

Threads

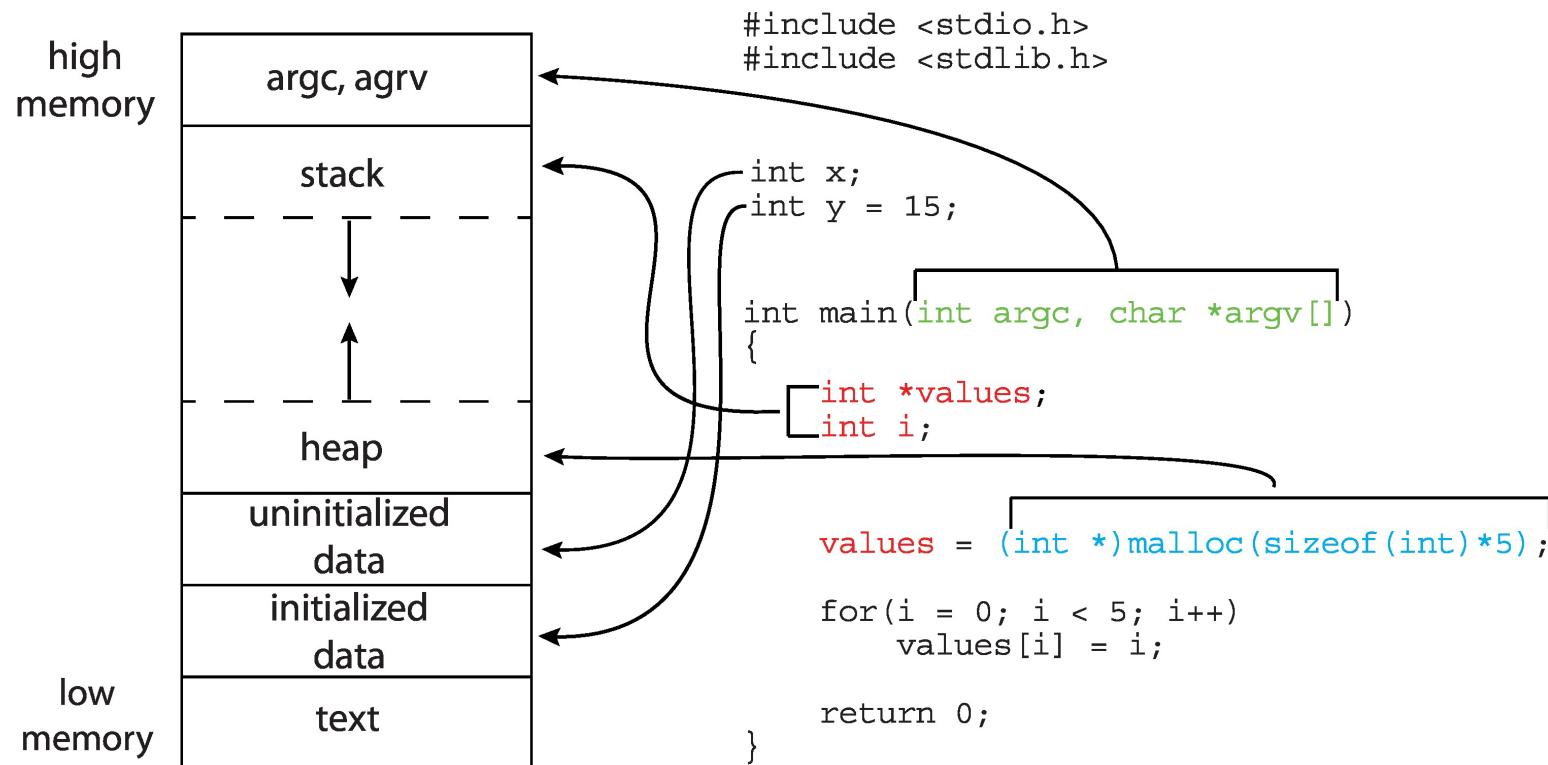


Operating Systems
Wenbo Shen

Revisit - Process Concept

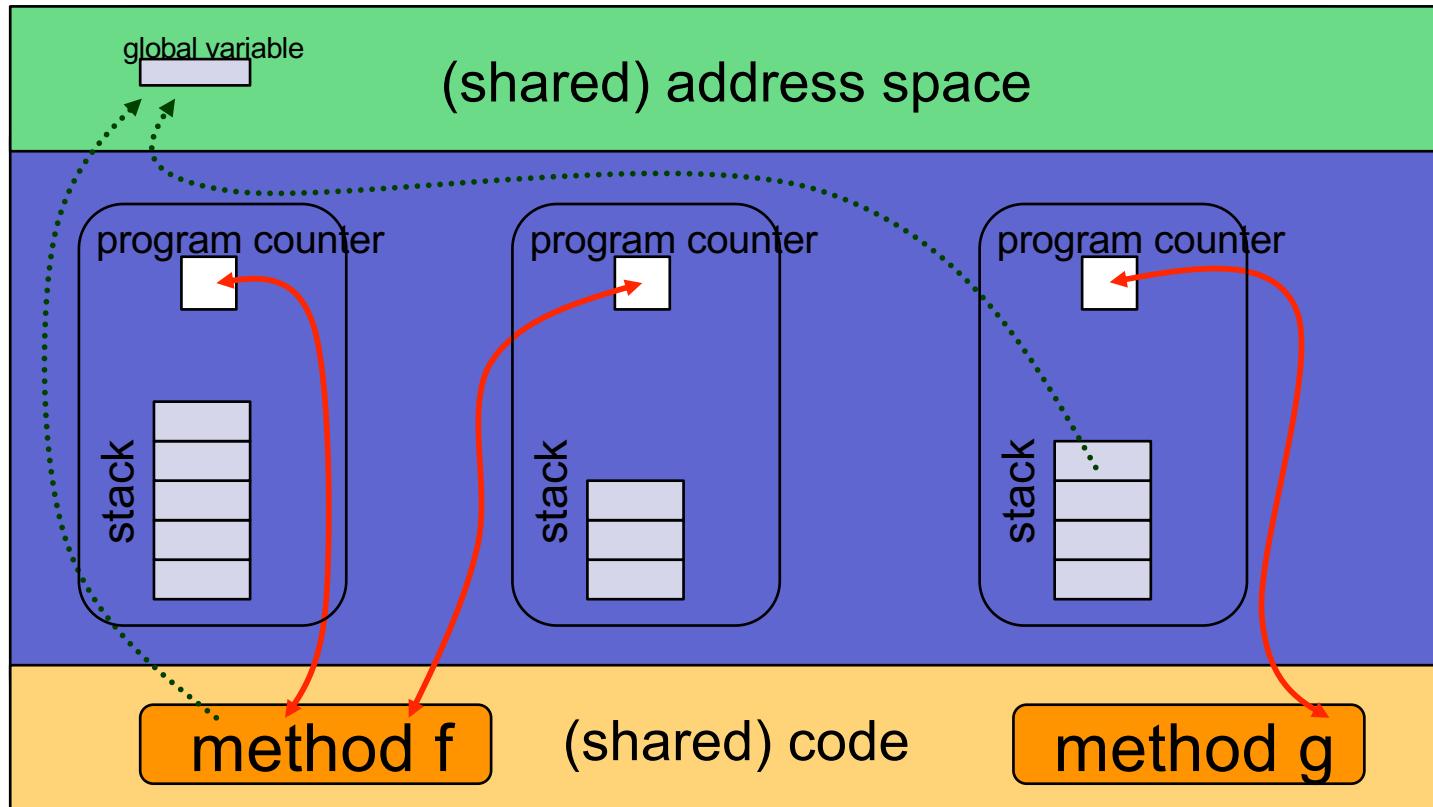
- Process =
 - code (also called the **text**)
 - initially stored on disk in an executable file
 - program counter
 - points to the next instruction to execute (i.e., an address in the code)
 - content of the processor's **registers**
 - a runtime **stack**
 - a **data section**
 - global variables (.bss and .data in x86 assembly)
 - a **heap**
 - for dynamically allocated memory (malloc, new, etc.)

Revisit - Memory Layout of a C Program



Why thread?

