Programming with Threads in Unix

Topics

- Processes and Threads
- Posix threads (Pthreads) interface
- Shared variables
- The need for synchronization
- Synchronizing with semaphores
- Synchronizing with mutex and condition variables
- Thread safety and reentrancy

Traditional view of a UNIX process

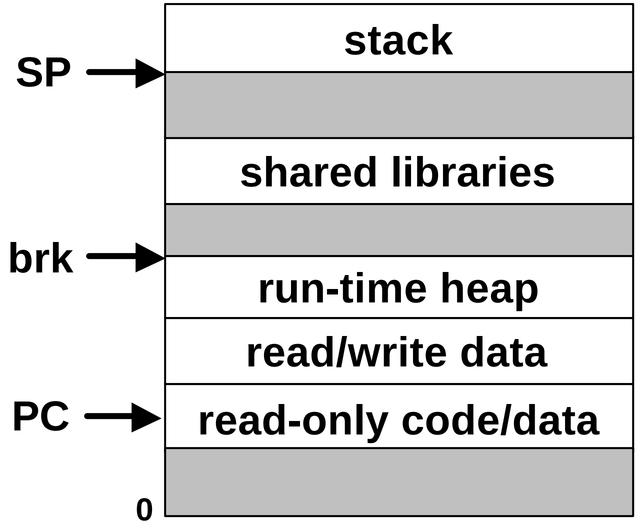
Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

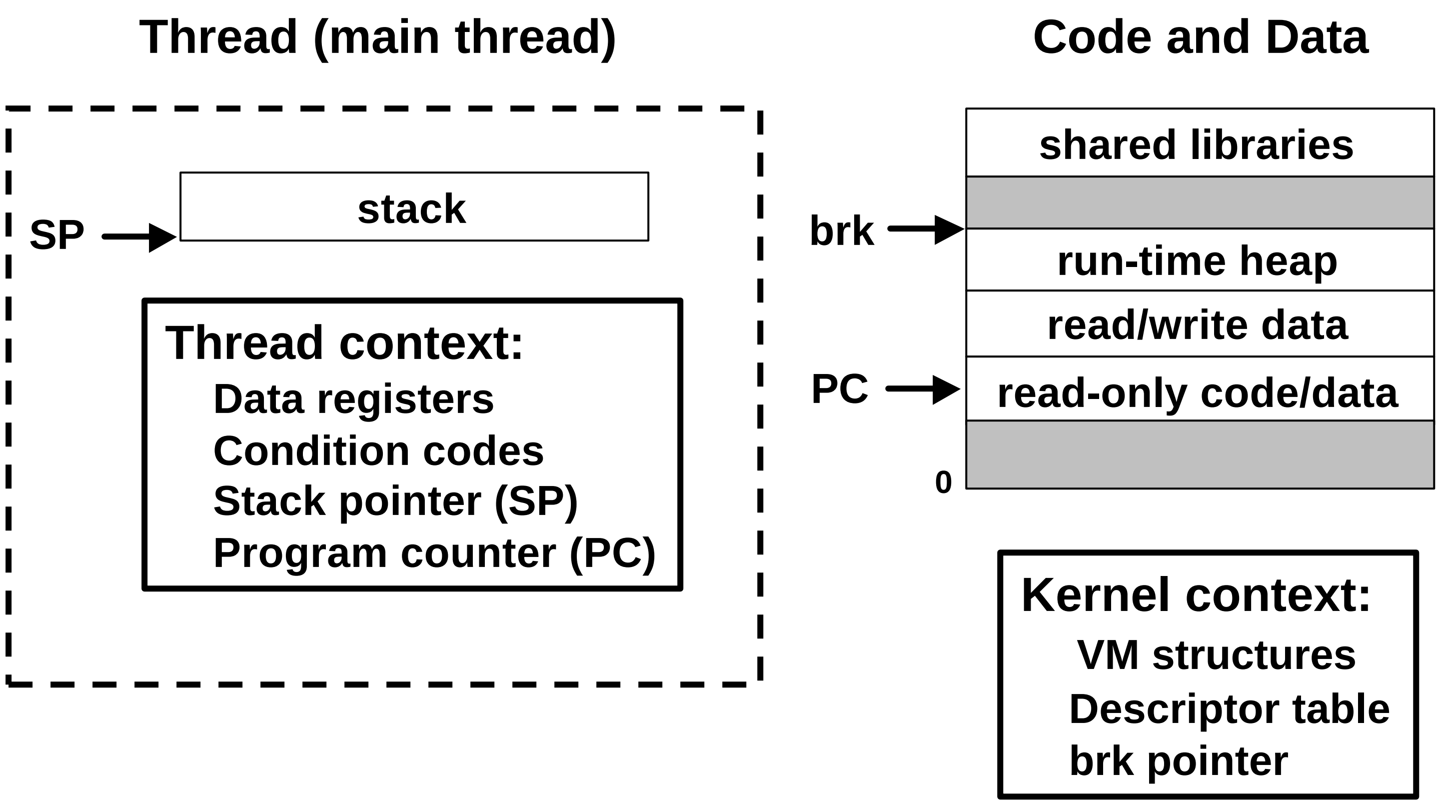
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



