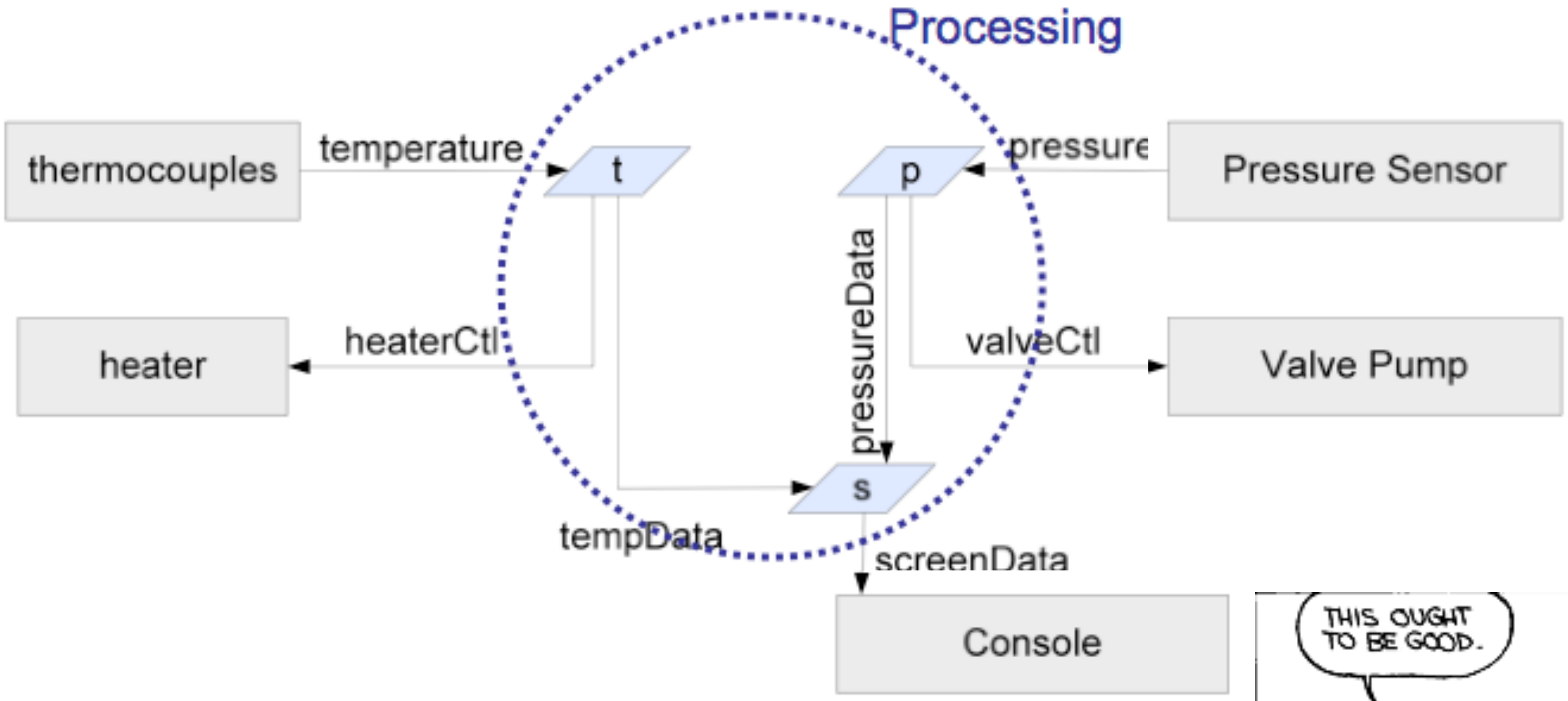
Processes and threads

Multiple threads or processes
The difference between a process and a thread
Kernel-level threads and user-level threads
POSIX functionality for multithreading
Communication between processes and threads

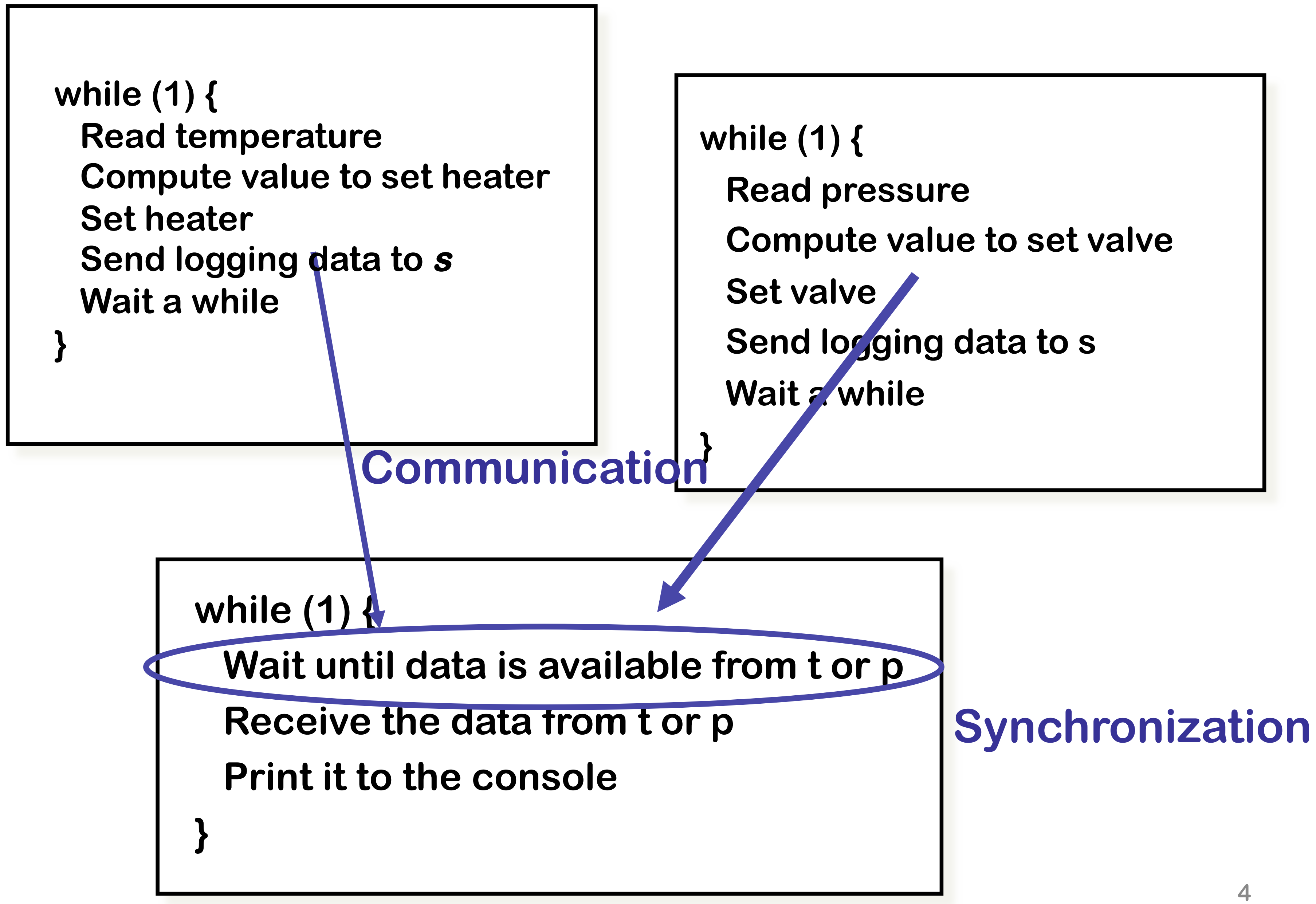
Material covered so far

- Processor scheduling
  - –Periodic tasks with static and dynamic priorities
    - Utilization bounds
    - Exact tests
- Operating system support
  - –The POSIX standard: an introduction
- *Now: Processes, threads, and POSIX support*

Example



We could write separate threads/processes for ***t***, ***p***, and ***s***.



```
while (1) {
   Read temperature
   Compute value to set heater
   Set heater
   Send logging data to s
   Wait a while
}
```

```
while (1) {
   Read pressure
   Compute value to set valve
   Set valve
   Send logging data to s
   Wait a while
}
```

```
while (1) {
   Wait until data is available from t or p
   Receive the data from t or p
   Print it to the console
}
```

