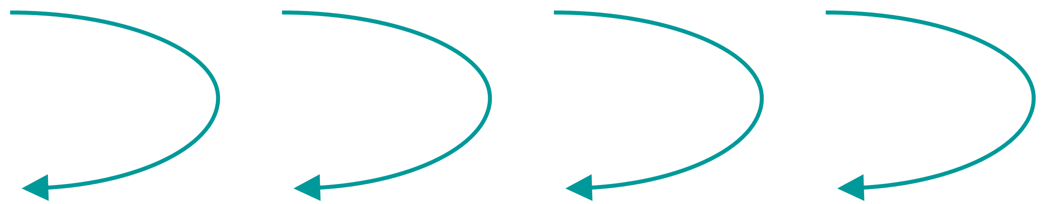
CSC 360

Operating System Structures:
From Processes to Threads

Threads

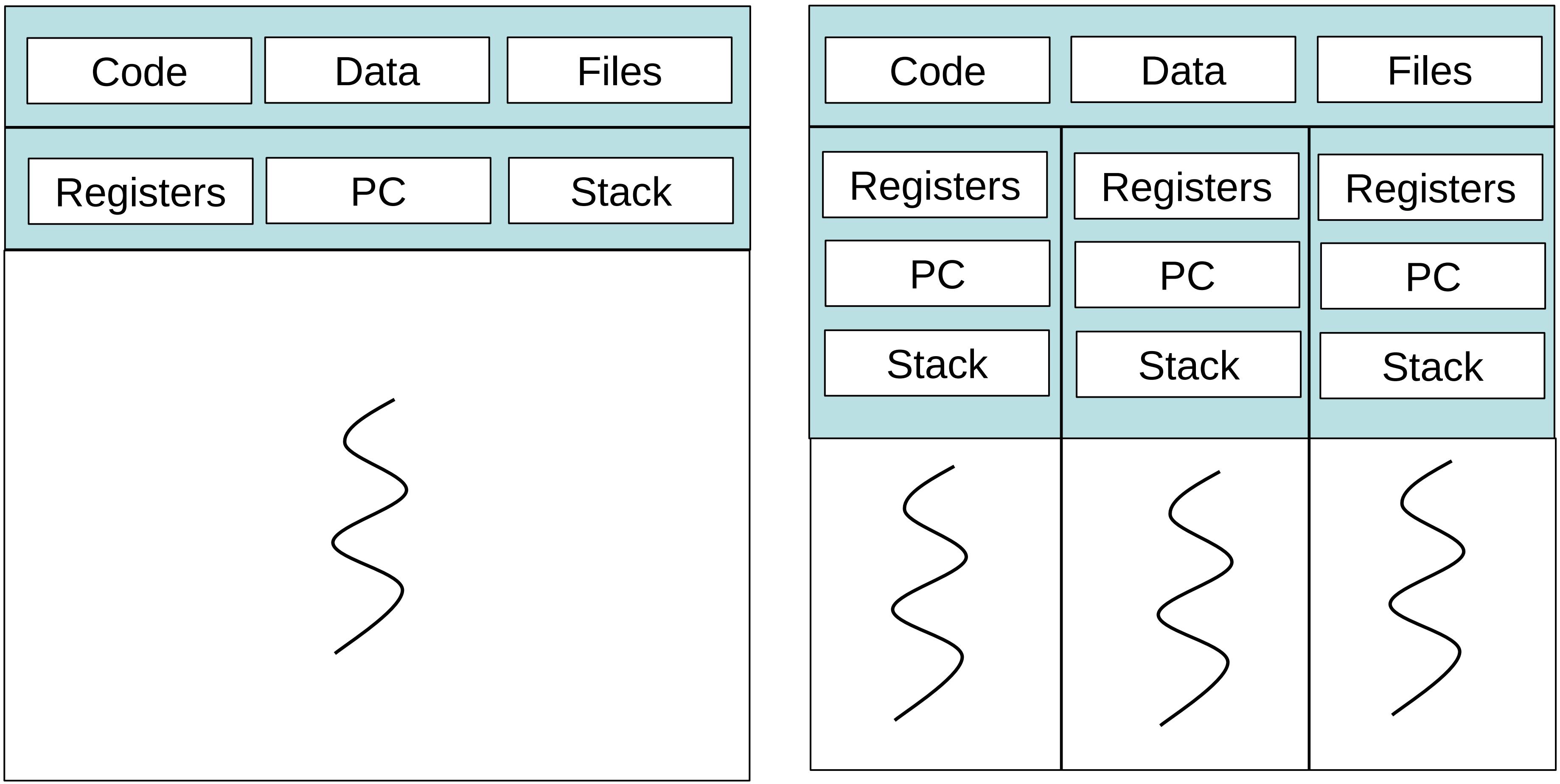
- What is shareable with processes before threads?
- Motivation for threads

Why Threads?



- **Thread**
  - short for **thread of control**
  - the sequence of executed instructions in a program, ***representing a single path of execution***
- Many algorithms are **easier** to write (and maintain) with threads
- Some algorithms run **faster** in a threaded implementation.