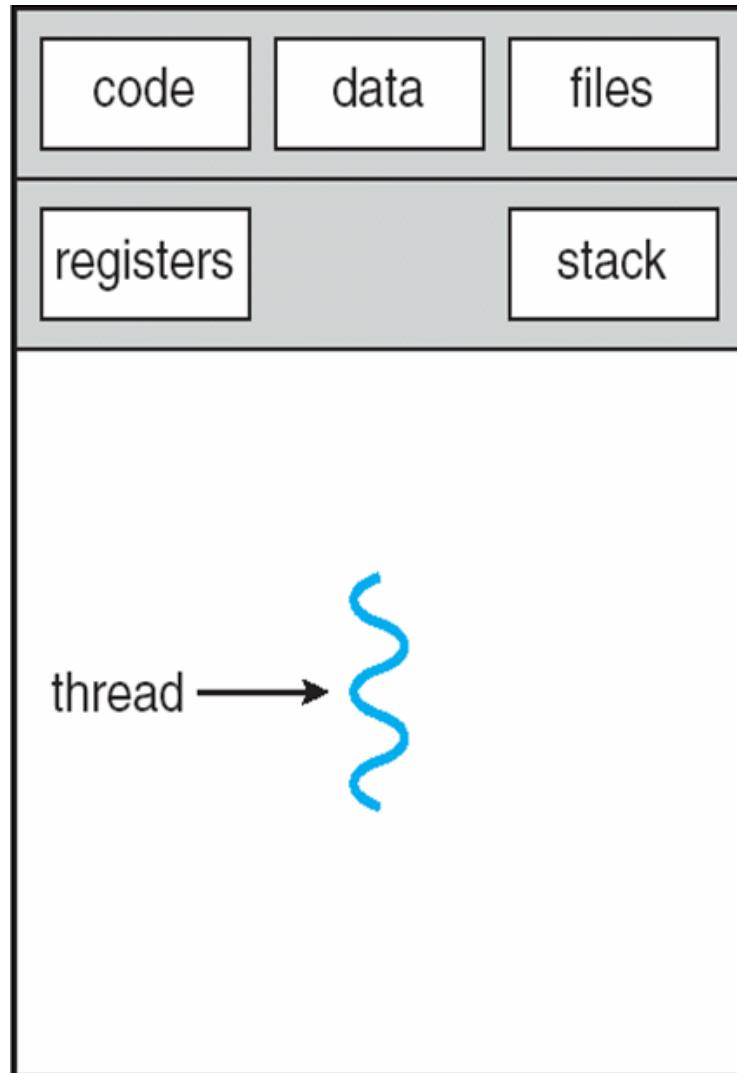
- How can we make a process run faster



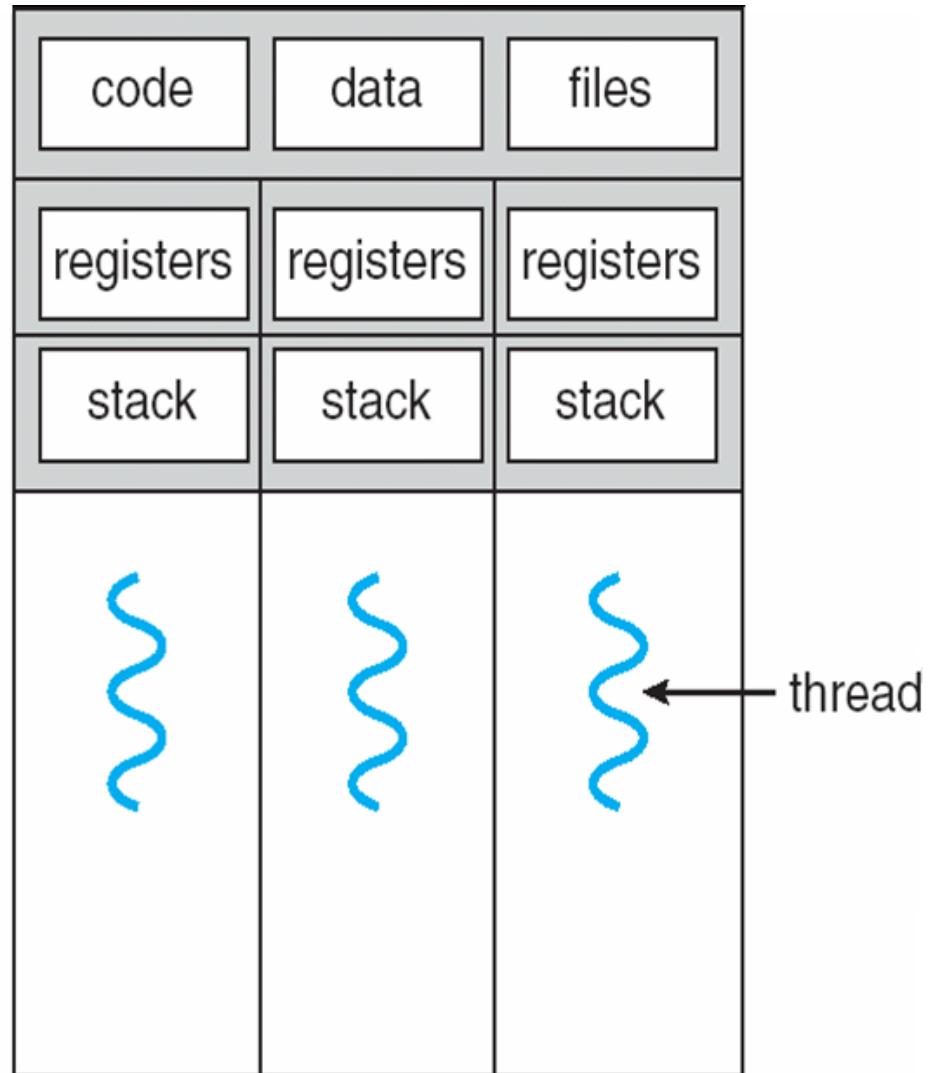
Thread Definition

- A thread is a basic unit of execution within a process
- Each thread has its own
 - thread ID
 - program counter
 - register set
 - Stack
- It shares the following with other threads within the same process
 - code section
 - data section
 - the heap (dynamically allocated memory)
 - open files and signals
- **Concurrency:** A multi-threaded process can do multiple things at once