Alternate view of a process

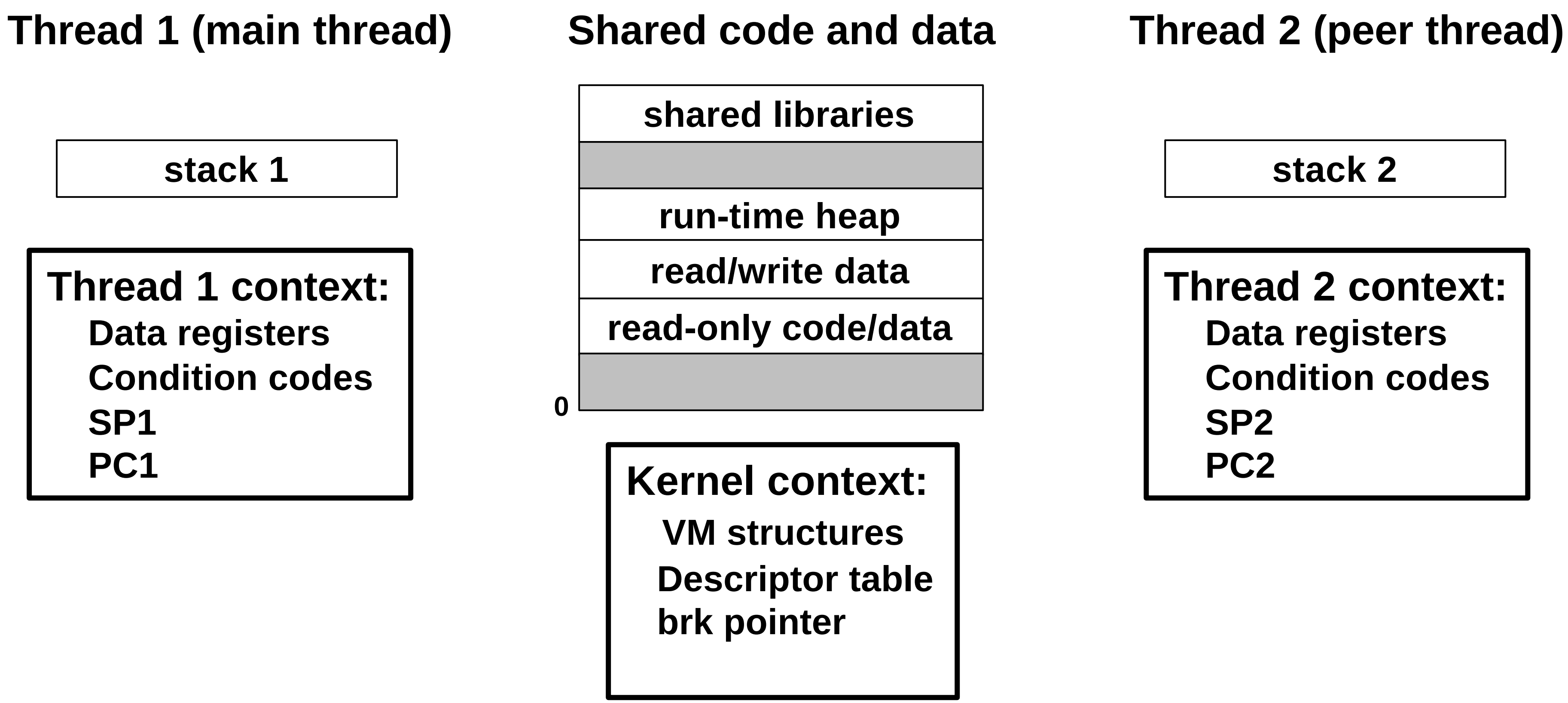
Process = thread + code, data, and kernel context