Single-thread vs. multithreaded process



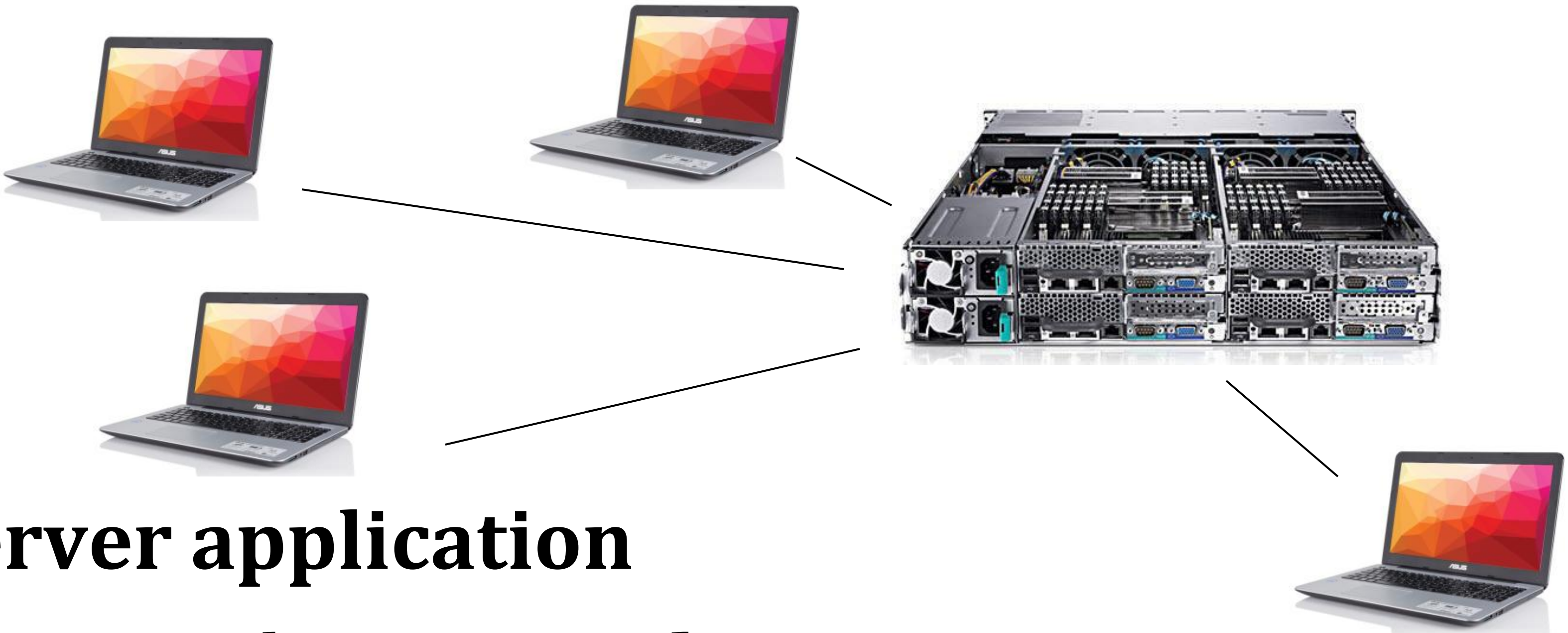
Process vs. Thread

- Processes have **separate address spaces**
  - Enforced by OS
  - *Sharing memory amongst processes requires OS intervention* (i.e., somewhat complex and expensive in time/CPU cycles)
- Threads exist **within** a process
  - A process may have **one or many threads**
  - Each of these threads **shares the same address space**
  - That is, *these threads by default share memory of their host process...*
  - **... and this sharing does not require the involvement of the kernel!**
  - For threads this semester, we'll focus on POSIX threads (`pthreads`).

Benefits of Using Threads

- Using threads is a natural way to achieve concurrency in programming
- Threads enable us to take advantage of multiprocessor systems,...
  - but they are just as useful on uniprocessor systems
- In many cases, a multithreaded solution is easier to develop, understand, and debug than its single-threaded counterpart
- By allowing multiple tasks to run concurrently, threads improve program responsiveness and resource utilization

Motivation...

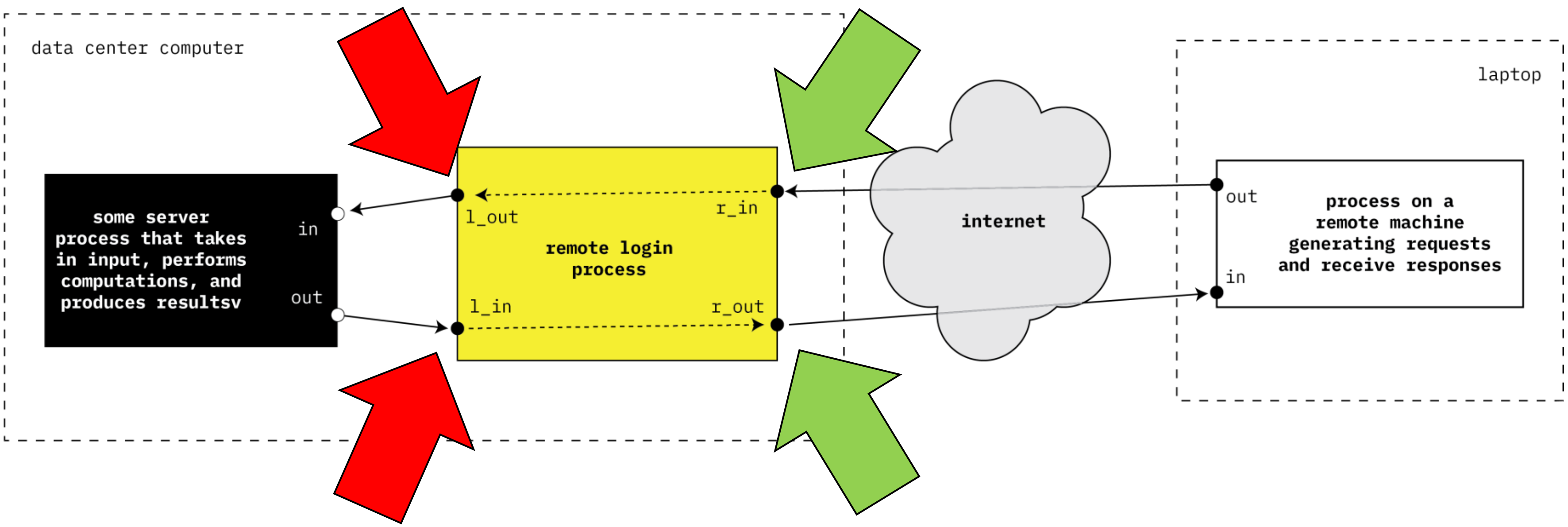


- **Server application**
  - Example: **remote login server**
  - Other examples: web server; database server

Describing action from the point of view of the **remote login process.** The names of the ports are are from the point of view of this process.

(B) Requests intended for server processing **are sent out here**.

(A) Incoming messages from remote machine for server processing **arrive here**



(C) Results produced by server processing **are sent out here**.

(D) Outgoing messages containing the results from server processing **are sent out here.**

Purpose of a remote login server

- This server (or *daemon*) acts as the go-between:
  - the computer outside the server machine, and ...
  - the processes within the server machine
- Remote-login server **receives** incoming messages from remote computer, and **sends** them to a process within the server computer
- Remote-login server **receives** responses from process within the server computer, and **sends** them to the remote computer
- Also:
  - The **`r_in`** and **`l_out`** ports only have messages when the remote computer or server machine have something to send (i.e., remote login server may need to wait for message on those incoming message ports)
  - The **`l_in`** and **`r_out`** ports have outgoing message queues that might be full (i.e., remote login server may need to wait for outgoing message ports to have room for new messages)

One implementation approach (bad)

- Remote login server (daemon) is organized around a **polling loop**
  - For each loop iteration, code examine each of the four ports
  - If a port requires servicing, then something is done
  - Otherwise return to top of loop
- **Observation**: Given the speed of CPUs relative to the frequency of server events, most loop iterations will do no processing (i.e., nothing to do) ....
  - Therefore, CPU cycles are wasted
- Polling is often used as a first approach to solving such coding problems...
  - ... but the wasted CPU cycles can become a problem as more and more such daemons exists in a computer

