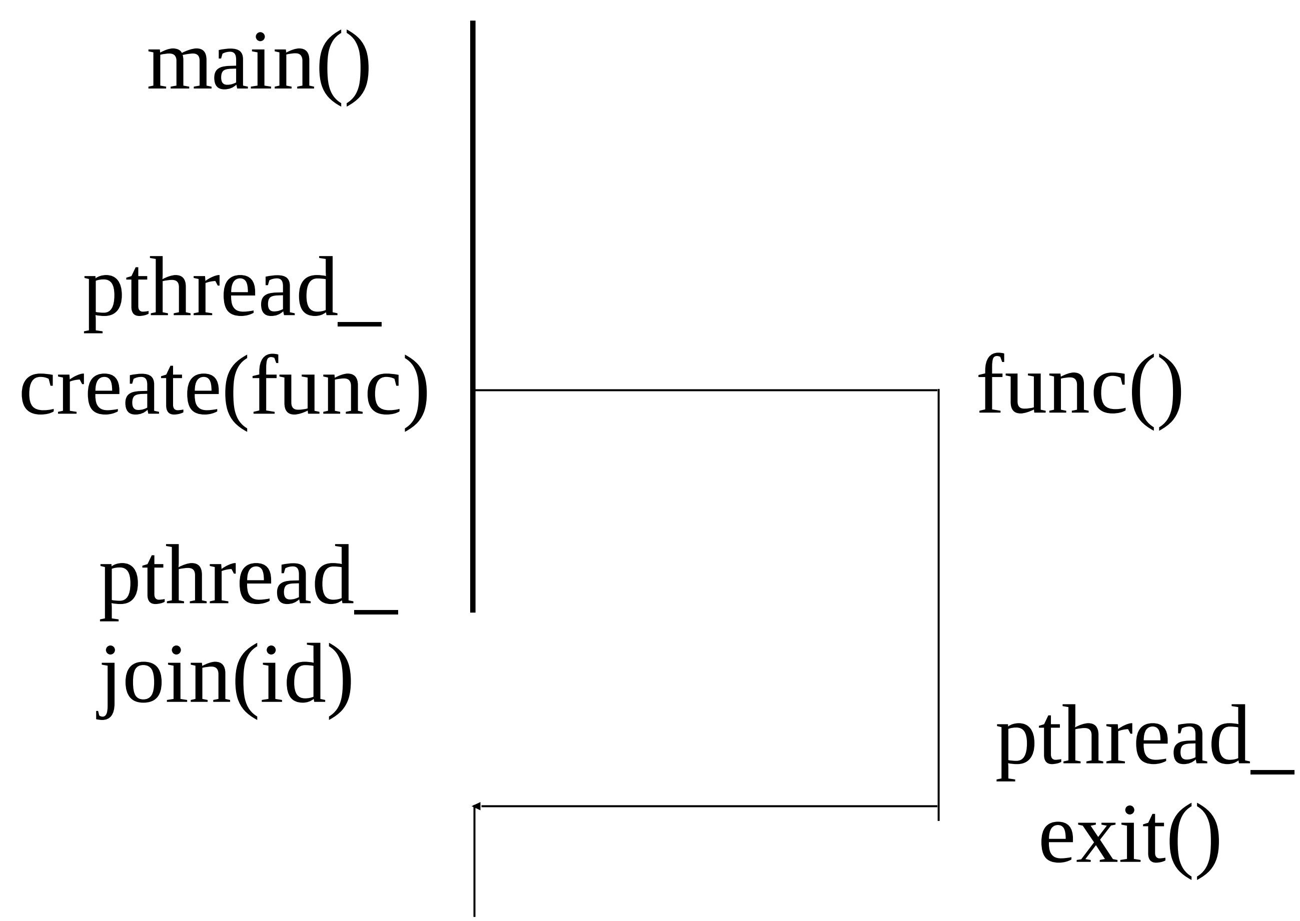
POSIX Threads

- 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



Thread creation

Types: `pthread_t` – type of a thread

Some calls:

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void *),
                   void *arg);
int pthread_join(pthread_t thread, void **status);
int pthread_detach();
void pthread_exit();
```

No explicit parent/child model, except main thread holds process info
Call `pthread_exit` in main, don't just fall through;
Most likely you wouldn't need `pthread_join`
`status` = exit value returned by joinable thread
Detached threads are those which cannot be joined (can also set this at creation)

POSIX Threads Example

Pthread Creation and Termination Example

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\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);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

POSIX Threads 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!
Hello World! It's me, thread #4!
```

Pthread Mutex

Type: `pthread_mutex_t`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Attributes: for shared mutexes/condition vars among processes, for priority inheritance, etc.

- use defaults

Important: Mutex scope must be visible to all threads!

Pthread Mutex

Using Mutexes Example

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

/*
The following structure contains the necessary information
to allow the function "dotprod" to access its input data and
place its output into the structure.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

```

Pthread Mutex

```
void *dotprod(void *arg)
{

   /* Define and use local variables for convenience */

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

   /*
   Perform the dot product and assign result
   to the appropriate variable in the structure.
   */

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   /*
   Lock a mutex prior to updating the value in the shared
   structure, and unlock it upon updating.
   */
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}
```

Pthread Mutex

```
int main (int argc, char *argv[])
{
   long i;
   double *a, *b;
   void *status;
   pthread_attr_t attr;

   /* Assign storage and initialize values */
   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i<VECLEN*NUMTHRDS; i++)
     {
     a[i]=1.0;
     b[i]=a[i];
     }

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;

   pthread_mutex_init(&mutexsum, NULL);

   /* Create threads to perform the dotproduct  */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

Pthread Mutex

```
    for(i=0; i<NUMTHRDS; i++)
    {
    /*
    Each thread works on a different set of data. The offset is specified
    by 'i'. The size of the data for each thread is indicated by VECLEN.
    */
    pthread_create(&callThd[i], &attr, dotprod, (void *)i);
    }

    pthread_attr_destroy(&attr);

    /* Wait on the other threads */
    for(i=0; i<NUMTHRDS; i++)
       {
       pthread_join(callThd[i], &status);
       }

    /* After joining, print out the results and cleanup */
    printf ("Sum =  %f \n", dotstr.sum);
    free (a);
    free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

Condition variables

- Used for condition - based synchronization between threads.
- Example: new data is available for a thread to compute.

Type `pthread_cond_t`

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                    pthread_mutex_t *mutex);
int pthread_cond_singal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Condition variables

- pthread_cond_init (condition,attr)
  Initialize condition variable.

- pthread_cond_destroy (condition)
  Destroy condition variable.

- pthread_cond_wait (condition,mutex)
  Wait on condition variable.

- pthread_cond_signal (condition)
  Wake up a random thread waiting on condition variable.

- pthread_cond_broadcast (condition)
  Wake up all threads waiting on condition variable.

Pthread Sync Example

- This simple example code demonstrates the use of several Pthread condition variable routines.
- The main routine creates three threads.
- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value

```
#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

```
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);

}
```

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);
}
```