A process with multiple threads

Multiple threads can be associated with a process

- Each thread has its own logical control flow (sequence of PC values)
- Each thread shares the same code, data, and kernel context
- Each thread has its own thread id (TID)

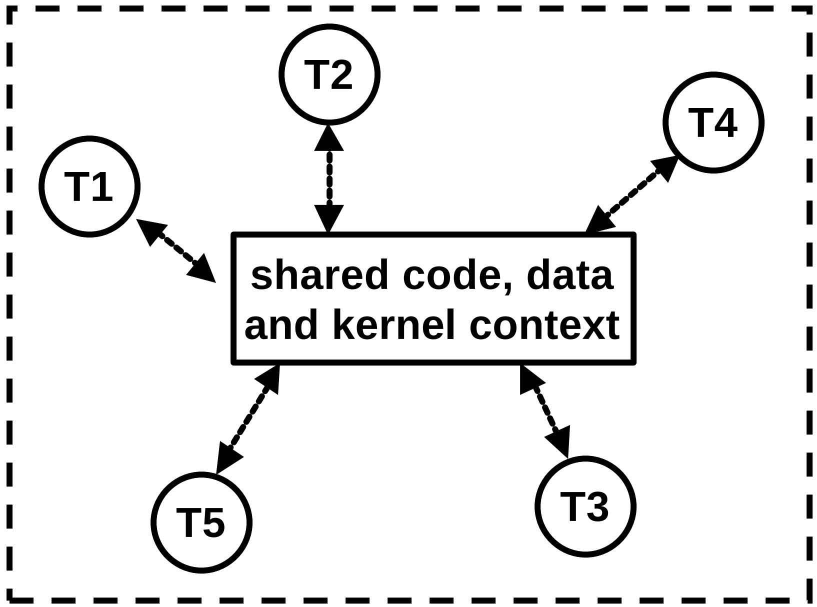


Logical view of threads

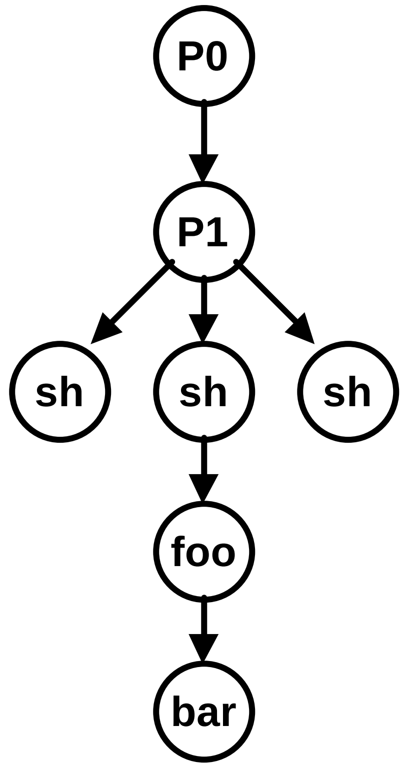
Threads associated with a process form a pool of peers.

- **Unlike processes which form a tree hierarchy**

Threads associated with process foo



Process hierarchy



User vs. Kernel Threads

1. **Kernel Threads**
   - **Independent of User processes (thread table in kernel space – not too large)**
   - **Ex. : Realization of asynchronous I/O**
   - **Efficient and cheap in use (kernel stack, register save area)**
   - **But, all calls that might block a thread are implemented as *sys-calls,* higher cost than a call to a run-time procedure -> use of recycled threads**
2. **Kernel-supported user threads (lightweight process, LWPs)**
   - **User threads are mapped to kernel threads (n:1)**
   - **Kernel is aware of only kernel threads**
   - **Threads share data**
   - **Truely parallel execution possible**
3. **User Threads**
   - **Thread infrastructure realized in user-space**
   - **Kernel is not aware of threads (sees only single-threaded processes)**
   - **Can be implemented on an OS that does not support threads**
   - **Use of customized scheduling algorithm**
   - **How are blocking calls are implemented (e.g., read from keyboard) so that not the whole process is blocked -> tell the OS in advance about the call**