Better implementation approach

- We still use a loop, but its contents are very different (***but still without threads…***)
- Code in loop uses system calls such that the OS:
  - **Suspends** the remote-login-daemon's process when there is no work (i.e., no messages to relay from port to port)
  - **Reawakens** the remote-login-daemon's process when there is a message to relay to a port
- **This is far, far better than a straight polling loop**
  - When the daemon has "nothing to do", the process does not consume CPU cycles
  - When the daemon does have something to do, it is normally
    - (1) moving messages from **`r_in`** to **`l_out`**, or
    - (2) moving messages from **`l_in`** to **`r_out`**
- One issue, however, with respect to the design of our code:
  - If our process possesses a single thread, then the code of the loop **must be carefully written to interleave actions (1) and (2)**

Life Without Threads

Is the purpose of the code immediately apparent?

```
logind(int r_in, int r_out, int l_in, int l_out) {
  fd_set in = 0, out;
  int  want_l_write = 0, want_r_write = 0;
  int want_l_read = 1, want_r_read = 1;
  int eof = 0, tsize, fsize, wret;
  char fbuf[BSIZE], tbuf[BSIZE];

  fcntl(r_in, F_SETFL, O_NONBLOCK);
  fcntl(r_out, F_SETFL, O_NONBLOCK);
  fcntl(l_in, F_SETFL, O_NONBLOCK);
  fcntl(l_out, F_SETFL, O_NONBLOCK);

  while(!eof) {
    FD_ZERO(&in);
    FD_ZERO(&out);
    if (want_l_read) FD_SET(l_in, &in);
    if (want_r_read) FD_SET(r_in, &in);
    if (want_l_write) FD_SET(l_out, &out);
    if (want_r_write) FD_SET(r_out, &out);

    select(MAXFD, &in, &out, 0, 0);

    if (FD_ISSET(l_in, &in)) {
      if ((tsize = read(l_in, tbuf, BSIZE)) > 0)
      {
        want_l_read = 0;
        want_r_write = 1;
      } else {
```

Life Without Threads

Is the purpose of the each "if" statement immediately apparent?

```
      eof = 1;
    }
    if (FD_ISSET(r_in, &in)) {
        if ((fsize = read(r_in, fbuf, BSIZE)) > 0)
        {
          want_r_read = 0;
          want_l_write = 1;
        } else
          eof = 1;
      }
    if (FD_ISSET(l_out, &out)) {
      if ((wret = write(l_out, fbuf, fsize)) == fsize)
      {
        want_r_read = 1;
        want_l_write = 0;
      } else if (wret >= 0)
        tsize -= wret;
      else
        eof = 1;
    }
    if (FD_ISSET(r_out, &out)) {
      if ((wret = write(r_out, tbuf, tsize)) == tsize)
      {
        want_l_read = 1;
        want_r_write = 0;
      } else if (wret >= 0)
        tsize -= wret;
      else
        eof = 1;
    }
  }
}
```

```
logind(int r_in, int r_out, int l_in, int l_out) {
  fd_set in = 0, out;
  int  want_l_write = 0, want_r_write = 0;
  int want_l_read = 1, want_r_read = 1;
  int eof = 0, tsize, fsize, wret;
  char fbuf[BSIZE], tbuf[BSIZE];

  fcntl(r_in, F_SETFL, O_NONBLOCK);
  fcntl(r_out, F_SETFL, O_NONBLOCK);
  fcntl(l_in, F_SETFL, O_NONBLOCK);
  fcntl(l_out, F_SETFL, O_NONBLOCK);

  while(!eof) {
    FD_ZERO(&in);
    FD_ZERO(&out);
    if (want_l_read) FD_SET(l_in, &in);
    if (want_r_read) FD_SET(r_in, &in);
    if (want_l_write) FD_SET(l_out, &out);
    if (want_r_write) FD_SET(r_out, &out);

    select(MAXFD, &in, &out, 0, 0);

    if (FD_ISSET(l_in, &in)) {
      if ((tsize = read(l_in, tbuf, BSIZE)) > 0)
      {
        want_l_read = 0;
        want_r_write = 1;
      } else {
```

l_in: **Local input** l_out: **Local output**
r_in: **Remote input** r_out: **Remote output**

Initially, we want to **read** from the local terminal & remote machine;
We do not write anything until we have data

Set Non-Blocking Mode; This ensures that read() and write() do not block execution

Monitoring I/O using select() system call; **continuously monitors the file descriptors** until at least one file descriptor is ready for reading or writing.

Read Data from Local Terminal

If l_in has data, we read it into tbuf;
1. We pause further reading (want_l_read = 0)
2. and prepare to write this data to the remote machine (want_r_write = 1)

```
      eof = 1;
    }
    if (FD_ISSET(r_in, &in)) {
        if ((fsize = read(r_in, fbuf, BSIZE)) > 0)
        {
          want_r_read = 0;
          want_l_write = 1;
        } else
          eof = 1;
      }
    if (FD_ISSET(l_out, &out)) {
      if ((wret = write(l_out, fbuf, fsize)) == fsize)
      {
        want_r_read = 1;
        want_l_write = 0;
      } else if (wret >= 0)
        tsize -= wret;
      else
        eof = 1;
    }
    if (FD_ISSET(r_out, &out)) {
      if ((wret = write(r_out, tbuf, tsize)) == tsize)
      {
        want_l_read = 1;
        want_r_write = 0;
      } else if (wret >= 0)
        tsize -= wret;
      else
        eof = 1;
    }
  }
}
```

Read Data from Remote Machine

If r_in has data, we read it into fbuf

1. We pause further reading from the remote (want_r_read = 0) and
2. prepare to write this data to the local terminal (want_l_write = 1).

Write Data to Local Terminal

If l_out is ready for writing, **we send the data received from the remote machine.**

Once the full message is sent, **we resume reading from the remote**.

Write Data to Remote Machine

If r_out is ready for writing, we send the data received from the local terminal.

Once the full message is sent, we resume reading from the local terminal.