The Typical Figure

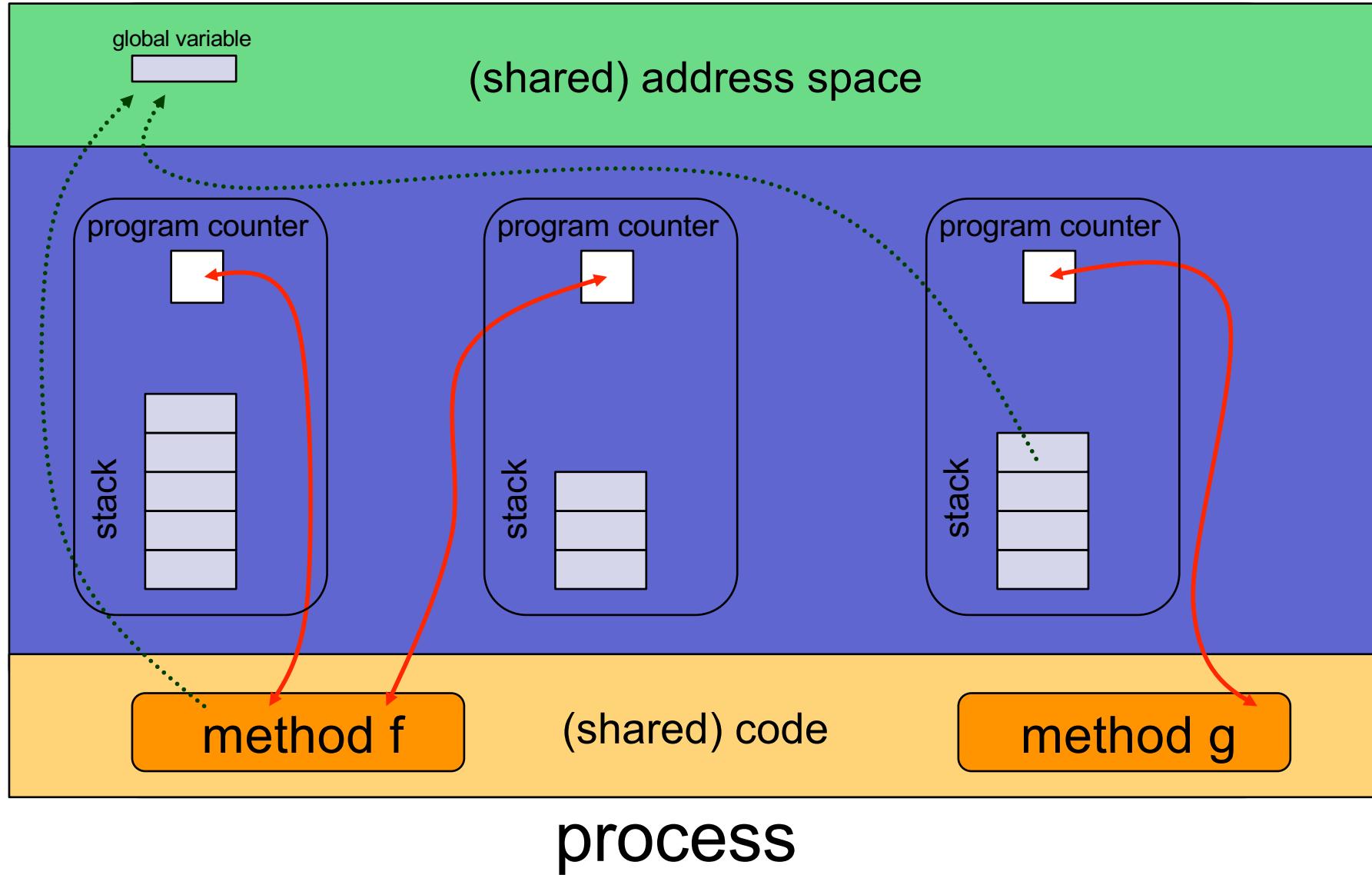


single-threaded process

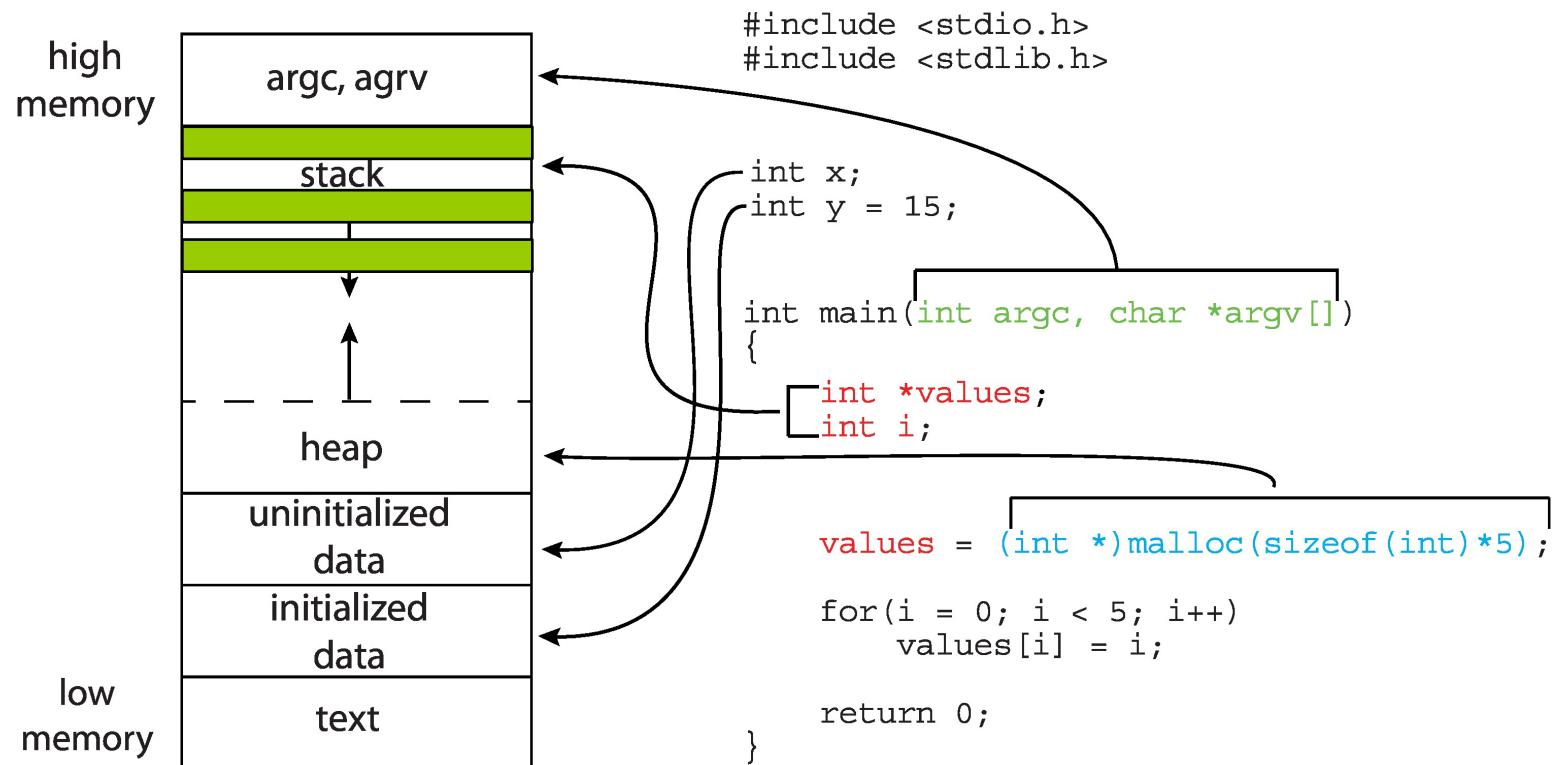


multithreaded process

A More Detailed Figure



Revisit - Memory Layout of a C Program



Multi-Threaded Program

- Source-code view
 - a blue thread
 - a red thread
 - a green thread

```

#include "submitrequest.h"
/* Lower-level function called by most of the functions below */
void submit_request(msghdr_t *host, scheduler_host_t *scheduler_host,
                    int id, int nodes,
                    double actual_duration, double requested_duration,
                    int port_start, int port_done, int port_queue_size) {
    job_request_t t;
    t.duration = id;
    t.job_id = id;
    t.nodes = nodes;
    t.requested_duration = requested_duration;
    t.actual_duration = actual_duration;
    t.port_start = port_start;
    t.port_end = port_end;
    t.port_queue_size = port_queue_size;
    t.to = MS_TASK_CREATE("job_start", 0, (void*)job_request);
    if (MS_TASK_PUT(t, host->task_sched, host->port, PORT_JOB_REQUEST, host->msg_host_get,
                    host->channel)) {
        return;
    }
}

/* submit_job()
 * submits the request to scheduler(s)
 */
void submit_job(msghdr_t *host, pending_job_t *pending_job,
                scheduler_host_t *scheduler_host,
                int num_schedulers,
                int port_start, int port_end, int cross_cluster_algorithm,
                int cross_cluster_percentage,
                int intra_cluster_algorithm, int intra_cluster_algorithm,
                int intra_cluster_percentage,
                int port_start, int port_end, int port_queue_size) {
    int i;
    int num_targetSchedulers;
    scheduler_info_t *targetSchedulers;
    /* Find out the list of open queue schedulers */
    ind_targetSchedulers(&targetSchedulers, numSchedulers, pendingJob->job_id,
                         pendingJob->requestedDuration,
                         pendingJob->actualDuration,
                         pendingJob->queueSize,
                         pendingJob->crossClusterAlgorithm,
                         pendingJob->crossClusterPercentage,
                         pendingJob->intraClusterAlgorithm,
                         pendingJob->intraClusterPercentage,
                         pendingJob->portStart,
                         pendingJob->portEnd,
                         pendingJob->portQueueSize);
    if (numTargetSchedulers == 0) {
        /* No target scheduler found for job */
        return;
    }

    /* Send requests to each target scheduler */
    for (i = 0; i < numTargetSchedulers; i++) {
        submit_job_to_target_scheduler(targetSchedulers[i], pendingJob,
                                       pendingJob->requestedDuration,
                                       pendingJob->actualDuration,
                                       pendingJob->queueSize,
                                       pendingJob->crossClusterAlgorithm,
                                       pendingJob->crossClusterPercentage,
                                       pendingJob->intraClusterAlgorithm,
                                       pendingJob->intraClusterPercentage,
                                       pendingJob->portStart,
                                       pendingJob->portEnd,
                                       pendingJob->portQueueSize);
    }
    free(targetSchedulers);
}

/* submit_job_to_target_scheduler()
 * With possible multiple requests to each
 */
void submit_job_to_target_scheduler(scheduler_info_t *scheduler,
                                   pending_job_t *pendingJob,
                                   int intracluster_algorithm,
                                   int intracluster_algorithm,
                                   int intracluster_percentage,
                                   int port_start, int port_end, int portqueue_size)
{
}

```

```

#include "printqueue.h"
void printqueue() {
    msghdr_t *host;
    job_descriptor_t *descriptor;
    int i;
    for (i = 0; i < host->queue_size; i++) {
        fprintf(stderr, "%d %d\n", i, host->queue[i].id);
        xbt_if_free(descriptor, host->queue[i].descriptor);
    }
}

/* start_job()
 * starts a job
 */
void start_job(job_descriptor_t *jd, scheduler_bookkeeping_t *bk) {
    /* Get time = MSG_get_clock(); */
    /* Init the jobimulator of a new job */
    auto_t task;
    task = MS_TASK_CREATE("job_started", jd->id, (void*)jd);
    if (MS_TASK_PUT(task, MS_HOST_SELF(), PORT_START_JOB) != MSG_OK) {
        xbt_error("Error while sending a job start notification to the job simulator");
    }

    /* Notify the submitter */
    auto_t task;
    task = jd->id;
    task = MS_TASK_CREATE("job_started", jd->id, (void*)jd);
    if (MS_TASK_PUT(task, jd->submitter, jd->channel->started) != MSG_OK) {
        xbt_error("Error while sending a job started notification to the submittor");
    }

    /* Notify the receiver */
    auto_t task;
    task = jd->id;
    task = MS_TASK_CREATE("job_started", jd->id, (void*)jd);
    if (MS_TASK_PUT(task, jd->submitter, jd->channel->queue_size) != MSG_OK) {
        xbt_error("Error while sending a job started notification to the receiver");
    }
}

/* scheduler_init()
 * initializes the job descriptor queues and the number of free nodes */
void scheduler_init(job_scheduling_algorithm_t alg, scheduler_bookkeeping_t *bk) {
    /* Initialize the job descriptor queues and the number of free nodes */
    bk->queued = xbt_if_free();
    bk->running = xbt_if_free();

    /* Alg-specific initialization */
    case FCFS:
        scheduler_init_fcfs(bk);
        break;
    case RR:
        scheduler_init_rr(bk);
        break;
    case SPT:
        scheduler_init_spt(bk);
        break;
    case EDF:
        scheduler_init_edf(bk);
        break;
    case HRRN:
        scheduler_init_hrrn(bk);
        break;
    case PRIORITY:
        scheduler_init_priority(bk);
        break;
    default:
        xbt_error("Unknown scheduling algorithm");
    }
}

```

```

if (*num_targetSchedulers > 1)
    pendingJob->filed_across++;
free(prunedSchedulers);
return;
}

/* find_RR_targetSchedulers()
 */
void find_RR_targetSchedulers(int x, scheduler_info_t preferred,
                             scheduler_info_t *Schedulers, int numSchedulers,
                             int *numTargetSchedulers, scheduler_info_t **targetSchedulers)
{
    int i;
    int res;
    done;

    numTargetSchedulers = 0;
    for (i = 0; i < numSchedulers; i++) {
        if (i == 0) { /* first, pick the preferred scheduler */
            numTargetSchedulers++;
            if (Schedulers[i] == preferred) {
                res = i;
            }
        } else {
            int r = random_integer(0, numSchedulers - 1);
            if (Schedulers[r] == preferred) {
                res = r;
            }
        }
    }
    if (done) {
        for (i = 0; i < numTargetSchedulers; i++) {
            if (targetSchedulers[i] == Schedulers[res]) {
                if (done)
                    break;
            }
        }
        numTargetSchedulers++;
        if (Schedulers[done] == preferred) {
            numTargetSchedulers++;
        }
        if (numTargetSchedulers == numSchedulers) {
            targetSchedulers[0] = Schedulers[done];
            return;
        }
    }
}

/* find_SK_targetSchedulers()
 */
void find_SK_targetSchedulers(int x, scheduler_info_t preferred,
                             scheduler_info_t *Schedulers, int numSchedulers,
                             int *numTargetSchedulers, scheduler_info_t **targetSchedulers)
{
    scheduler_info_t *tosort;
    int i;
    ...
}

```

```

#include "simulator.h"
#include "stdlib.h"

XBT_LOG_NEW_DEFAULT_CATEGORY(receiver, "Logging for the receiver process");

/* Receiver function
 * @param #1 port
 * @param #2 dynar name
 * @param #3 dynar
 */
int receiver(int argc, char *argv[])
{
    int port = 0;
    //xbt_if_free();
    /* Process the first argument */
    if (sscanf(argv[1], "%d", &port) != 1) {
        xbt_error("invalid port %s for a receiver process", argv[1]);
    }

    /* get the dynar */
    if ((dynar = (dynar_t*)xbt_dict_get_or_null(receiver_dynar_dict, (const char*)argv[1]))) {
        xbt_error("Cannot find dynar %s for a receiver process", argv[1]);
    }

    /* init loop */
    while(1) {
        msghdr_t task = NULL;
        if (MS_TASK_GET(&task, port) != MSG_OK) {
            xbt_error("Error while receiving a task in a receiver process");
        }

        /* Process the task of a dynar (location in the dynar) */
        random_integer(0, MAX(dynar->length, dynar->size));
        dynar_insert_at(dynar, dynar->length, task);
    }
    return 1;
}