Support for multiple processes/threads

- A real-time OS needs to provide:
- An API that allows the user to:
  - Create and kill processes and threads
  - Communicate between processes and threads
- A method to time-slice processes and threads on one or more CPUs such that:
  - Hard Deadlines are obeyed (first priority)
  - Soft Deadlines are obeyed (second priority)
  - Deadlocks do not occur (another first priority)
    - Scheduling policies that we have studied so far (at least in part)

Threads and processes

- What is the difference between a thread and a process?
- Recall that a process includes many things:
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, registers, etc.)
- Creating a new process is costly because of all of the
  - data structures that must be allocated and initialized
  - FreeBSD: 81 fields, 408 bytes
- …which does not even include page tables, etc.
- Communicating between processes is costly because
  - Most communication goes through the OS
  - Overhead of system calls and copying data

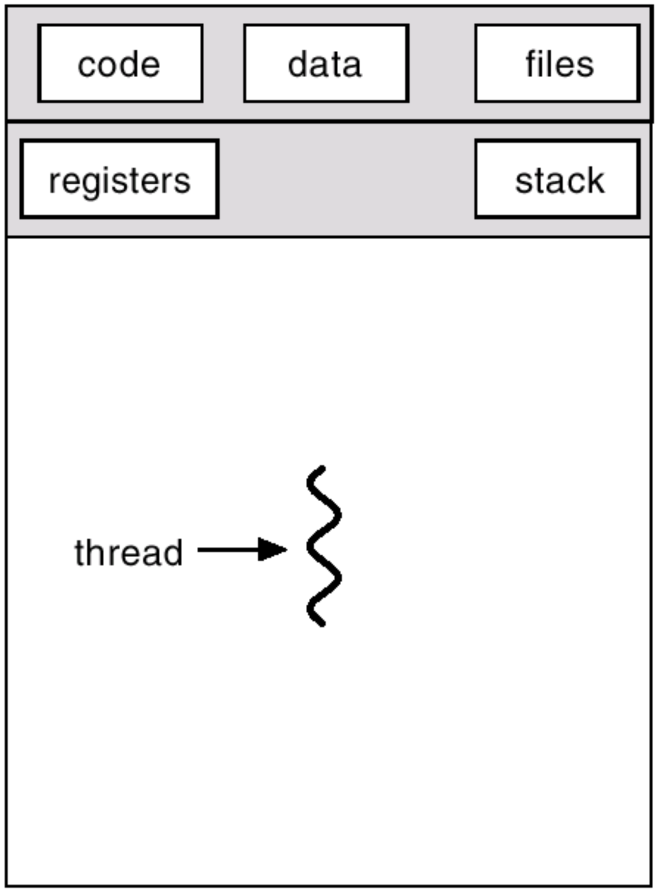
Rethinking processes

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)
- What don’t they share?
  - Each has its own execution state: PC, SP, and registers
- Key idea: Why don’t we separate the concept of a process from its execution state?
  - Process: address space, privileges, resources, etc.
  - Execution state: PC, SP, registers
- Exec state also called thread of control, or thread

Threads

- Modern OSes separate the concepts of processes and threads:
  - The *thread* defines a sequential execution stream within a process (PC, SP, registers)
  - The *process* defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities

Threads



This is a single-threaded process (a process with only one thread)

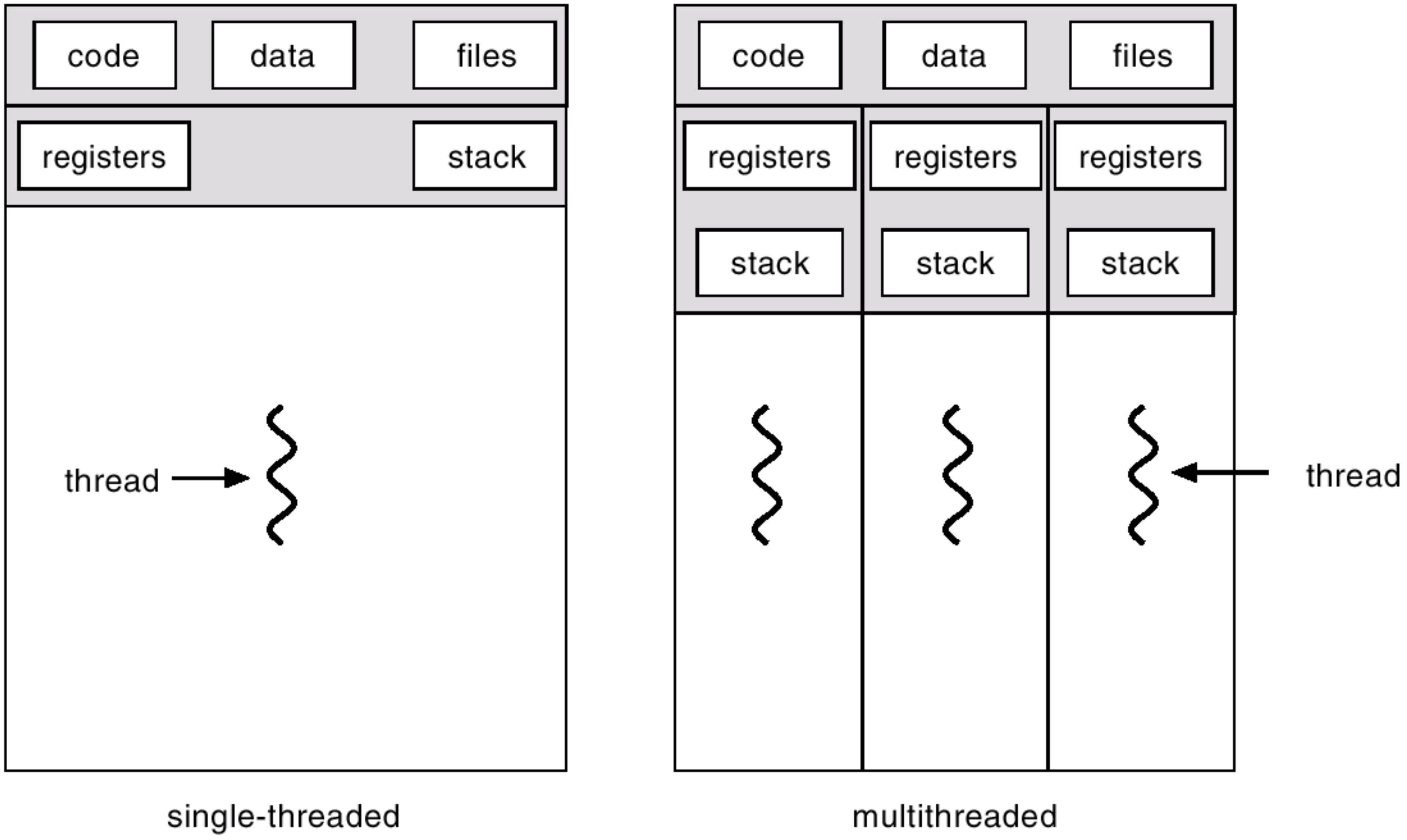
You could have multiple interacting processes of this type.