Better implementation approach

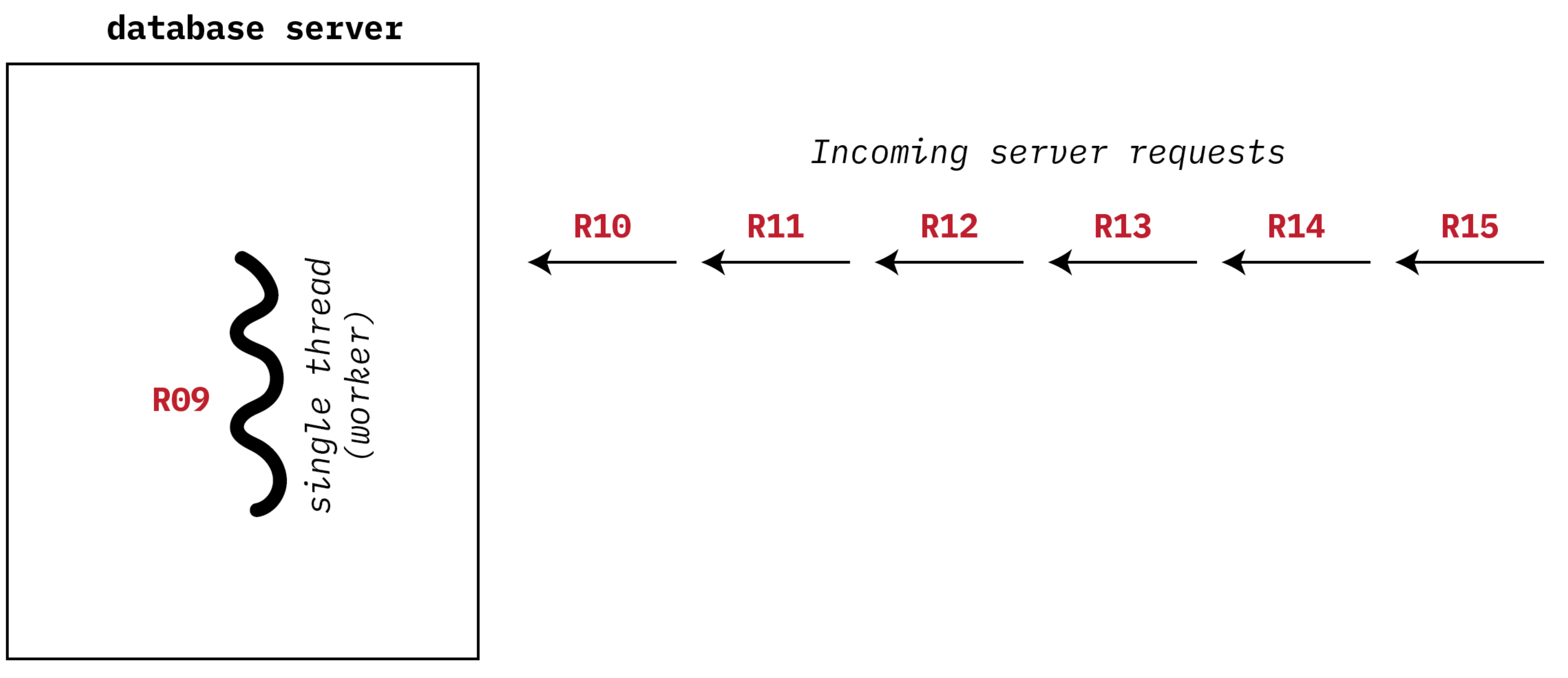
- **Multithreading**:
  - Our process will now consist of two separate threads
  - Both threads will share the processes global (i.e., program scope) variables/data...
  - ... but each thread will have its own local variables.
- **The important idea:**
  - One thread handles the message transfer from **`r_in`** to **`l_out`**...
  - ... while the other thread handles the message transfer from **`l_in`** to **`r_out`**
- (We will see later how to start these threads, but for now let us look at one possible version of their code)

Life With Threads

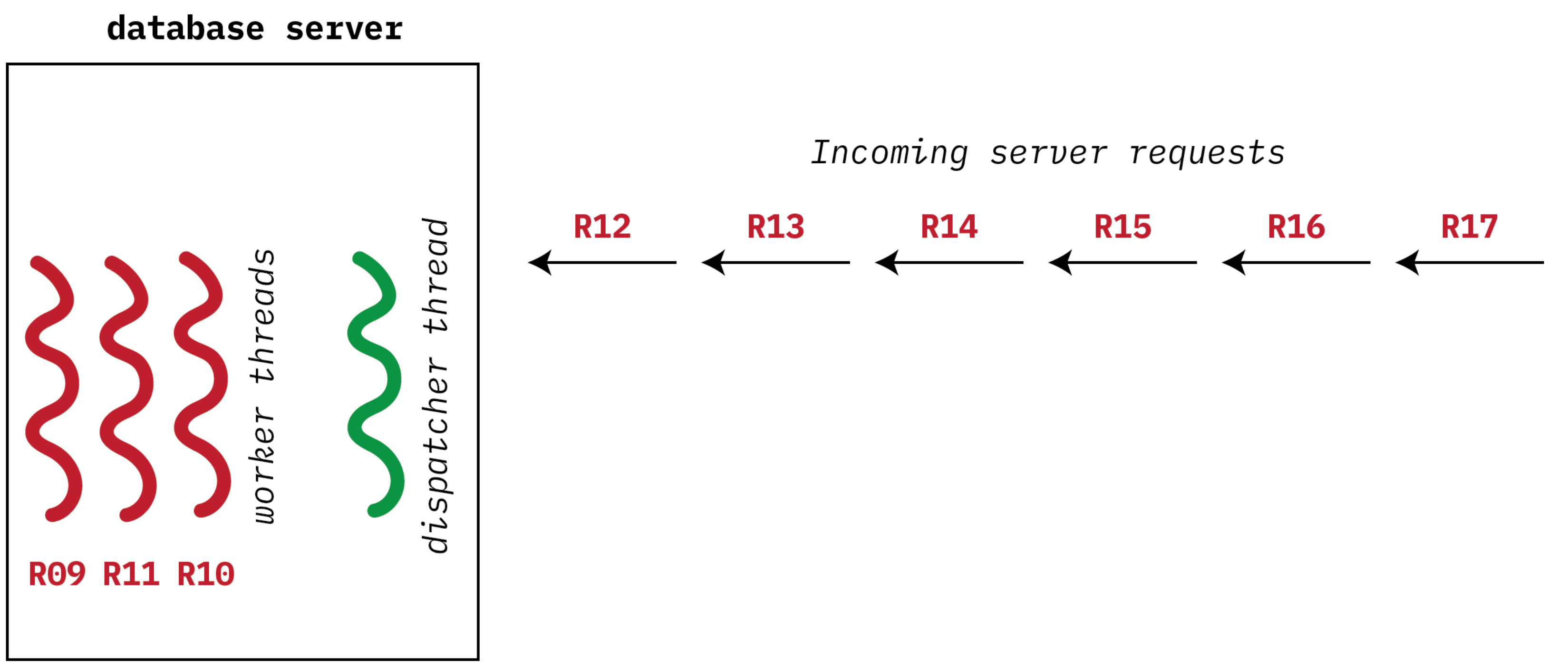
```
incoming(int r_in, int l_out) {
  int eof = 0;
  char buf[BSIZE];
  int size;

  while (!eof) {
    size = read(r_in, buf, BSIZE);
    if (size <= 0)
      eof = 1;
    if (write(l_out, buf, size) <= 0)
      eof = 1;
  }
}
```

```
outgoing(int l_in, int r_out) {
    int eof = 0;
    char buf[BSIZE];
    int size;

    while (!eof) {
        size = read(l_in, buf, BSIZE);
        if (size <= 0)
            eof = 1;
        if (write(r_out, buf, size) <= 0)
            eof = 1;
    }
}
```