```

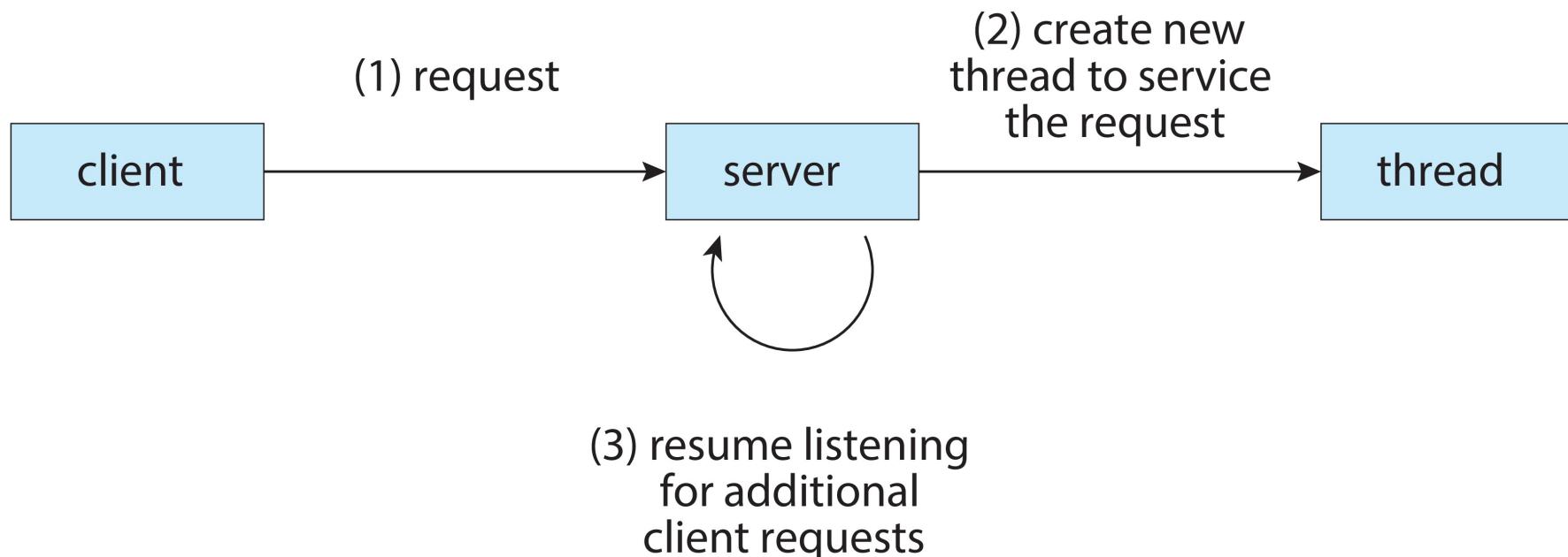
Advantages of Threads?

- Economy:
 - Creating a thread is cheap
 - › Much cheaper than creating a process
 - Code, data and heap are already in memory
 - › Context-switching between threads is cheap
 - › Much cheaper than between processes
 - No cache flush
 - Resource Sharing:
 - Threads naturally share memory
 - › With processes you have to use possibly complicated IPC (e.g., Shared Memory Segments)
 - › **IPC is not needed**
 - Having concurrent activities in the same address space is very powerful
 - › But fraught with danger

Advantages of Threads?

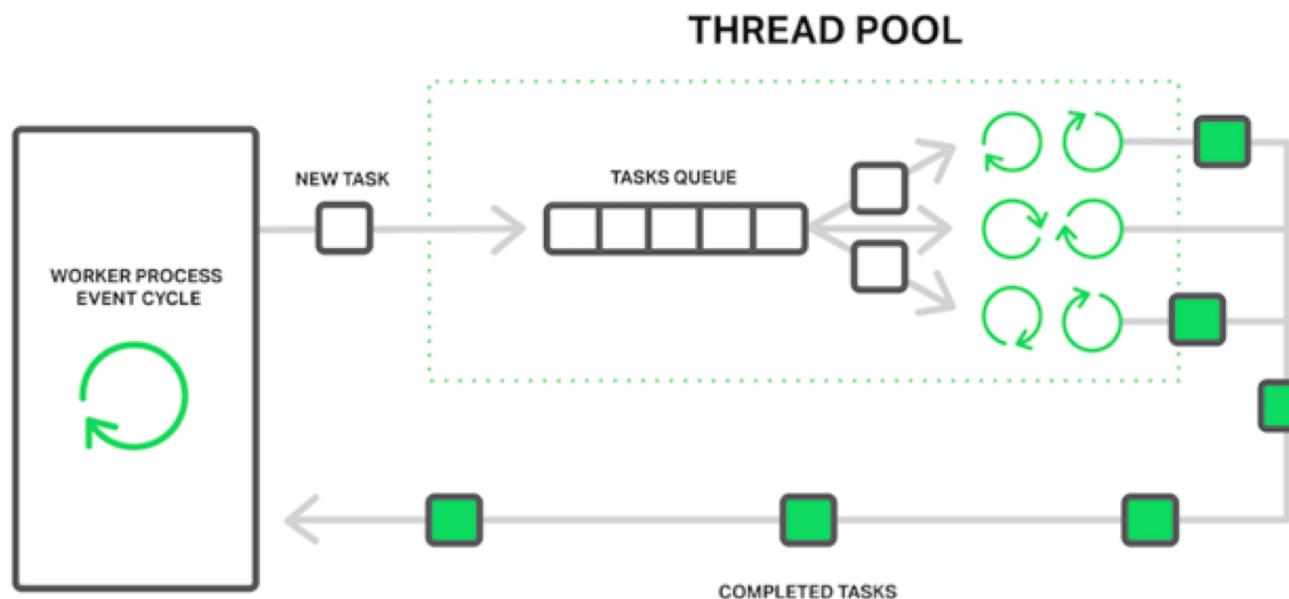
- Responsiveness

- A program that has concurrent activities is more responsive
 - › While one thread blocks waiting for some event, another can do something
 - › e.g. Spawn a thread to answer a client request in a client-server implementation
- This is true of processes as well, but with threads we have better sharing and economy



Advantages of Threads?

- Thread Pools in NGINX Boost Performance 9x
 - nginx : worker process -> thread pool



Advantages of Threads?

- Responsiveness
 - A program that has concurrent activities is more responsive
 - › While one thread blocks waiting for some event, another can do something
 - › e.g. Spawn a thread to answer a client request in a client-server implementation
 - This is true of processes as well, but with threads we have better sharing and economy
- Scalability
 - Running multiple “threads” at once uses the machine more effectively
 - › e.g., on a multi-core machine
 - This is true of processes as well, but with threads we have better sharing and economy

Drawbacks of Threads

- One drawback of thread-based concurrency compared to process-based concurrency: If one thread fails (e.g., a segfault), then the process fails
 - And therefore the whole program
- This leads to process-based concurrency
 - e.g., The Google Chrome Web browser
 - See <http://www.google.com/googlebooks/chrome/>
- Sort of a throwback to the pre-thread era
 - Threads have been available for 20+ years
 - Very trendy recently due to multi-core architectures

Drawbacks of Threads

- Threads may be more memory-constrained than processes
 - Due to OS limitation of the address space size of a single process
 - Not a problem any more on 64-bit architecture
- Threads do not benefit from memory protection
 - Concurrent programming with Threads is hard
 - › But so is it with Processes and Shared Memory Segments

Threads on My Machine?

- Let's run ps aux and look at several applications
 - ps -eLf
 - Chrome
 - Terminal
 - ...

Multi-Threading Challenges

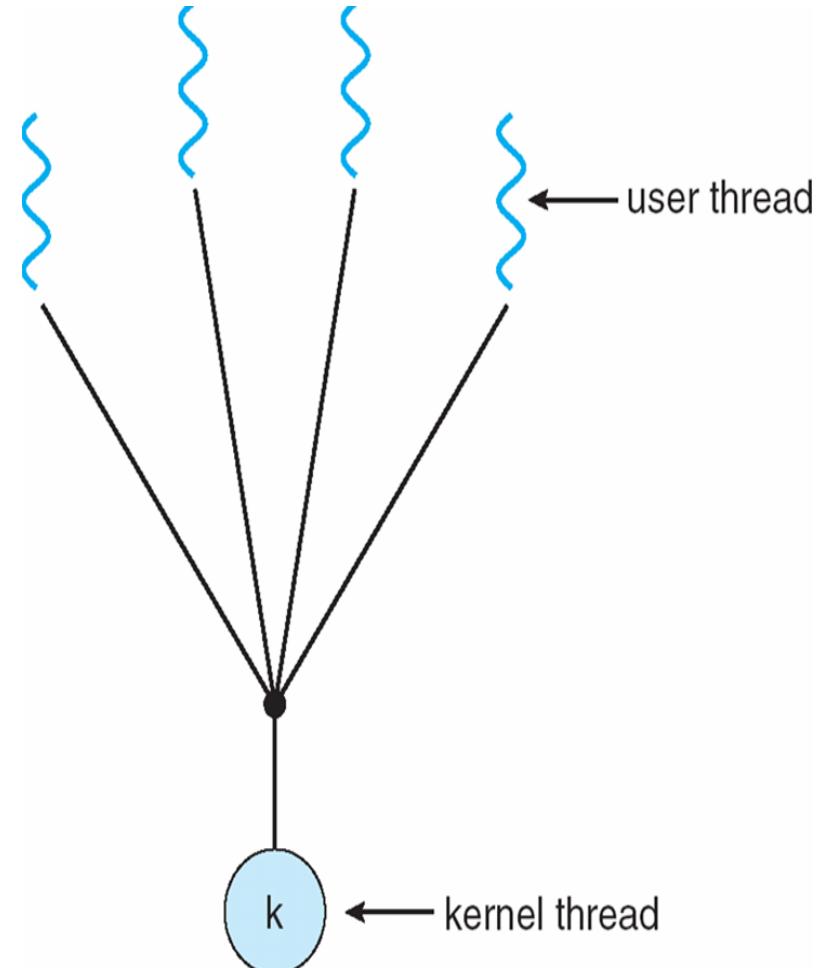
- Typical challenges of multi-threaded programming
 - Dividing activities among threads
 - Balancing load among threads
 - Split data among threads
 - Deal with data dependency and synchronization
 - Testing and Debugging