Each process would maintain its own: address space contents

- registers
- program counter
- stack
- state of system calls state of system calls
- all files from the process and their state

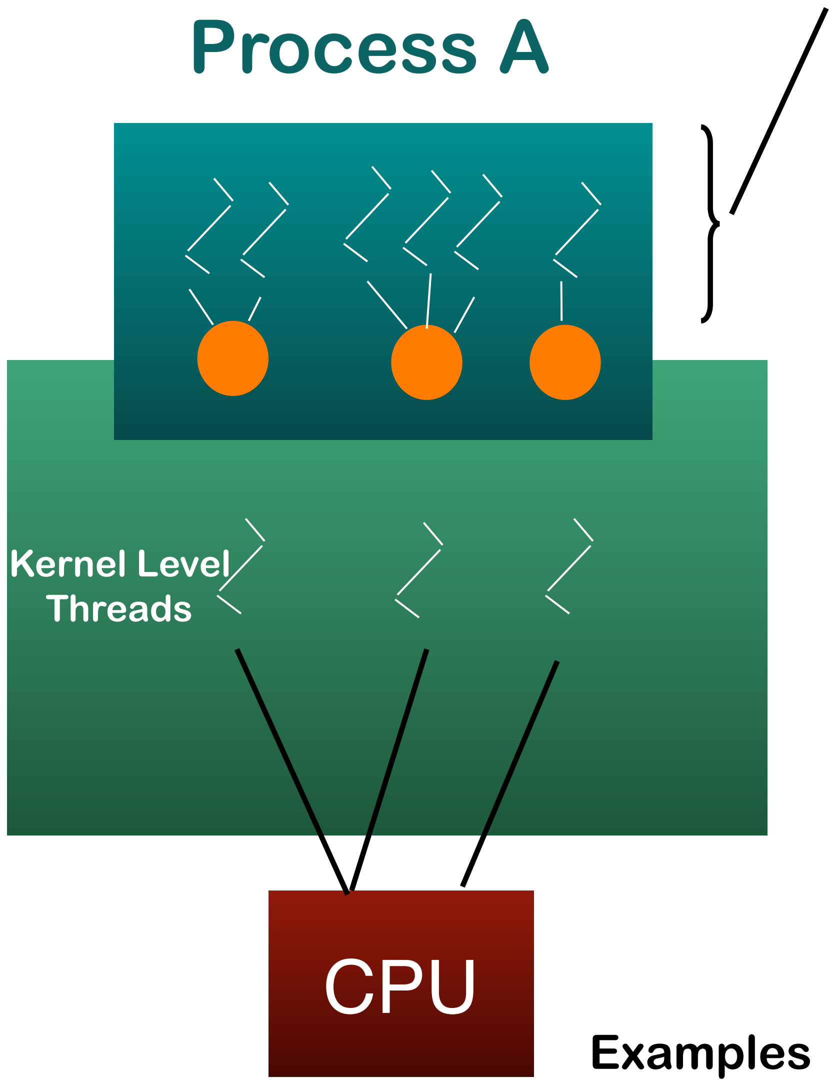
Threads

Threads belonging to the same process shares its code, data section, & other O/S resources e.g. open files, signals, etc.



Kernel-level threads vs. user-level threads

- Another distinction:
  - Kernel-Level Threads: the threads that the O/S kernel “knows about”, i.e., the kernel switches between kernel threads using some kernel-defined scheduling strategy
  - User-Level Threads: Threads within an application that the kernel is unaware of



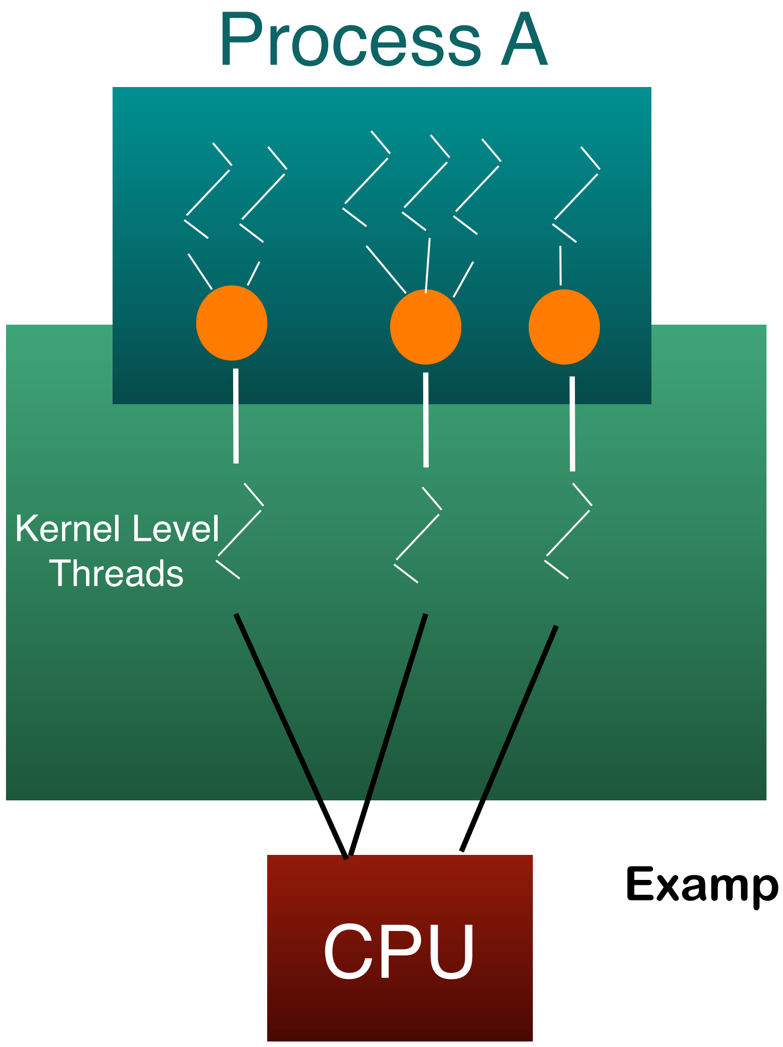
User-level threads (ULT)

- Supported above the kernel
- Exist only within a process
- Implemented by a thread library at user level
- ULTs in process A cannot access a ULTs in process B
- Used by programmers to handle multiple flows of control within a program
- Library provides support for thread creation, scheduling management with no support from the kernel
- Kernel is unaware of user level threads. All thread creation and scheduling are done in user space without the need for kernel intervention – thus ULTs are generally fast to create and manage

Examples

POSIX Pthreads

Solaris threads



Kernel threads

- Supported directly by O/S
- The O/S will have a separate thread for each process and that will perform O/S activities on behalf of the O/S
- The kernel performs thread creation, scheduling & management in kernel space
- Thread management is carried out by O/S and kernel threads are generally slower to crate and manage than user threads

Examples

- Windows 95/98/NT/2000
- Solaris
- Tru64 UNIX
- Linux

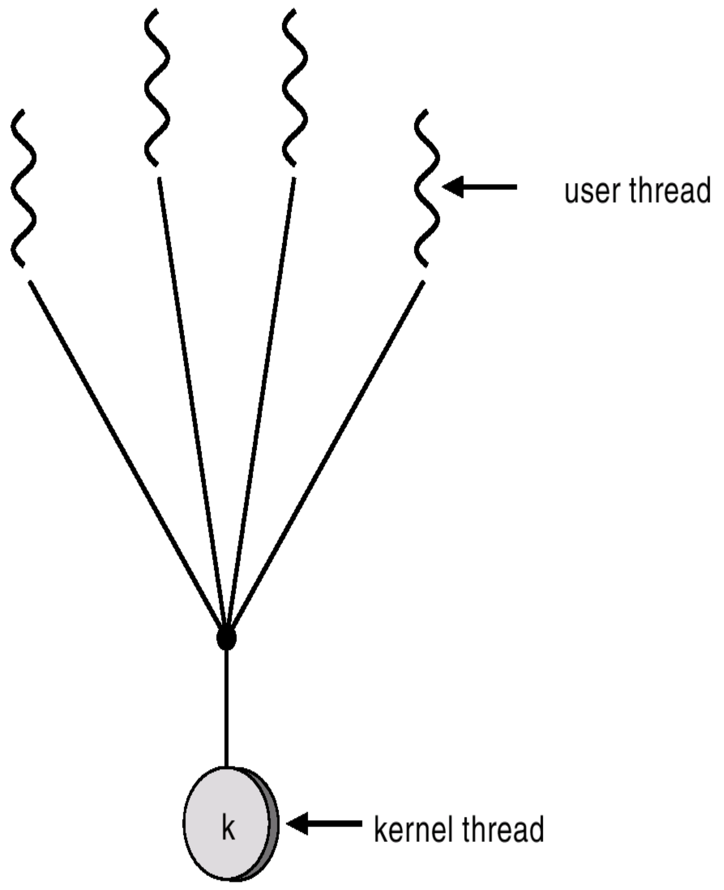
Multithreading modes

- When an OS supports more than one kernel thread per process, then
- possible to multiplex the execution of different sets of user threads on
- different kernel threads