Single-Threaded Database Server



Multithreaded Database Server



Benefits of threads

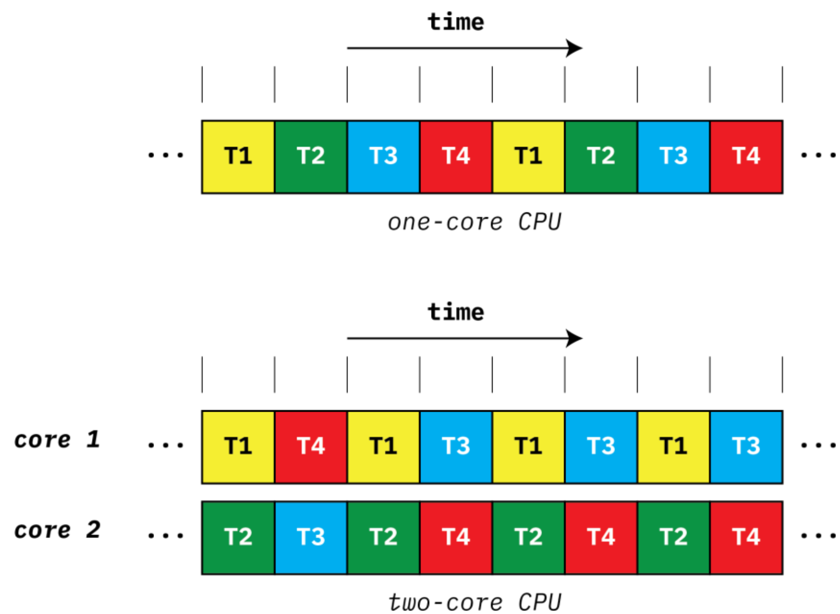
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process (open files, program-scope variables) which is easier to use than shared memory or message passing between processes
- **Economy** – thread creation is “cheaper” than process creation; thread switching has lower overhead than context switching
- **Scalability** – program in the process can be written to take advantage of multiprocessor architectures by adding and removing threads and processors become more or less available

Multicore Programming

- Multicore or multiprocessor systems do, however, put pressure on programmers
- Programming challenges include:
  - **Dividing up activities**
  - **Balancing activities**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- Hardware architecture support for software threads continues to increase over time
  - CPUs have **cores** as well as **hardware threads**
  - Example: Intel Xeon Platinum 8592+ processors has 64 cores (price: US$12K)
  - **Note: hardware threads are *not the same* as the software threads we're examining in this course.**



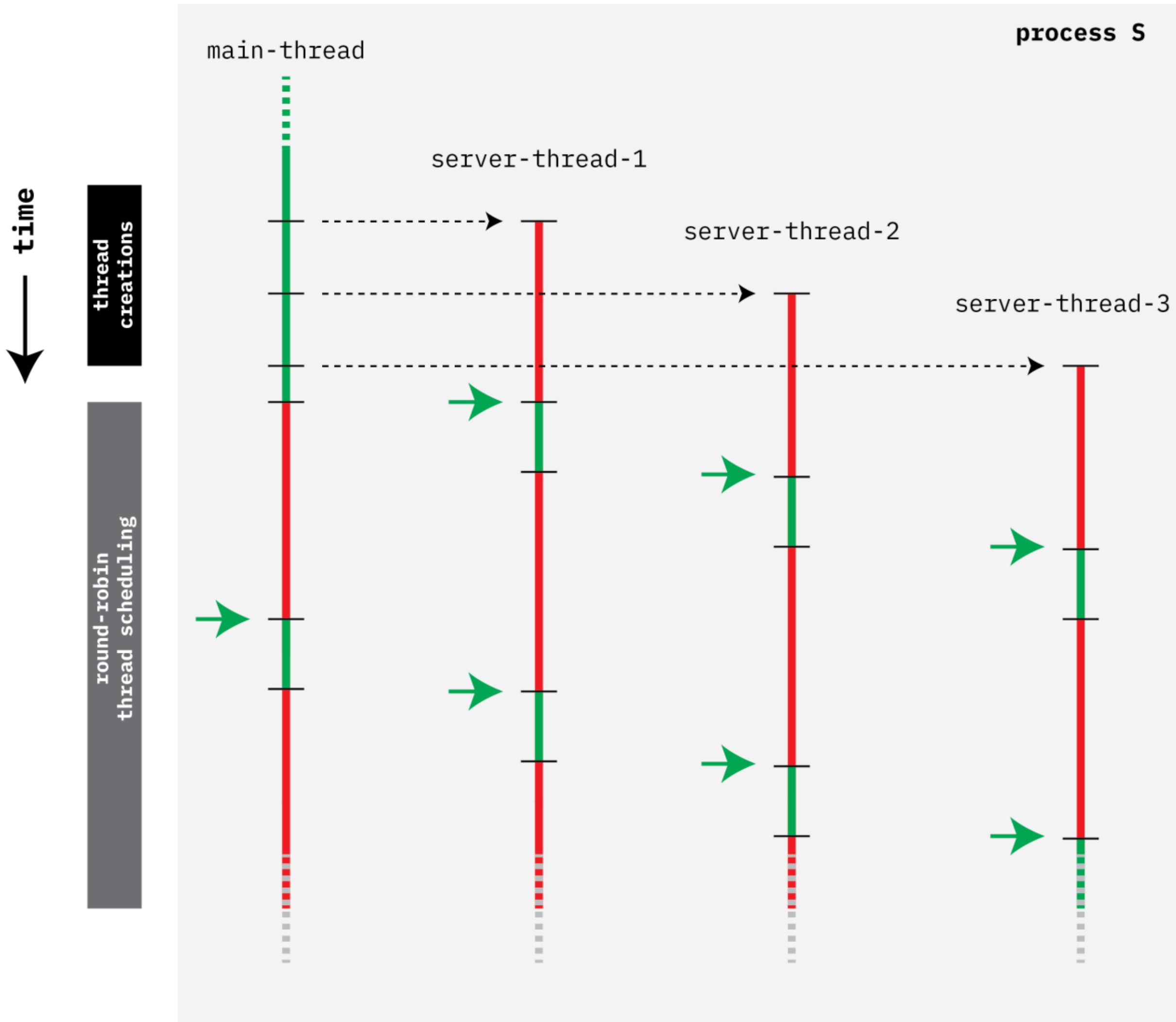
`pthread` (POSIX threads)