User Threads vs. Kernel Threads

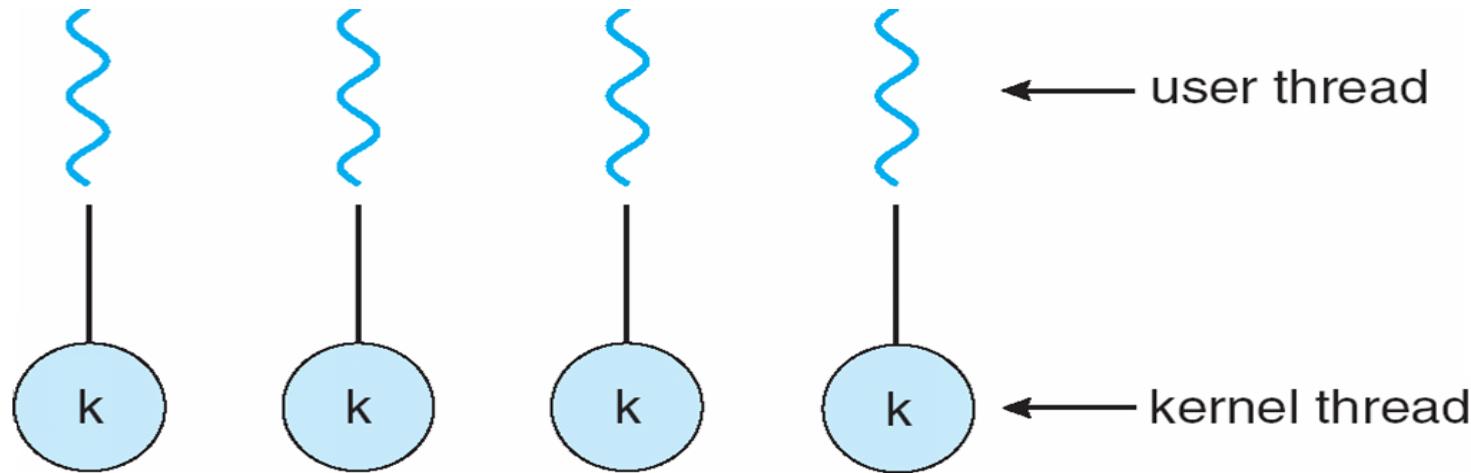
- Threads can be supported solely in User Space
 - Threads are managed by some user-level thread library (e.g., Java Green Threads)
- Threads can also be supported in Kernel Space
 - The kernel has data structure and functionality to deal with threads
 - Most modern OSes support kernel threads
 - ▶ In fact, Linux doesn't really make a difference between processes and threads (same data structure)

Many-to-One Model

- Advantage: multi-threading is efficient and low-overhead
 - No syscalls to the kernel
- Major Drawback #1: cannot take advantage of a multi-core architecture!
- Major Drawback #2: if one threads blocks, then all the others do!
- Examples (User-level Threads):
 - Java Green Threads
 - GNU Portable Threads



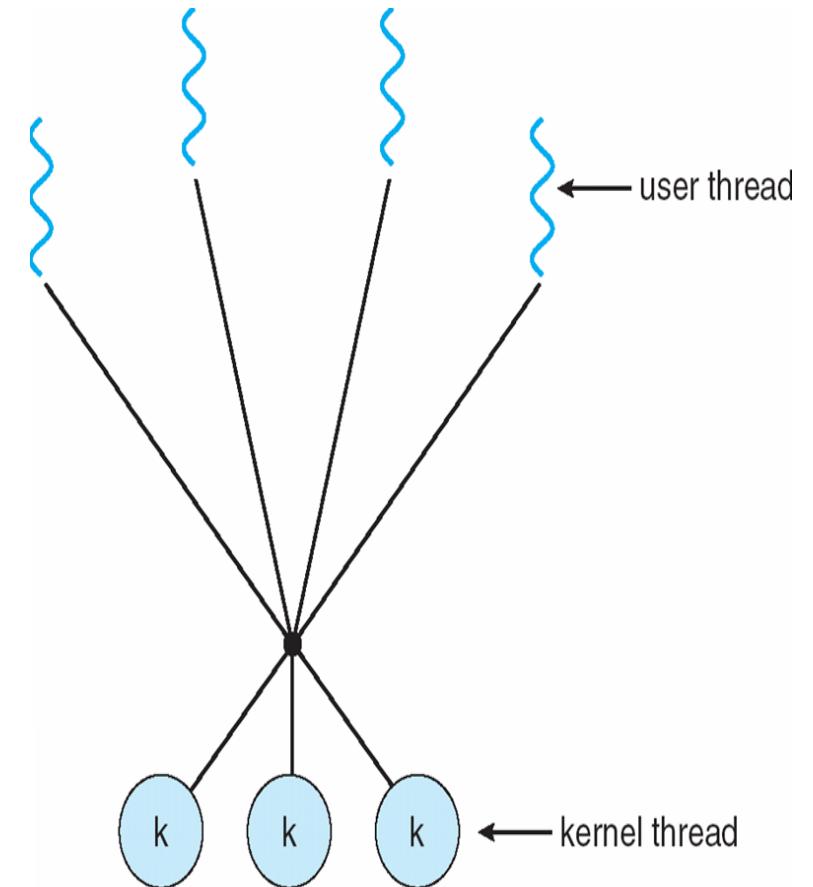
One-to-One Model



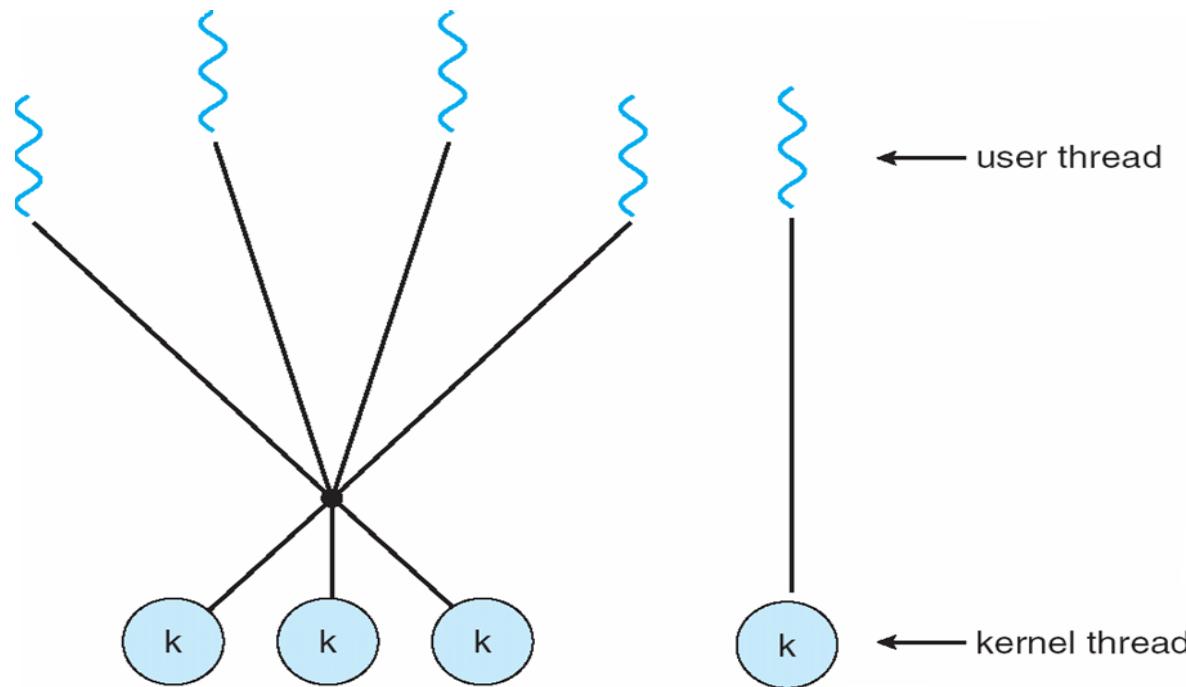
- Removes both drawbacks of the Many-to-One Model
- Creating a new threads requires work by the kernel
 - Not as fast as in the Many-to-One Model
- Example:
 - Linux
 - Windows
 - Solaris 9 and later

Many-to-Many Model

- A compromise
- If a user thread blocks, the kernel can create a new kernel threads to avoid blocking all user threads
- A new user thread doesn't necessarily require the creation of a new kernel thread
- True concurrency can be achieved on a multi-core machine
- Examples:
 - Solaris 9 and earlier
 - Win NT/2000 with the ThreadFiber package



Two-Level Model



- The user can say: “Bind this thread to its own kernel thread”
- Example:
 - IRIX, HP-UX, Tru64 UNIX
 - Solaris 8 and earlier

Thread Libraries

- Thread libraries provide users with ways to create threads in their own programs
 - In C/C++: Pthreads
 - › Implemented by the kernel
 - In C/C++: OpenMP
 - › A layer above Pthreads for convenient multithreading in “easy” cases
 - In Java: Java Threads
 - › Implemented by the JVM, which relies on threads implemented by the kernel

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)

Pthreads Example

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}
```

Pthreads Example

```
/* The thread will execute in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
```

Windows Multithreaded C Program

```
int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
```

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

#pragma omp parallel

Create as many threads as there are cores

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}

#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

Java Threads

- All memory-management headaches go away with Java Threads
 - In nice Java fashion
- Several programming languages have long provided constructs/abstractions for writing concurrent programs
 - Modula, Ada, etc.
- Java threads may be created by:
 - Extending Thread class

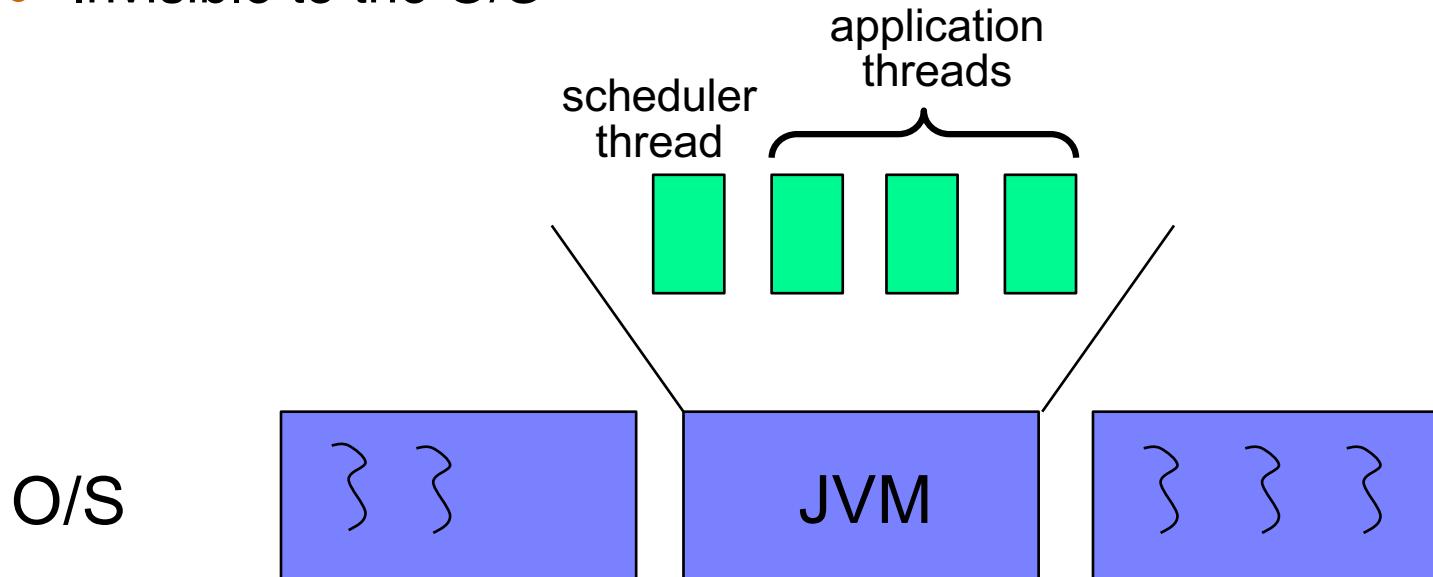
```
class MyThread extends Thread {  
    public void run() {  
        . . .  
    }  
    MyThread t = new MyThread();
```

- Implementing the Runnable interface