- Different models exist for mixing and matching kernel and user threads:
  - One-to-one
  - Many-to-one
  - Many-to-many

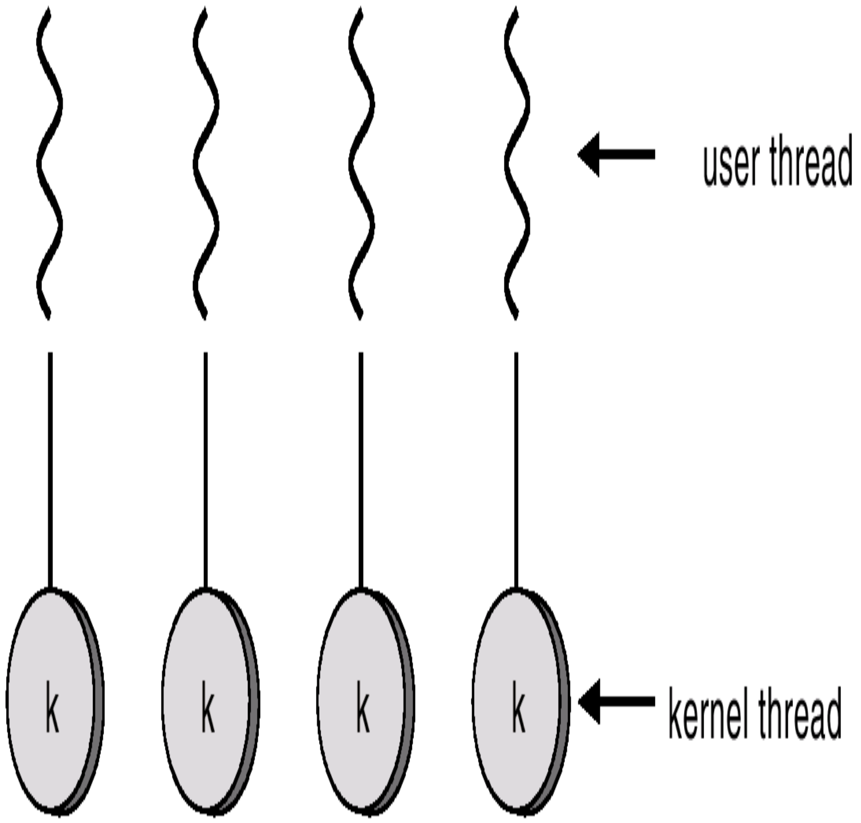
Multithreading models: Many-to-one

- Maps many ULTs to one KLT
- Advantage:
  - Thread management is carried out in user space so it is efficient
- Disadvantages:
  - The entire process will block if a thread makes a blocking system call
  - Only one thread can access the kernel at a time, thus multiple threads are unable to run in parallel on multiprocessors
- Example: Solaris 2



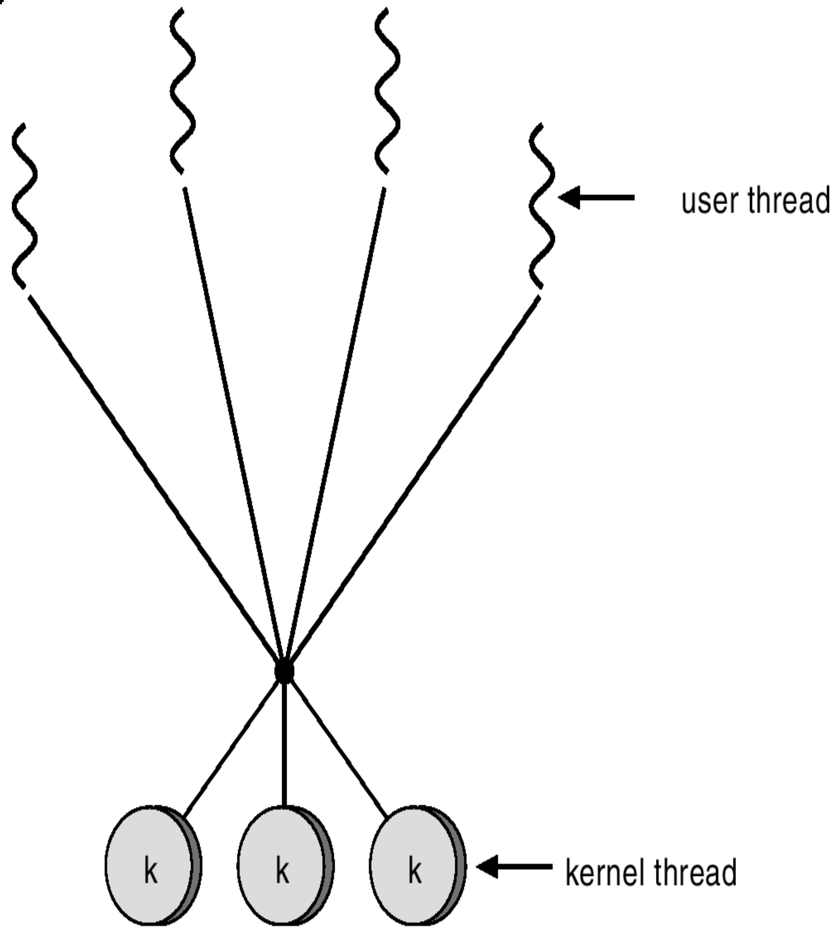
Multithreading models: One-to-one

- Map each user thread to a kernel thread
- Advantages:
  - Provides more concurrency by allowing another thread to run when a thread makes a blocking system call
  - Allows multiple threads to run in parallel on multiprocessors
- Disadvantages:
  - Creating a user thread requires creating a corresponding kernel thread which can burden the performance of an application
  - Most implementations of this model restrict the number of threads supported by the system
- Examples: Windows NT, 2000, OS/2