- The creation and management of threads is normally done through a **thread library**
- **We will use POSIX threads** (also known as `pthreads`)
  - Which means we will be using the pthread library installed with on `jhub-cosi`
- In what follows we will look at some examples of `pthread` library calls
  - thread **creation**
  - **parameterizing** threads
  - thread **attributes**
  - and more...
- More specific details on `pthreads` will eventually be provided as part of assignments and tutorials

Creating a POSIX Thread

```
void start_servers( ) {
    pthread_t thread;
    int i;
    for (i=0; i<nr_of_server_threads; i++)
        /* Making a biiiiiig assumption below, that thread
         * creation always succeeds. More robust code
         * must check value returned from pthread_create.
         */
        pthread_create(&thread,  /* pointer to thread ID */
            0,                   /* default attributes */
            server,              /* start routine */
            argument);           /* pointer to any argument */
}

void *server(void *arg) {     /* the worker thread */
    for (;;) {
        /* get and handle request */
    }
}
```

Threads in one process



(a wee complication)

```
rlogind(int r_in, int r_out, int l_in, int l_out)
{
    pthread_t in_thread, out_thread;

    pthread_create(&in_thread,
        0,
        incoming,
        r_in, l_out);       /* Sadly, cannot do this... */

    pthread_create(&out_thread,
        0,
        outgoing,
        l_in, r_out);       /* Can't do this, either... */

    /* How do we wait till both threads are done? */
}
```

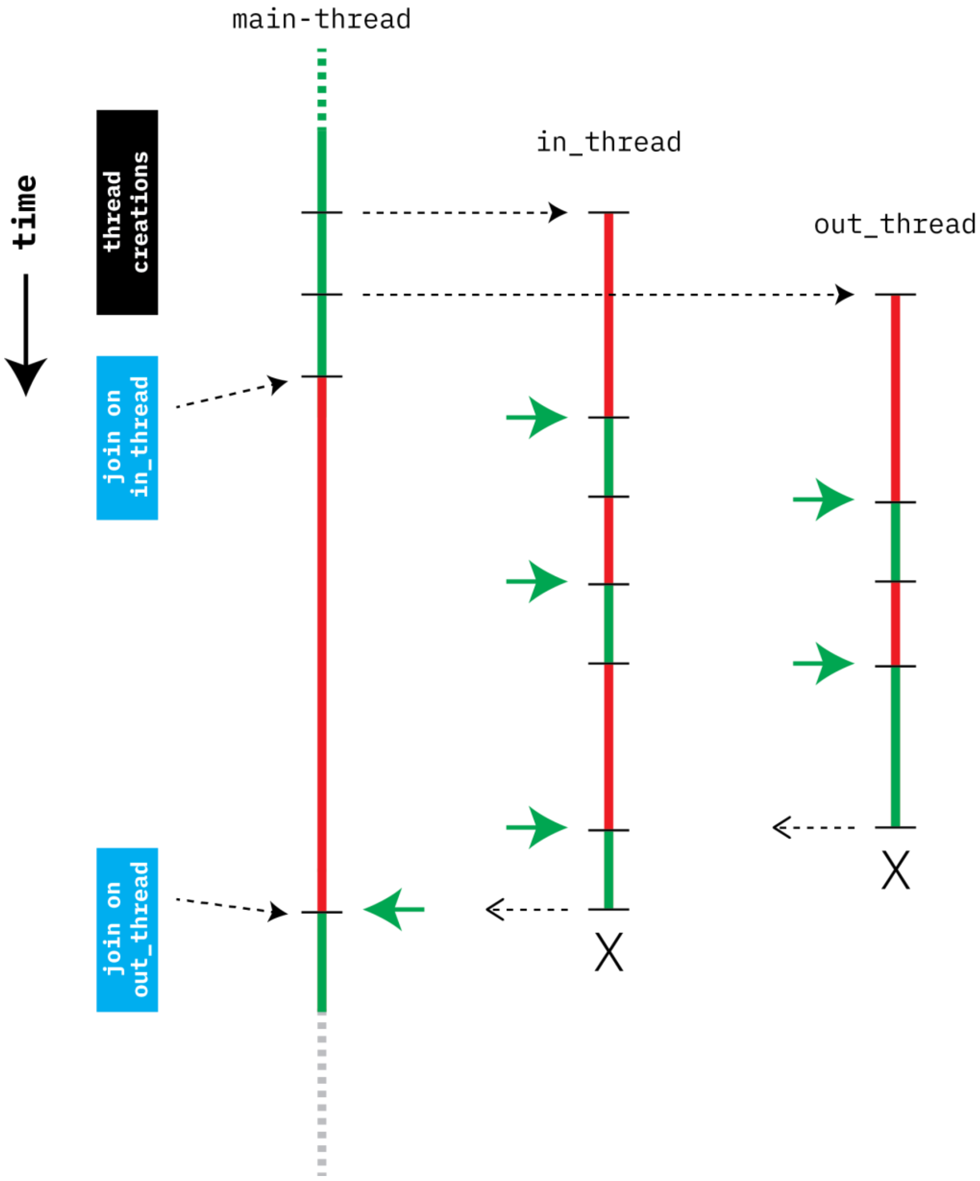
Multiple Arguments

```
typedef struct {
    int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;

    two_ints_t in = {r_in, l_out}, out = {l_in, r_out};

    pthread_create(&in_thread, 0, incoming, &in);
    pthread_create(&out_thread, 0, outgoing, &out);
}
```

When are the threads done?

```
void rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};

    pthread_create(&in_thread, 0, incoming, &in);
    pthread_create(&out_thread, 0, outgoing, &out);

    pthread_join(in_thread, 0);
    pthread_join(out_thread, 0);
}
```

- pthread_join() blocks execution until both threads complete their work.
- Ensures that both threads finish before rlogind (mainline code) exits.




```
void rlogind(int r_in, ...
     ...

     pthread_create(&in_thread, ...
     pthread_create(&out_thread, ...

     pthread_join(in_thread, 0);
     pthread_join(out_thread, 0);
}
```

Notice that even though **out_thread** completes before **in_thread**, the **main-thread** is blocked until it joins with **in_thread**.