```
public interface Runnable  
{  
    public abstract void run();  
}
```

Thread Scheduling

- The JVM keeps track of threads, enacts the thread state transition diagram
- Question: who decides which runnable thread to run?
- Old versions of the JVM used **Green Threads**
 - User-level threads implemented by the JVM
 - Invisible to the O/S

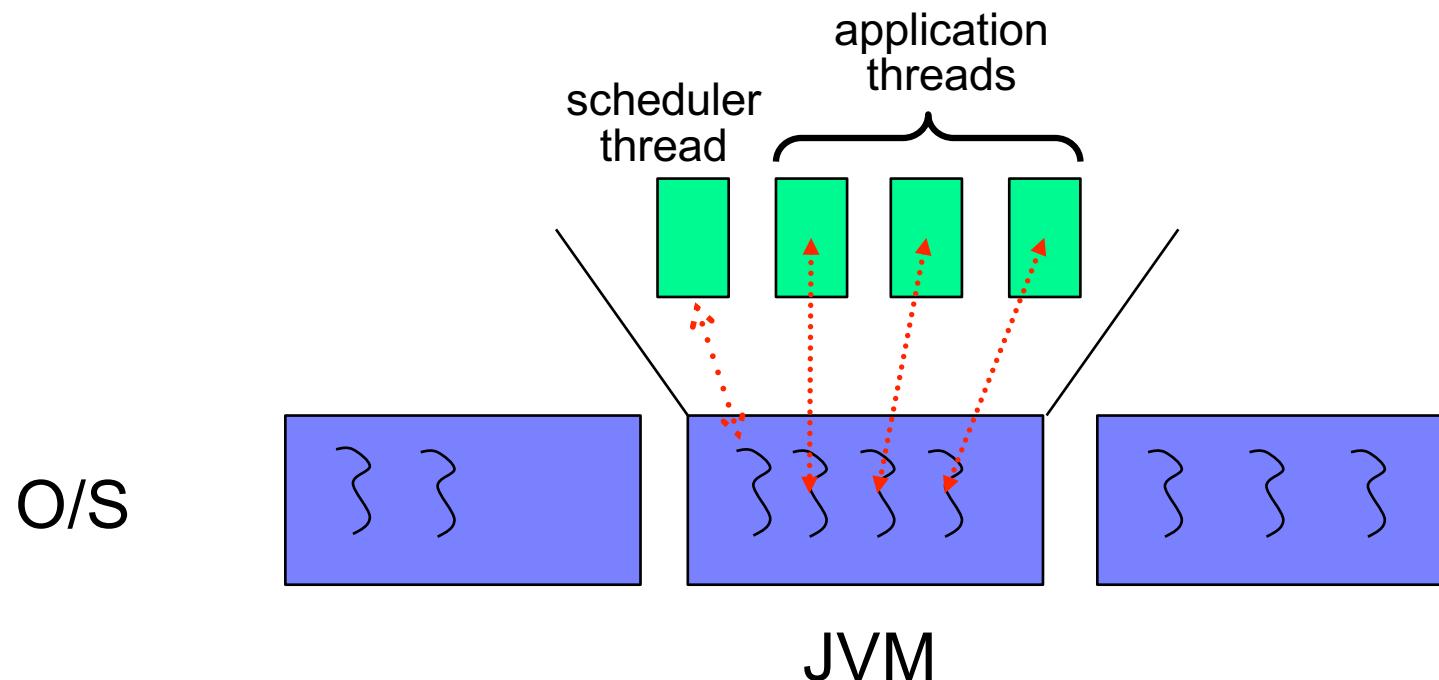


Beyond Green Threads

- Green threads have all the disadvantages of user-level threads (see earlier)
 - Most importantly: Cannot exploit multi-core, multi-processor architectures
- The JVM now provides **native threads**
 - Green threads are typically not available anymore
 - you can try to use “java -green” and see what your system says

Java Threads / Kernel Threads

- In modern JVMs, application threads are *mapped* to kernel threads



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- What happens when a thread calls fork()?
- Two possibilities:
 - A new process is created that has only one thread (the copy of the thread that called fork()), or
 - A new process is created with all threads of the original process (a copy of all the threads, including the one that called fork())
- Some OSes provide both options
 - In Linux the first option above is used
- If one calls exec() after fork(), all threads are “wiped out” anyway

Signals

- We've talked about signals for processes
 - Signal handlers are either default or user-specified
 - `signal()` and `kill()` are the system calls
- In a multi-threaded program, what happens?
- Multiple options
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals

Signals

- Most UNIX versions: a thread can say which signals it accepts and which signals it doesn't accept
- On Linux: dealing with threads and signals is tricky but well understood with many tutorials on the matter and man pages
 - `man pthread_sigmask`
 - `man sigemptyset`
 - `man sigaction`

Safe Thread Cancellation

- One potentially useful feature would be for a thread to simply terminate another thread
- Two possible approaches:
 - **Asynchronous** cancellation
 - One thread terminates another immediately
 - **Deferred** cancellation
 - A thread periodically checks whether it should terminate

Safe Thread Cancellation

- Two possible approaches:
 - **Asynchronous** cancellation
 - One thread terminates another immediately
 - **Deferred** cancellation
 - A thread periodically checks whether it should terminate
- Pthread code to create and cancel a thread:

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
.  
. . .  
  
/* cancel the thread */  
pthread_cancel(tid);  
  
/* wait for the thread to terminate */  
pthread_join(tid,NULL);
```

Safe Thread Cancellation

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
 - Cancellation only occurs when thread reaches **cancellation point**
 - I.e. `pthread_testcancel()`
 - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Safe Thread Cancellation

- The problem with asynchronous cancellation:
 - may lead to an inconsistent state or to a synchronization problem if the thread was in the middle of “something important”
 - Absolutely terrible bugs lurking in the shadows
- The problem with deferred cancellation: the code is cumbersome due to multiple cancellation points
 - should I die? should I die? should I die?
- In Java, the `Thread.stop()` method is deprecated, and so cancellation has to be deferred

Operating System Examples

- Windows Threads
- Linux Threads

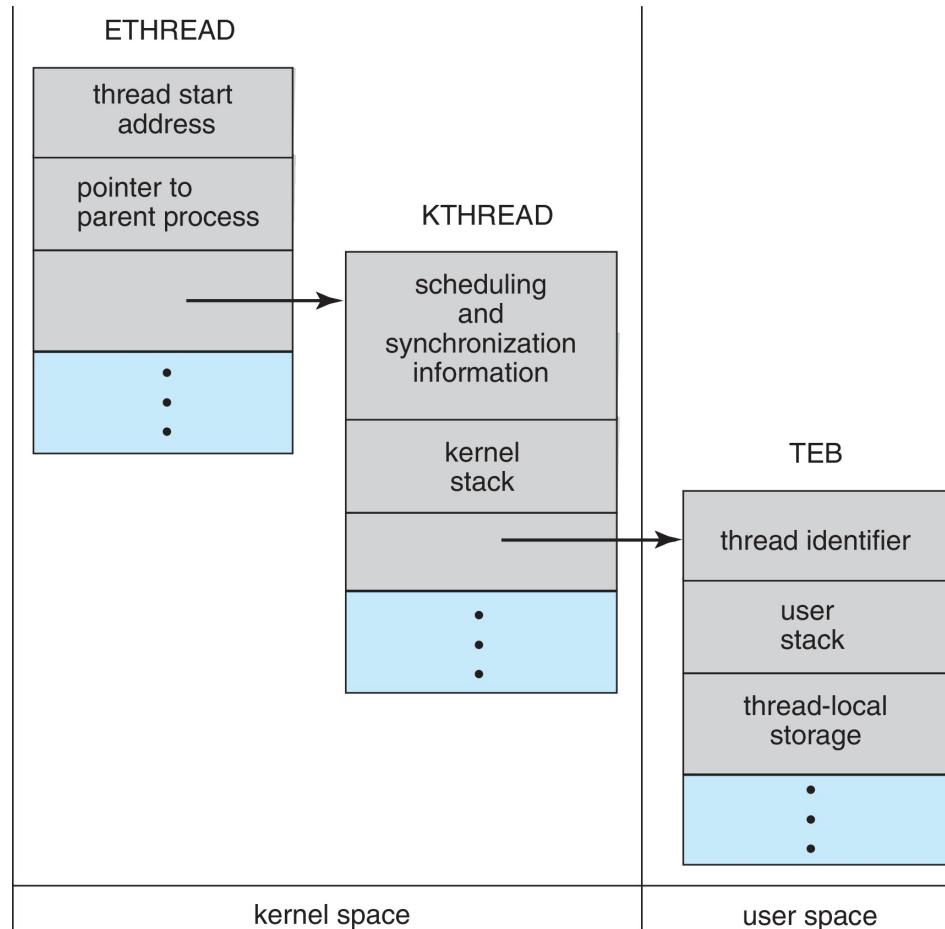
Windows Threads

- Windows API – primary API for Windows applications
- Implements the one-to-one mapping, kernel-level
- Each thread contains
 - A thread id
 - Register set representing state of processor
 - Separate user and kernel stacks for when thread runs in user mode or kernel mode
 - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:
 - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
 - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
 - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