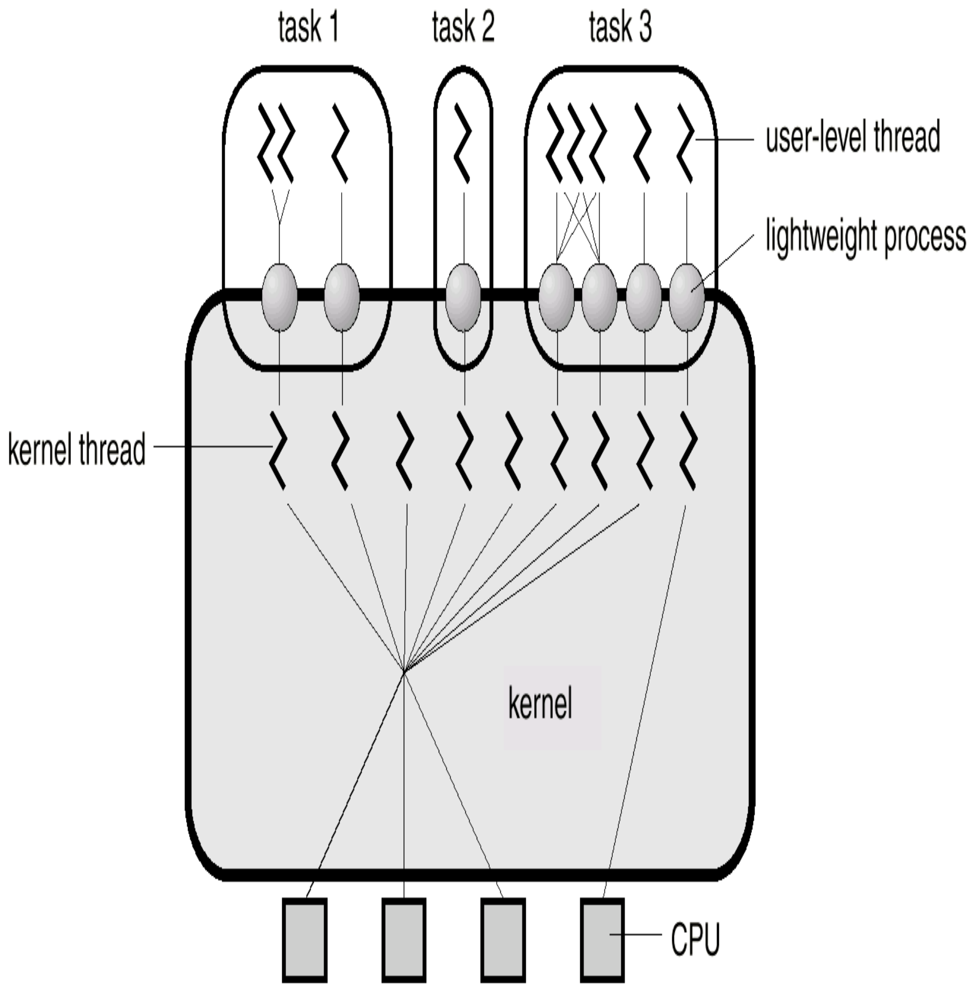


Multithreading models: Many-to-many

- Multiplexes many ULTs to a smaller or equal number of kernel threads
- The number of threads may be specific to either a particular application or a particular machine
  - Example: An application may be allocated more kernel threads on a multiprocessor than on a uniprocessor
- Advantages
  - Developers can create as many user threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor
  - When a thread performs a blocking system call, the kernel can schedule another thread of execution
  - Allows the operating system to create a sufficient number of kernel threads
- Examples: Solaris 2, Windows NT/2000



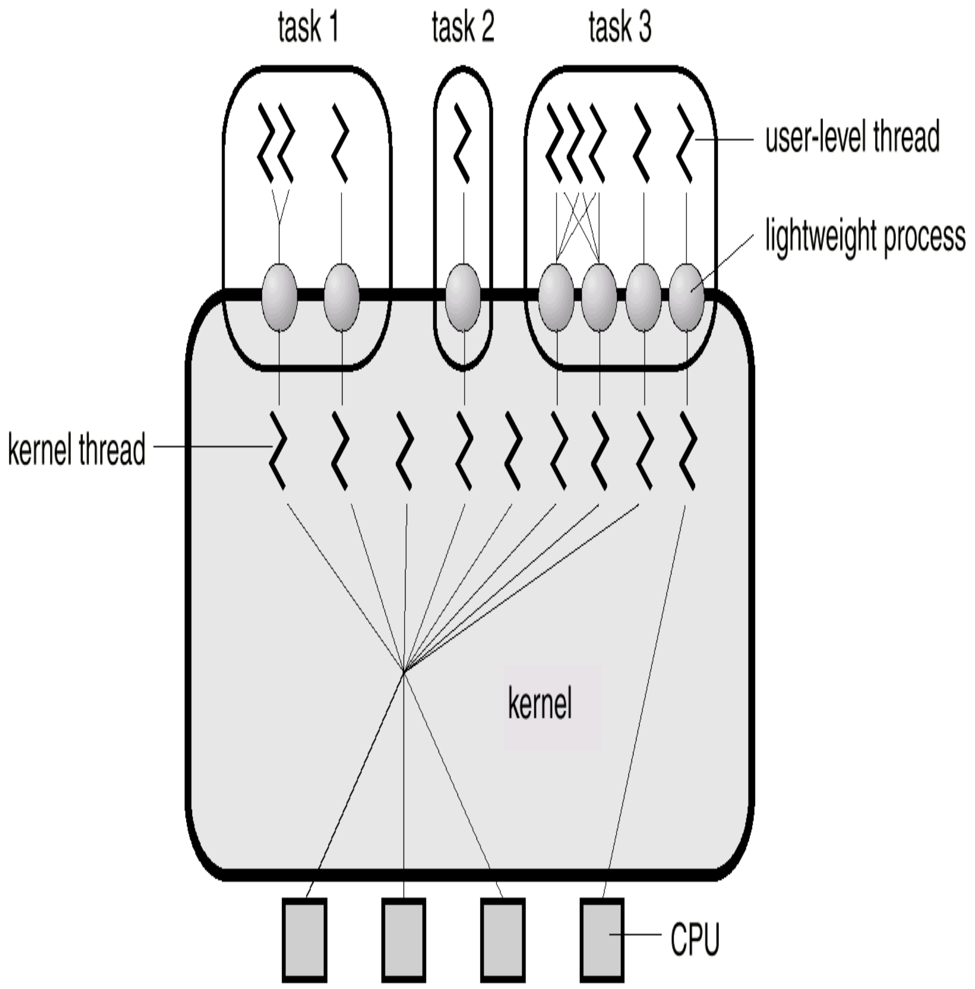
Solaris 2 as an example



Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, SMP, and real-time scheduling

Standard kernel-level threads execute all operations within the kernel. Each LWP processes (intermediate level of threads – each process contains at least 1 LWP) has a kernel level thread. Some KLT run on the kernel's behalf and have no associated LWPs e.g. a thread to service disk requests.

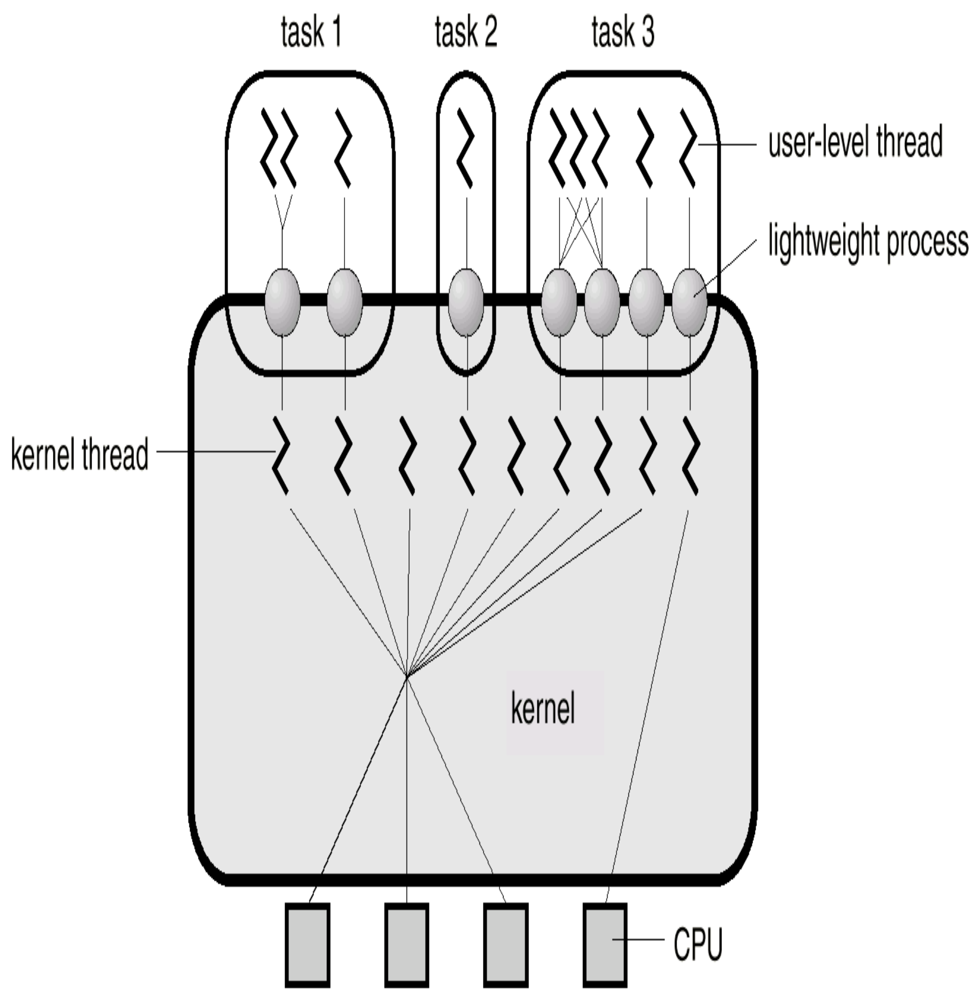
Solaris 2 as an example



Any one process may have many ULTs. These ULTs may be scheduled and switched among the LWPs by the thread library without kernel intervention.

ULTs are extremely efficient because no kernel support is required for thread creation or destruction, or for the thread library to context switch from one ULT to another.