Termination

- Careful use of several calls (signatures shown below)
  - #1 is completely different from regular process `exit(1)`
  - #2 may be needed to keep the compiler happy
  - #3 really only needed if the thread creator must explicitly wait for the created thread to complete its work.

```
/* 1 */ pthread_exit((void *) value);

/* 2 */ return((void *) value);

/* 3 */ pthread_join(thread, (void **) &value);
```

Detached Threads

```
void start_servers( ) {
    pthread_t thread;
    int i;
    for (i=0; i<nr_of_server_threads; i++) {
        pthread_create(&thread, 0, server, 0);
        pthread_detach(thread);
    }
    ...
}

server( ) {
    ...
}
```

Thread Attributes

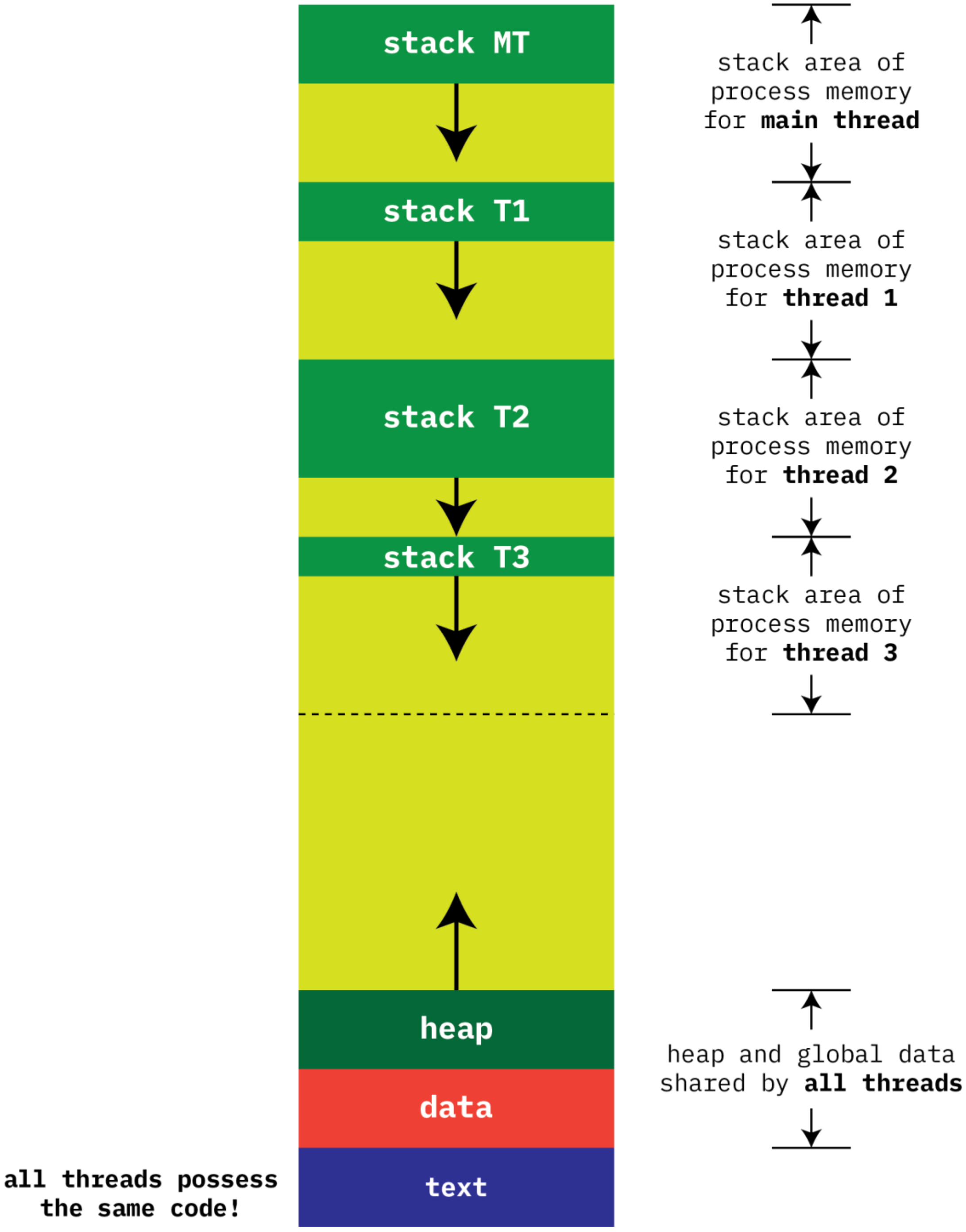
```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);

...
/* establish some attributes */
...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
pthread_attr_destroy(&thr_attr);
```

Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
pthread_attr_destroy(&thr_attr);
```

View of the process address space showing multiple local stacks.

Note: For simplicity, kernel portion of the address space is not shown.



User Threads and Kernel Threads

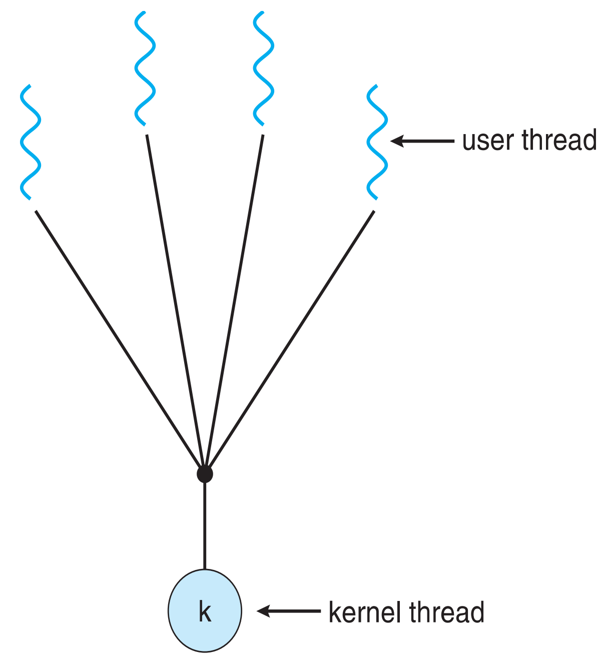
- **User threads**
  - Management performed by code in a user-level threads library
  - Kernel **is not aware of threads** (that is, any system call from any thread appears to the thread **as if it comes from the whole process**)
- Three primary thread libraries:
  - Traditional POSIX Pthreads
  - Windows threads
  - Java threads (but this requires a Java VM)
- **Kernel threads**
  - Supported by code in the kernel
  - Kernel **is aware of system calls** (including blocking ones) made by thread
- Examples: virtually all general purpose operating systems, including
  - Windows
  - Solaris
  - **Linux (i.e., `pthread`)**
  - Tru64 UNIX
  - macOS

Multithreading Models

- Many-to-One
  - Many user threads to one kernel thread
- One-to-One
  - One user thread to one kernel thread
- Many-to-Many
  - M user threads to N kernel threads

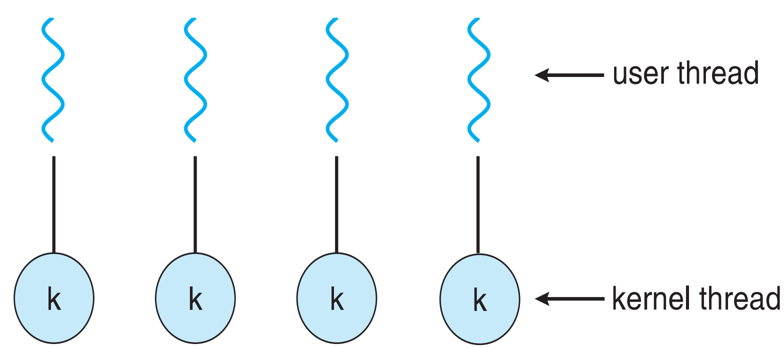
Many-to-One

- Many user-level threads mapped to single kernel thread
- **One thread blocking causes all to block**
- Multiple threads may not run in parallel on a multicore system because only one may be in kernel at a time
- Although this was the very first widely-implemented form of threading, **few systems currently use this model**
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



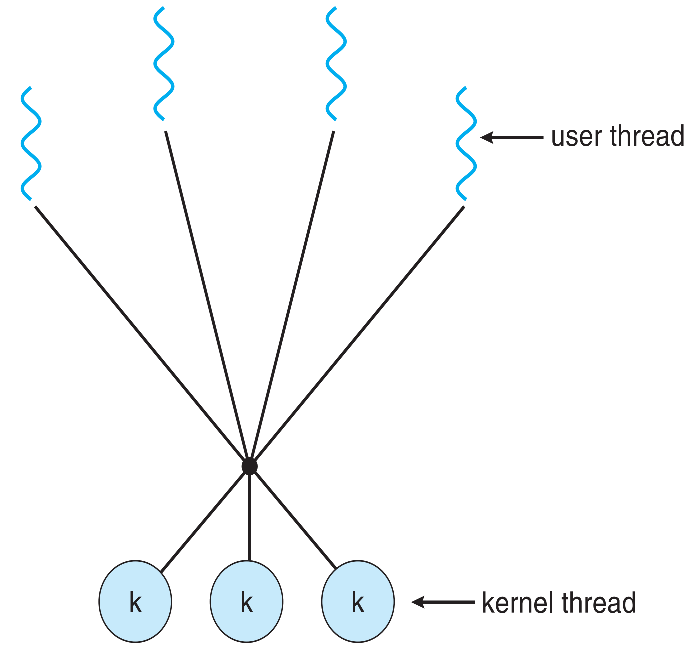
One-to-One

- Each user-level thread maps to kernel thread
- **Creating a user-level thread creates a kernel thread**
- More physical concurrency possible than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows NT
  - Linux
  - Solaris 9 and later



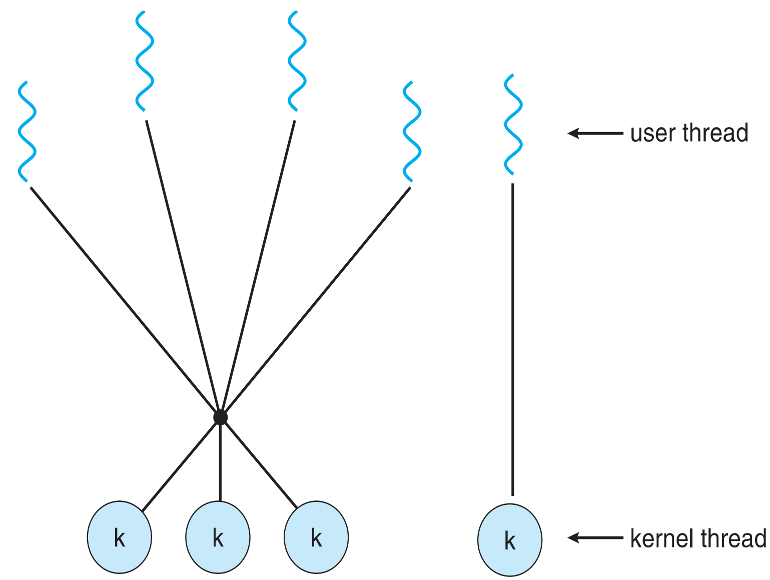
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
  - (Also known as an M to N -- or **M:N model**)
- Allows the operating system to **create a sufficient number of kernel threads**
- Solaris prior to version 9
- Windows with the ThreadFiber package



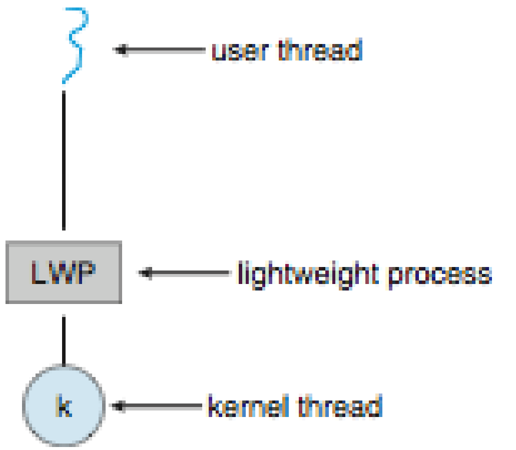
Two-level Model

- Similar to M:N, except that it allows a single user thread within a process to be bound to single kernel thread
  - That is, all threads are in the same process...
  - ... but some are M:N, some are 1:1
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Scheduler Activations

- Both M:N and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads **– lightweight process (LWP)**
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls**
  - a communication mechanism from the kernel to the upcall handler in the thread library
  - This communication allows an application to maintain the correct number kernel threads



Linux Threads

- Linux refers to them as **tasks** rather than **threads**
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
  - Flags control behavior
- `struct task_struct` points to process data structures (shared or unique)

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.

Summary

- Threads are an important structural mechanism for programmatic expression of concurrency
  - Easier to write an application using multiple threads (i.e. compared with using multiple processes)
  - Threads can be implemented on top of a systems process model
- There are many different realizations of threads
  - Our course will focus on POSIX threads (`pthreads`)
  - Different models represented different relationships between user-level threads and kernel-level threads
- Creating threads is (oddly enough) the easy part!
  - **Ensuring threads collaborate together in safe ways is the hard part.**
  - We look next at this hard part...