Linux Threads

- Linux does not distinguish between processes and threads: they're called **tasks**
 - Kernel data structure: task_struct

```
591
592 struct task_struct {
593 #ifdef CONFIG_THREAD_INFO_IN_TASK
594     /*
595      * For reasons of header soup (see current_thread_info()), this
596      * must be the first element of task_struct.
597      */
598     struct thread_info           thread_info;
599 #endif
600     /* -1 unrunnable, 0 runnable, >0 stopped: */
601     volatile long                state;
602
603     /*
604      * This begins the randomizable portion of task_struct. Only
605      * scheduling-critical items should be added above here.
606      */
607     randomized_struct_fields_start
608
609     void                         *stack;
610     atomic_t                      usage;
611     /* Per task flags (PF_*), defined further below: */
612     unsigned int                  flags;
613     unsigned int                  ptrace;
614
```

Linux Threads

- In Linux, a thread is also called a light-weight process (LWP)
- The clone() syscall is used to create a task
 - Shares execution context with its parent
 - pthread library uses clone() to implement threads. Refer to
./nptl/sysdeps/pthread/createthread.c

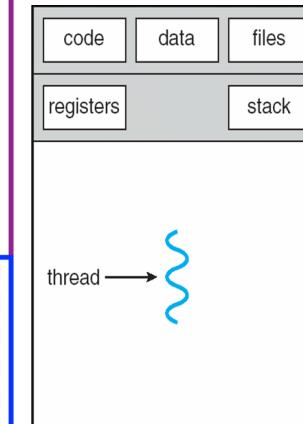
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Linux Threads

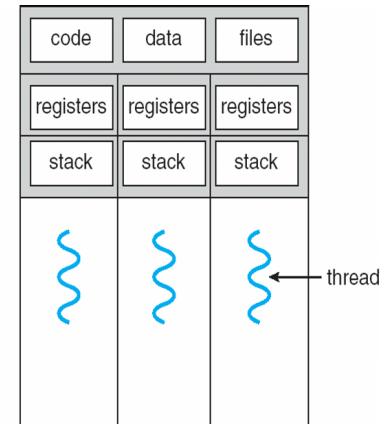
- Single-threaded process vs multi-threaded process

```
wenbo@wenbo-desktop:~/KERNEL/linux.git$ ps -elf
```

UID	PID	PPID	LWP	C	NLWP	STIME	TTY	TIME	CMD
root	1	0	1	0	1	3月 11 ?		00:00:19	/sbin/init splash
root	2	0	2	0	1	3月 11 ?		00:00:00	[kthreadd]
root	4	2	4	0	1	3月 11 ?		00:00:00	[kworker/0:0H]
root	6	2	6	0	1	3月 11 ?		00:00:00	[mm_percpu_wq]
root	7	2	7	0	1	3月 11 ?		00:00:00	[ksoftirqd/0]
root	8	2	8	0	1	3月 11 ?		00:00:31	[rcu_sched]
root	9	2	9	0	1	3月 .(1 ?		00:00:00	[rcu_bh]
root	10	2	10	0	1	3月 11 ?		00:00:00	[migration/0]
root	11	2	11	0	1	3月 11 ?		00:00:00	[watchdog/0]
root	704	1	704	0	1	3月 11 ?		00:00:00	/usr/sbin/cron -f
root	718	1	718	0	16	3月 11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	882	0	16	3月 11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	883	0	16	3月 11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	884	0	16	3月 11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	885	0	16	3月 11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	917	0	16	3月 11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	921	0	16	3月 11 ?		00:00:01	/usr/lib/snapd/snapd
root	718	1	922	0	16	3月 11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	923	0	16	3月 11 ?		00:00:01	/usr/lib/snapd/snapd
root	718	1	924	0	16	3月 11 ?		00:00:01	/usr/lib/snapd/snapd



single-threaded process



multithreaded process

Linux Threads

- Single-threaded process vs multi-threaded process

```
wenbo@wenbo-desktop:~/KERNEL/linux.git$ ps -elf
UID  PID  PPID  LWP  C NLWP STIME TTY          TIME CMD
root    1    0      1  0   1 3月11 ?        00:00:19 /sbin/init splash
root    2    0      2  0   1 3月11 ?        00:00:00 [kthreadd]
root    4    2      4  0   1 3月11 ?        00:00:00 [kworker/0:0H]
root    6    2      6  0   1 3月11 ?        00:00:00 [mm_percpu_wq]
root    7    2      7  0   1 3月11 ?        00:00:00 [ksoftirqd/0]
root    8    2      8  0   1 3月11 ?        00:00:31 [rcu_sched]
root    9    2      9  0   1 3月11 ?        00:00:00 [rcu_bh]
root   10    2     10  0   1 3月11 ?        00:00:00 [migration/0]
root   11    2     11  0   1 3月11 ?        00:00:00 [watchdog/0]

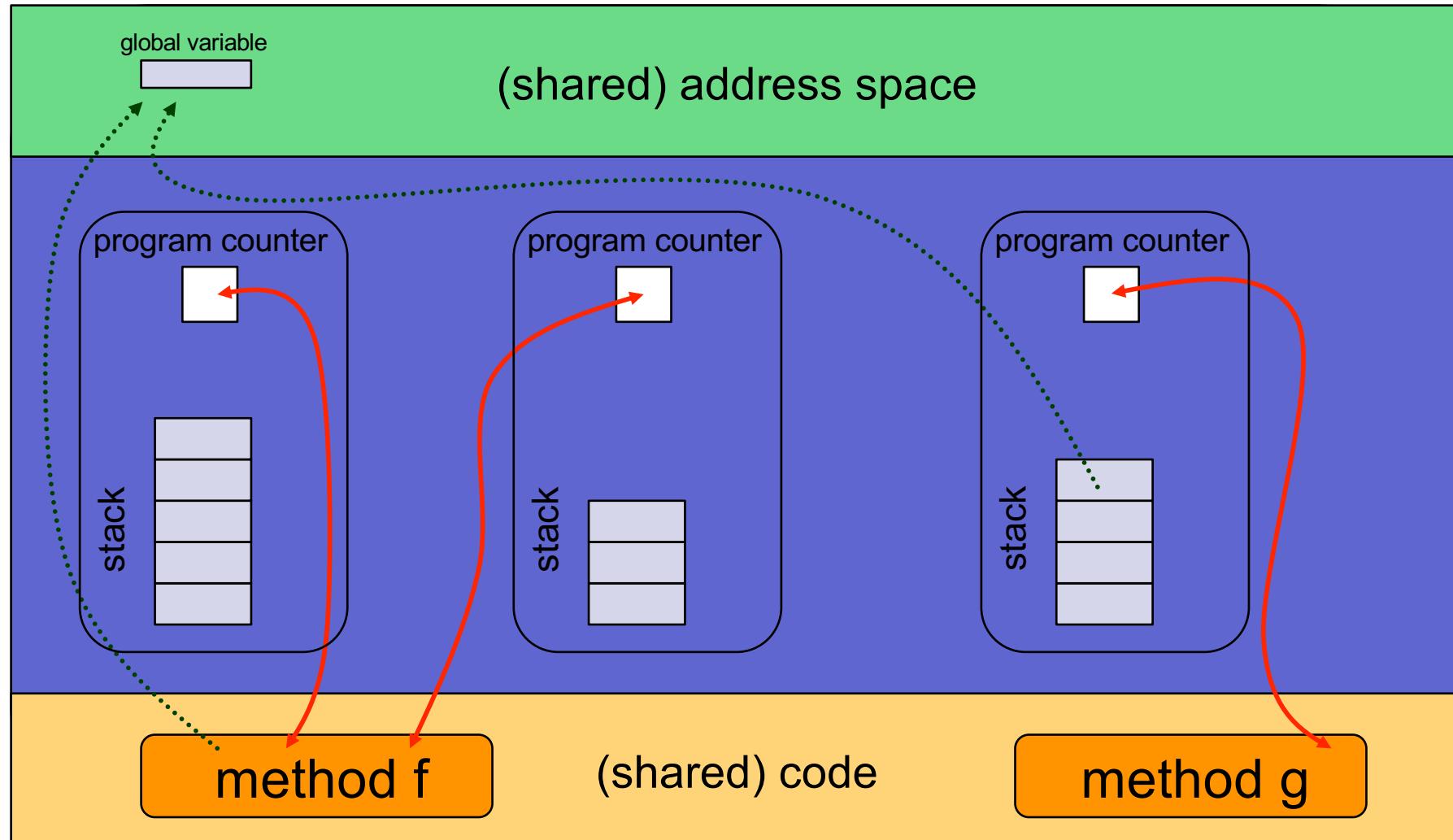
root   704    1    704  0   1 3月11 ?        00:00:00 /usr/sbin/cron -f
root   718    1    718  0   16 3月11 ?       00:00:00 /usr/lib/snapd/snapd
root   718    1    882  0   16 3月11 ?       00:00:00 /usr/lib/snapd/snapd
root   718    1    883  0   16 3月11 ?       00:00:00 /usr/lib/snapd/snapd
root   718    1    884  0   16 3月11 ?       00:00:00 /usr/lib/snapd/snapd
root   718    1    885  0   16 3月11 ?       00:00:00 /usr/lib/snapd/snapd
root   718    1    917  0   16 3月11 ?       00:00:00 /usr/lib/snapd/snapd
root   718    1    921  0   16 3月11 ?       00:00:01 /usr/lib/snapd/snapd
root   718    1    922  0   16 3月11 ?       00:00:00 /usr/lib/snapd/snapd
root   718    1    923  0   16 3月11 ?       00:00:01 /usr/lib/snapd/snapd
root   718    1    924  0   16 3月11 ?       00:00:01 /usr/lib/snapd/snapd
```

```
787  /* PID/PID hash table linkage. */
788  struct pid                         *thread_pid;
789  struct hlist_node                   pid_links[PIDTYPE];
790  struct list_head                   thread_group;
791  struct list_head                   thread_node;
792
793  struct completion                  *vfork_done;
794
795  /* CLONE_CHILD_SETTID: */
796  int __user                         *set_child_tid;
---
```

Linux Threads

- Linux does not distinguish between processes and threads: they're called **tasks**
 - Kernel data structure: task_struct
- A process is
 - either a **single** thread + an address space
 - ▶ PID is thread ID
 - or **multiple** threads + an address space
 - ▶ PID is the leading thread ID