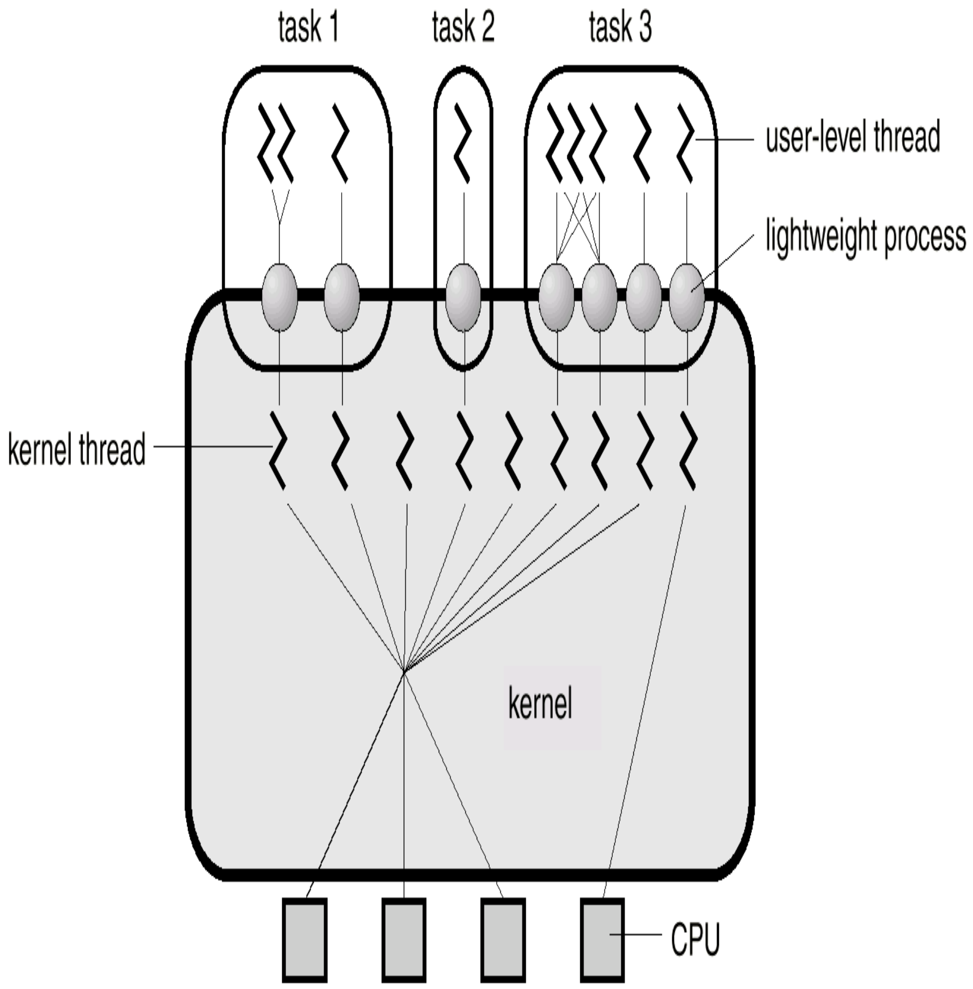
Solaris 2 as an example



ULT threads may be either :

- Bound (Permanently attached to an LWP)
    - Only that thread runs on LWP and request the LWP can be dedicated to a single processor (see rightmost one)
    - Binding a thread is useful in situations that require quick response time e.g. a real-time applications
- Unbound (Not permanently attached to any LWP process)
    - All unbound threads in an application are multiplexed into the pool of available LWPs for the application

Solaris 2 as an example



The kernel threads are scheduled by the kernel's scheduler and execute on the CPU or CPUs in the system

If a kernel thread blocks, the CPU is free to run another kernel thread

- If the thread that blocked was running on behalf of an LWP, the LWP blocks as well
- The ULT currently attached to the LWP also blocks
- If the process has more than one LWP, the kernel can schedule another LWP

Examples using processes and threads

- Creating new processes (using *fork*)
- Creating new threads (using pthreads)

Spawning new processes (C++ example)

```
#include <iostream>
#include <string>

// Required by for routine
#include <sys/types.h>
#include <unistd.h>

using namespace std;

int globalVariable = 2;

main()
{
  string sIdentifier;
  int    iStackVariable = 20;

  pid_t pID = fork();
  if (pID == 0)              // child
  {
    // Code only executed by child process

    sIdentifier = "Child Process: ";
    globalVariable++;
    iStackVariable++;
  }
  else if (pID < 0)          // failed to fork
  {
     cerr << "Failed to fork" << endl;
     exit(1);
     // Throw exception
  }
  else                       // parent
  {
   // Code only executed by parent process

   sIdentifier = "Parent Process:";
  }

  // Code executed by both parent and child.

  cout << sIdentifier;
  cout << " Global variable: " << globalVariable;
  cout << " Stack variable: "  << iStackVariable << endl;
}
```

OUTPUT

```
Parent Process: Global variable: 2 Stack variable: 20
Child Process:  Global variable: 3 Stack variable: 21
```

Creating a thread

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

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

   /* Create independent threads each of which will
      execute function */

    iret1 = pthread_create( &thread1, NULL,
       print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL,
       print_message_function, (void*) message2);

    /* Wait till threads are complete before main
       continues. Unless we wait we run the risk of
       executing an exit which will terminate the
       process and all threads before the threads have
       completed */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

OUTPUT

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

More on pthread_create

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

Arguments

- **thread**: returns the thread identifier
- **attr**: can be NULL for the default options
  - other options are:
    - **detached state** (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
    - **scheduling policy** (real-time? PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED, SCHED_OTHER)
    - **scheduling parameter**
    - **inheritsched attribute** (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
    - **scope** (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS. Pick one or the other not both.)
    - **stack address** (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
    - **stack size** (default minimum PTHREAD_STACK_SIZE set in pthread.h),
- **void * (*start_routine)**: pointer to the function to be threaded
- ***arg**: pointer to function arguments

Many other pthread functions

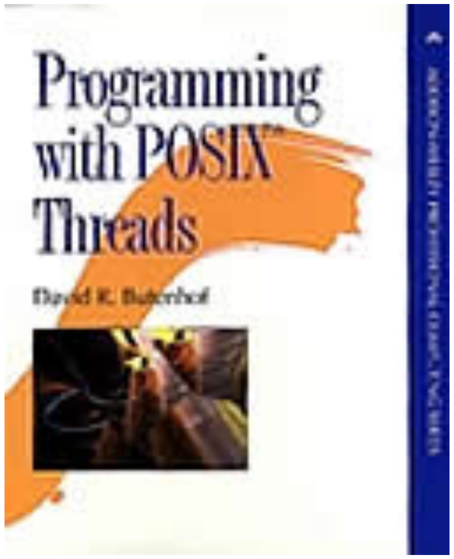
- pthread_cancel
- pthread_cleanup_pop
- pthread_cleanup_push
- pthread_create
- pthread_equal
- pthread_exit
- pthread_getschedparam
- pthread_getspecific
- pthread_join

- pthread_key_create
- pthread key_delete
- pthread_kill
- pthread_once
- pthread_self
- pthread_setcancelstate
- pthread_setspecific
- pthread_sigmask
- pthread_testcancel
- sched_yield

To get more information on POSIX threads

- The web site: http://www.opengroup.org/onlinepubs/007908799/
- Or this tutorial: http://www.llnl.gov/computing/tutorials/pthreads/
- Or Butenhof's book
- And many other resources on the WWW

- *Remember: this is not a course on using threads*
  - *We will use threads, but*
  - *You will need to learn most features on your own*
  - *We will discuss only some essential details*



Highlights

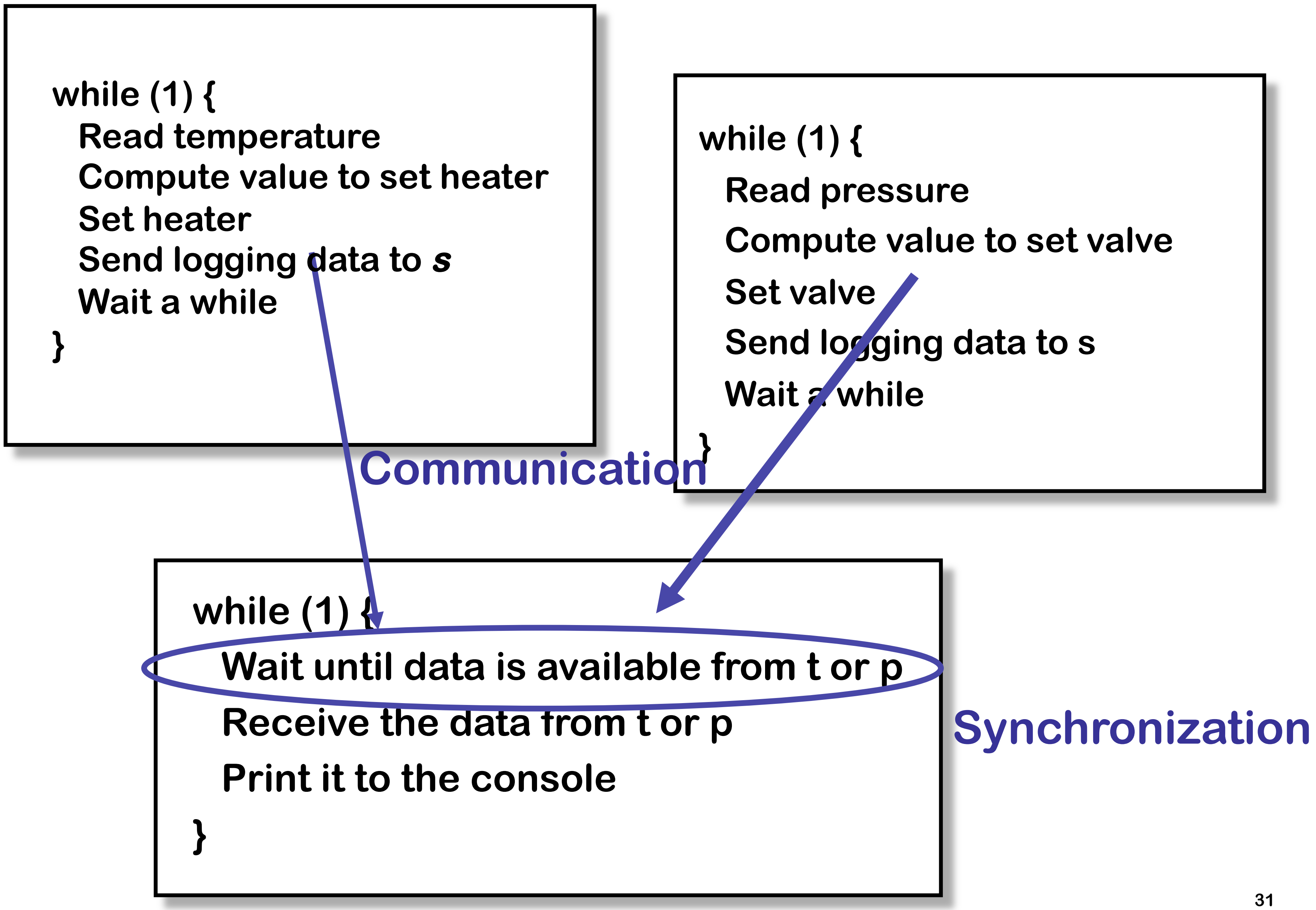
- We can simplify programming by employing multiple threads / processes
- Threads vs. Processes:
  - *Thread:* a sequential execution stream within a process
  - *Process:* defines the address space and general process attributes
- User-level threads vs. Kernel-level threads:
  - *Kernel-Level Threads*: threads that the O/S kernel "knows about"
  - *User-Level Threads*: threads that the kernel is unaware of
- POSIX programming:
  - Processes: fork/join
  - pthreads: pthread_create, pthread_join, pthread_detach, etc.
- We can implement a program consisting of multiple tasks.
- But, unless these tasks can talk to each other, they are not very useful.
- Later, we will discuss typical RTOS/OS communication mechanisms.

Communicating between processes and threads

OS features for exchanging information between concurrent tasks

Outline

- In this slide set, you will learn about various mechanisms that are available for communication between processes.
- To make the discussion concrete, we will focus on the mechanisms provided in POSIX.1 and POSIX.4
    - Signals
    - Pipes
    - FIFOs
    - Message Queues
    - Shared Memory
- By the end of these slides, you should be able to look at a RTOS and be able to understand what communication mechanisms are available.



```
while (1) {
   Read temperature
   Compute value to set heater
   Set heater
   Send logging data to s
   Wait a while
}
```

```
while (1) {
   Read pressure
   Compute value to set valve
   Set valve
   Send logging data to s
   Wait a while
}
```

```
while (1) {
   Wait until data is available from t or p
   Receive the data from t or p
   Print it to the console
}
```

Communication mechanisms

- If an RTOS is to support any multiple process/multiple thread design, it must provide mechanisms for communication between processes or threads.
- When you go out in the real world, and are evaluating RTOSs for your real-time system, you need to consider this.
- There are many different mechanisms for providing this communication
  - Several of these are provided in POSIX.1/POSIX.4/Pthreads
  - By looking at some of these, we will get a good overview of what is available
- We also need synchronization, but we'll come back to that later.

Communication mechanisms

- If we have two tasks that want to talk to each other:

1. Signals: used in POSIX to send a short messages between processes
2. Pipes: A simple "mailbox" between one process and another
3. FIFOs: A simple "mailbox" with more than one entry
4. Messages: Useful for sending a message from one process to another. Unlike the other choices above, message queues can be set up for any width. (Not supported in our implementation of POSIX.4.)
5. Shared Memory
    1. Between Processes: abstraction provided by POSIX
    2. Between Threads: true shared memory

POSIX signals

- Software equivalent of an interrupt or exception
  - When a process “gets” a signal, it knows that something has happened which requires the process’s attention
- Might happen as a result of
  - Software exceptions (e.g. divide-by-zero error)
- So for example, you might define a “handler” that
  - handles “division by zero” errors; Control-C (stop), Control-Z (job control); Timer expiration
  - Another process explicitly sends a signal
- Normally, not used to communicate data
  - POSIX.4 signals allow you to attach a 32 bit payload to signals

POSIX signals

- Analogous to Hardware Interrupts:
  - When programming with Hardware Interrupts, you write an *interrupt service routine* (ISR) and set up a vector so when the interrupt occurs, your ISR is called. The ISR handles the interrupt.
- When programming with signals, you write a *signal handling routine*.
  - You call an O/S routine called sigaction to indicate that the signal handler should be called when the signal arrives.
  - The signal handling routine does all processing to react to the signal.

POSIX.1 Signals

- There are a number of signal types defined, each represented by a number:
  - SIGFPE: Floating Point Exception
  - SIGILL: Illegal Instruction
  - SIGSEGV: Memory access Exception
  - SIGQUIT: User hits a Control-C
  - SIGUSR1: User Signal 1
  - SIGUSR2: User Signal 2
  - (there are several others…)

Constants (Integers)

As a programmer, you can use

these two signal types

Setting up a signal handler for SIGUSR1 or SIGUSR2

- A signal handler is a function with one argument and void return value
- void my_signal_handler (int signo)
- {
- // Handle your signal here
- }
- Parameter signo contains the signal type that caused us to get here (eg. SIGUSR1, SIGUSR2)

Telling the OS What to do when a signal arrives

- int sigaction (SIGNO, &sa, NULL)

Returns type int

(0 means success)

Don't worry about this for now

Signal Number (eg. SIGUSR1 or SIGUSR2)

```
struct sigaction {
    void (*sa_handler)();
    sigset_t sa_mask;
    int sa_flags;
    .....
}
```

Signal handler routine

List of signals that are masked while we are in signal handler

Flags. Just set to 0 for now

```
my_sig_handler (int signo)
{
     printf("I received signal %d\n",signo);
     /* Do something interesting */
}
```

```
struct  sigaction  my_params;
int  success;

my_params.sa_handler  =  my_sig_handler;
my_params.sa_flags  =  0;
sigemptyset(&my_params.sa_mask);
success = sigaction (SIGUSR1,  &my_params,  NULL);

/* From now on, whenever someone sends a SIGUSR1, we will
   enter routine my_sig_handler  */
```