Threads within Process



Threads with Process – What is shared

```
-- 29 static void traversal_thread_group(struct task_struct * tsk){  
30     struct task_struct * curr_thread = NULL;  
31     unsigned long tg_offset = offsetof(struct task_struct, thread_group);  
32  
33     curr_thread = (struct task_struct *) (((unsigned long)tsk->thread_group.next) - tg_offset);  
34     while (curr_thread != tsk){  
35         printk("\t\THREAD TSK=%llx\tPID=%d\tSTACK=%llx \tCOMM=%s\tMM=%llx\tACTIVE_MM=%llx\n",  
36             (u64)curr_thread, curr_thread->pid, (u64)curr_thread->stack,  
37             curr_thread->comm, (u64)curr_thread->mm, (u64)curr_thread->active_mm);  
38         curr_thread = (struct task_struct *) (((unsigned long)curr_thread->thread_group.next) - tg_offset);  
39     }  
40 }  
41  
42 static void traversal_process(void) {  
43     struct task_struct * tsk = NULL;  
44  
45     traversal_thread_group(&init_task);  
46     for_each_process(tsk){  
47         printk("PROCESS\THREAD TSK=%llx\tPID=%d\tSTACK=%llx \tCOMM=%s\tMM=%llx\tACTIVE_MM=%llx\n",  
48             (u64)tsk, tsk->pid, (u64)tsk->stack, tsk->comm,  
49             (u64)tsk->mm, (u64)tsk->active_mm);  
50         traversal_thread_group(tsk);  
51     }  
52 }
```

PROCESS	THREAD TSK=fffff8c4c4bf3c5c0	PID=718 STACK=fffff985c82268000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c46d52e80	PID=882 STACK=fffff985c82390000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c46d545c0	PID=883 STACK=fffff985c822e8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c491b45c0	PID=884 STACK=fffff985c8218c000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c4beb1740	PID=885 STACK=fffff985c821ec000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c4ae1ae80	PID=917 STACK=fffff985c823c8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c4b562e80	PID=921 STACK=fffff985c82418000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c48340000	PID=922 STACK=fffff985c823b0000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c472bae80	PID=923 STACK=fffff985c821f4000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c4b5945c0	PID=924 STACK=fffff985c81fa8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c46775d00	PID=925 STACK=fffff985c822a8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c4b692e80	PID=973 STACK=fffff985c82438000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c4b78ae80	PID=974 STACK=fffff985c823c0000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD TSK=fffff8c4c46e1dd00	PID=975 STACK=fffff985c824b8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840

Threads within Process – What is shared

PROCESS	THREAD	TSK=fffff8c4c4bf3c5c0	PID=718	STACK=fffff985c82268000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	task_struct	pid	stack	comm		mm_struct	
	THREAD	TSK=fffff8c4c46d52e80	PID=882	STACK=fffff985c82390000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c46d545c0	PID=883	STACK=fffff985c822e8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c491b45c0	PID=884	STACK=fffff985c8218c000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c4beb1740	PID=885	STACK=fffff985c821ec000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c4ae1ae80	PID=917	STACK=fffff985c823c8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c4b562e80	PID=921	STACK=fffff985c82418000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c48340000	PID=922	STACK=fffff985c823b0000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c472bae80	PID=923	STACK=fffff985c821f4000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c4b5945c0	PID=924	STACK=fffff985c81fa8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c46775d00	PID=925	STACK=fffff985c822a8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c4b692e80	PID=973	STACK=fffff985c82438000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c4b78ae80	PID=974	STACK=fffff985c823c0000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840
	THREAD	TSK=fffff8c4c46e1dd00	PID=975	STACK=fffff985c824b8000	COMM=snapd	MM=fffff8c4c46400840	ACTIVE_MM=fffff8c4c46400840

Not Shared

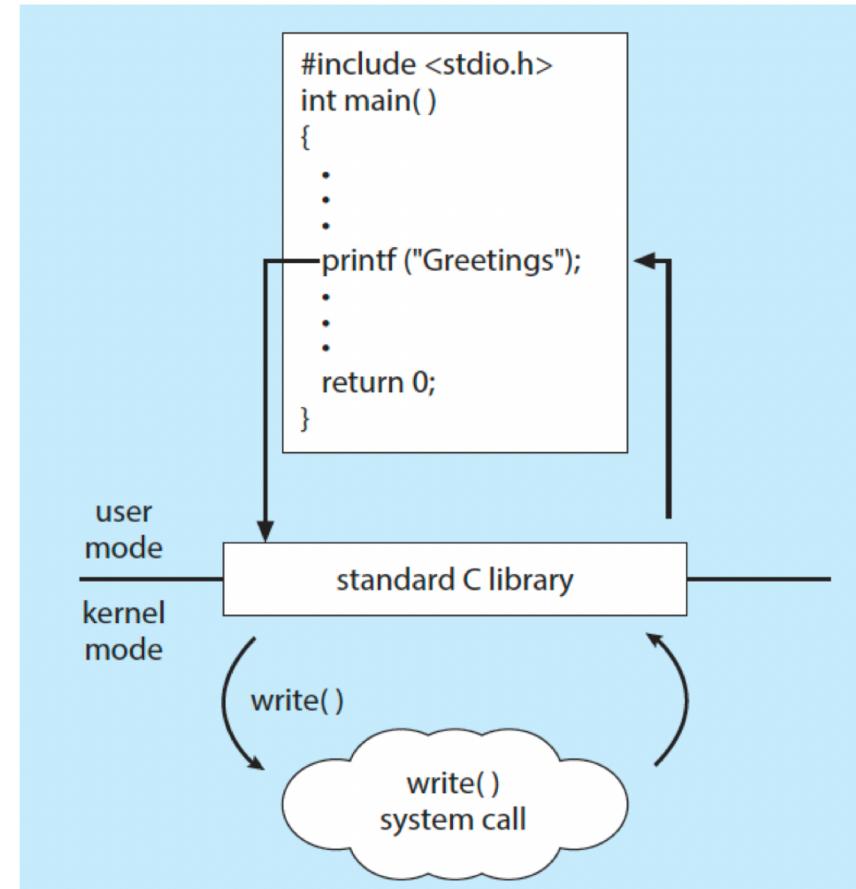
Shared

User thread to kernel thread mapping

- One task
 - One task struct – PCB
 - Can be executed in user space
 - › User code, user space stack
 - Can be executed in kernel space
 - › Such as calls a system call
 - › Execution flow traps to kernel
 - › Execute kernel code, use kernel space stack

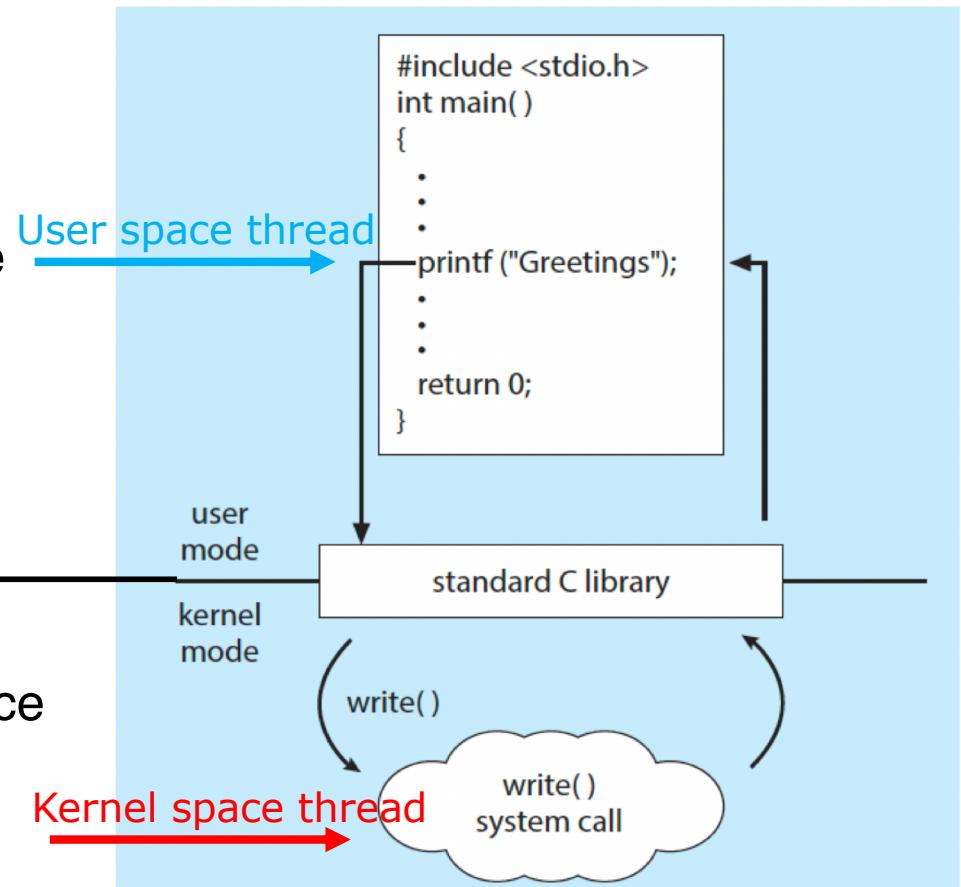
User thread to kernel thread mapping

- One task
 - One task struct – PCB
 - Execute in one thread
 - Can be executed in user space
 - › User code, user space stack
 - Can be executed in kernel space
 - › Such as calls a system call
 - › Execution flow traps to kernel
 - › Execute kernel code, use kernel space stack



User thread to kernel thread mapping

- One task
 - One task struct – PCB
 - Execute in one thread
 - Can be executed in user space
 - › User code, user space stack
 - Can be executed in kernel space
 - › Such as calls a system call
 - › Execution flow traps to kernel
 - › Execute kernel code, use kernel space stack



User thread to kernel thread mapping

- One task in Linux
 - Same task_struct (PCB) means same thread
 - Also viewed as 1:1 mapping
 - One user thread maps to one kernel thread
 - But actually, they are the same thread
 - Can be executed in user space
 - User code, user space stack
 - Can be executed in kernel space
 - Kernel code, kernel space stack
- Kernel thread also uses to represent threads that has no user space part
 - Such as kernel wants to flush dirty buffer to disk
 - It creates a thread, running only in kernel mode

Takeaway

- Thread is the basic execution unit
 - Has its own registers, pc, stack
- Thread vs Process
 - What is shared and what is not
- Pros and cons of thread