Sending signals

```
int  kill  (pid_t  destination_pid,   int  signo);
```

Returns type int

(0 means success)

PID (process id) of destination process

Signal Number (e.g., SIGUSR1 or SIGUSR2)

Problems with POSIX.1 signals

- Only two signals for the user to utilize
- Signals are not queued correctly
  - May be delivered out of order
- No information is contained in the signal
  - Apart from the fact that it was generated
- High latency
  - Need to enter the signal handler every time a signal is triggered

POSIX.4 signals

- Improvements over POSIX.1
- There are more signals (SIGRTMIN to SIGRTMAX)
    - The exact number depends on the system
    - Up to 32 application-defined signals
    - ssh-linux.ece.ubc.ca: 32
    - ssh.ece.ubc.ca: 8
- Signals are queued
- Signals can carry a 32-bit payload
- Signals can be prioritized

Using POSIX.4 signals

```
int  sigqueue  (pid_t  destination_pid,   int  signo,  union sigval payload);
```

pid (process id) of destination process

Signal number (e.g., SIGRTMIN)

The payload you want to send

```
union sigval {
    int sival_int;
    void *sival_ptr;
};
```

Using POSIX.4 signals

```
void  my_signal_handler (int signo,  siginfo_t *info,  void *ignored)
```

```
typedef struct {
    ....
    union sigval si_value;
} siginfo_t;
```

```
union sigval {
    int sival_int;
    void *sival_ptr;
} ;
```

Use one of these, not both

(since it is a “union”, both sival_int and sival_ptr refer to the same

32-bit quantity. You can access this quantity using either name.)

```
my_sig_handler (int signo, siginfo_t *info, void *ignored))
{
    printf("I received signal %d\n",signo);
    printf("The payload was: %d\n", info->si_value.sival_int);
}
```

32 bit payload received along with signal

```
struct  sigaction  my_params;
int  success;

my_params . sa_sigaction  =  my_sig_handler;
my_params . sa_flags  = SA_SIGINFO;
sigemptyset( & my_params . sa_mask);
success = sigaction (SIGRTMIN,  &my_params,  NULL);

/* From now on, whenever someone sends a SIGUSR1, we will
   enter routine my_sig_handler  */
```

Similar to sa_handler, but for new POSIX.4 signal handlers

Queued Signal

Looking for signals without invoking a handler

Sets up a "mask" containing only SIGRTMIN

Tells the OS to not call a signal handler for this signal

```
sigset_t  look_for_these;
siginfo_t  extra;
int status;
sigemptyset ( &look_for_these );
sigaddset ( &look_for_these,  SIGRTMIN );
sigprocmask( SIG_BLOCK,  &look_for_these,  NULL );
....
sigreceived = sigwaitinfo( &look_for_these,  &extra );
If  (sigreceived < 0) {
printf("Error waiting for signal\n");
}
else {
printf("received signal %d, payload %d\n", sigreceived, extra.
si_value.si_val_int);
}
```

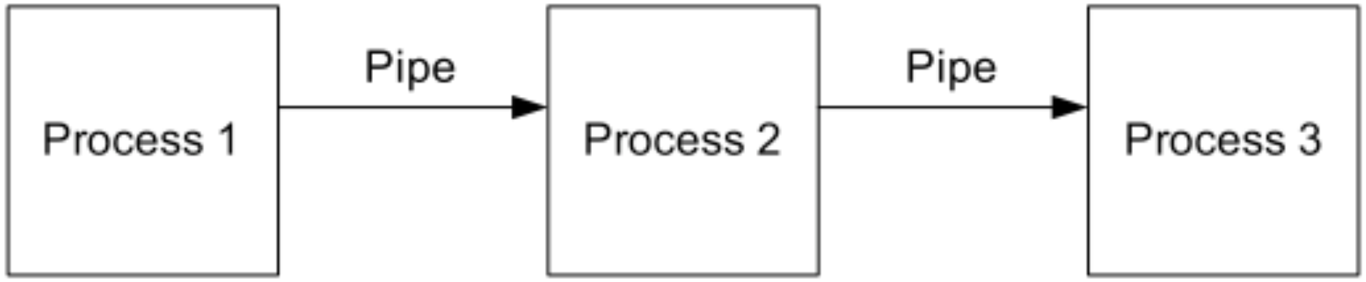
Block (suspend execution) until signal arrives

We received the signal and can look at the payload

Timeouts

- struct timespec timeout;
- timeout.tv_sec = 1;
- timeout.tv_nsec = 0;
- /* Set up “look_for_these” as before */
- …
- sigreceived = sigtimedwait (&look_for_these, &extra, &timeout);
- if (sigreceived < 0) {
-             printf(“Error: we probably timed out\n”);
- } else {
-             printf(“received signal %d, payload %d\n”, sigreceived,
-   extra. si_value . si_val_int);
- }

Pipes

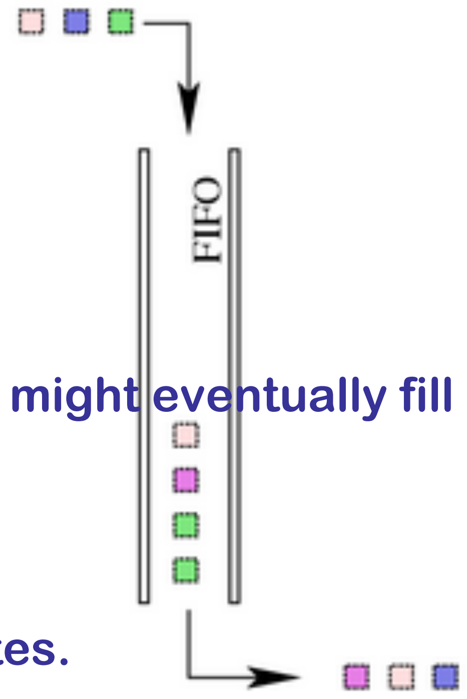


- Good for pipelined architectures:
- Set it up in a parent process, fork a child, then the child has access to the other end of the pipe
- Limitation: pipes are unidirectional
  - But you can set up two pipes, one going in each direction

FIFOs

- A FIFO is simply a pipe with a name.
- Because the pipe has a name, any process can access either end of the pipe.
  - Makes them a bit more flexible than a simple pipe.
- But there are still some problems:

1. Prioritization (all messages have the same priority)
2. Asynchronous operation (no matter how big, the FIFO might eventually fill up, and then you will block)
3. How many elements in the FIFO? No way to know
4. Lack of structure in the data stream: you just send bytes.
5. Limited numbers of pipes and FIFOs.



Message queues

- POSIX.4 provides a mechanism to create named queues:
- Queues can have any depth and width
    - Elements in a queue have a priority
    - Higher priority elements are popped off the queue before lower priority elements
    - No restrictions on which processes can access a queue
    - Just have to know the queue name
- Problems:
    - If you want to implement something that is not a queue (such as a stack), these queues won't help you.
    - Not very efficient: have to copy data from sender address space to receiver address space
- Note: Message queues not supported on ssh-linux.ece.ubc.ca.

Shared memory (POSIX.4)

- You can map a region of memory to one or more processes using the mmap function
  - Can share arbitrary data structures
  - Quite a lot of overhead
  - If you want shared memory, more natural to use threads
    - Remember: threads share an address space

Shared memory (Pthreads)

- int some_shared_data;
- void *thread1_routine(void *arg)
- {
  - can access variable some_shared_data
- }
- ....
- void *thread2_routine(void *arg)
- {
- can access variable some_shared_data
- }

Both threads see the same variable. Any variable can be defined this way.

Danger! Need some synchronization mechanism if we are going to use shared memory in this way!

Highlights

- We discussed various mechanisms for communications between processes
- We focused on POSIX mechanisms.
- The exact details of these mechanisms aren’t so important (*except for the programming assignments*).
    - What is important is to have a good overview of the range of communication mechanisms that can be provided and how they compare.
- The most natural form of communication is shared memory.
    - However, to make shared memory work well, synchronization mechanisms are required.