LINUX kernel threads

Under Linux, there are three kinds of tasks:

- **the idle thread(s),**
- **kernel threads,**
- **user processes.**

- **The idle thread is created at compile time for the first CPU;**
- **Kernel threads are created using *kernel_thread()*. Kernel threads usually have no user address space, i.e. p->mm = NULL**
- **Kernel threads can always access kernel address space directly. They are allocated PID numbers in the low range.**

  They cannot be pre-empted by the scheduler.
- **User processes are created by means of *clone(2)* or *fork(2)* system calls, both of which internally invoke kernel/fork.c:do_fork().**

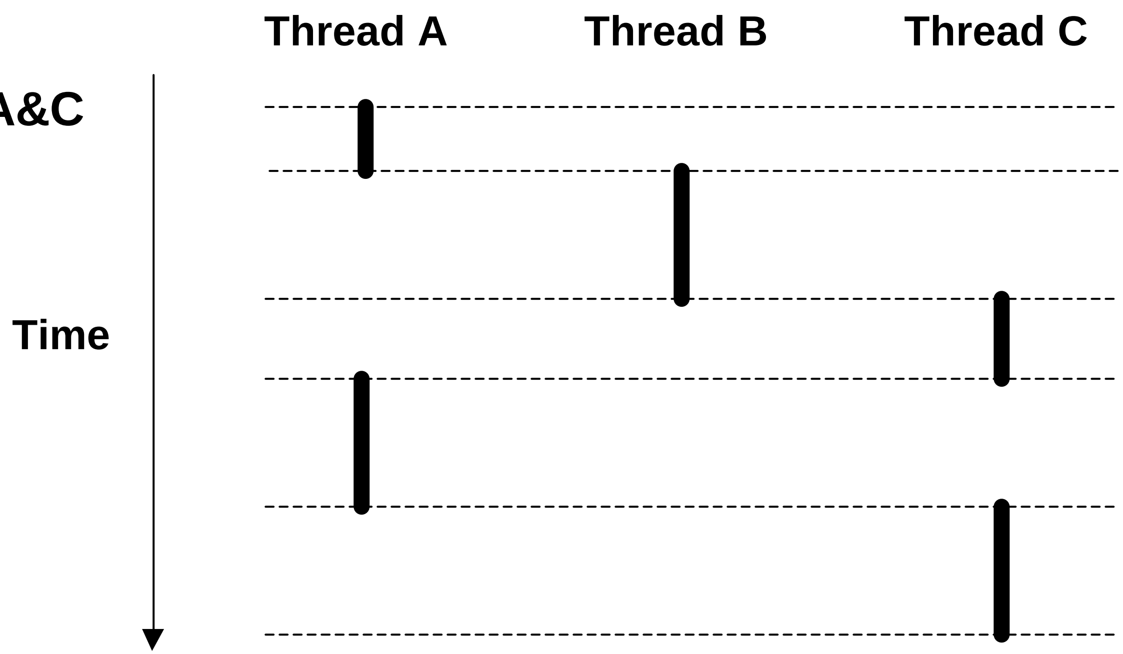
Concurrent thread execution

Two threads *run concurrently* (*are concurrent)* if their logical flows overlap in time.

Otherwise, they are *sequential.*

Examples:

- **Concurrent: A & B, A&C**
- **Sequential: B & C**



Threads vs. processes

How threads and processes are similar

- **Each has its own logical control flow.**
- **Each can run concurrently.**
- **Each is context switched.**

How threads and processes are different

- **Threads share code and data, processes (typically) do not.**
- **Threads are somewhat less expensive than processes.**
  - process control (creating and reaping) is twice as expensive as thread control.
  - Linux/Pentium III numbers:
    - » 20K cycles to create and reap a process.
    - » 10K cycles to create and reap a thread.

Posix threads (Pthreads) interface

Pthreads: **Standard interface for ~60 functions that manipulate threads from C programs.**

- **Creating and joining threads.**
  - **`pthread_create`**
  - **`pthread_join`**
- **Determining your thread ID**
  - **`pthread_self`**
- **Terminating threads**
  - **`pthread_cancel`**
  - **`pthread_exit`**
  - **`exit` [terminates all threads] , `ret` [terminates current thread]**
- **Synchronizing access to shared variables**
  - Specified later

The Pthreads "hello, world" program

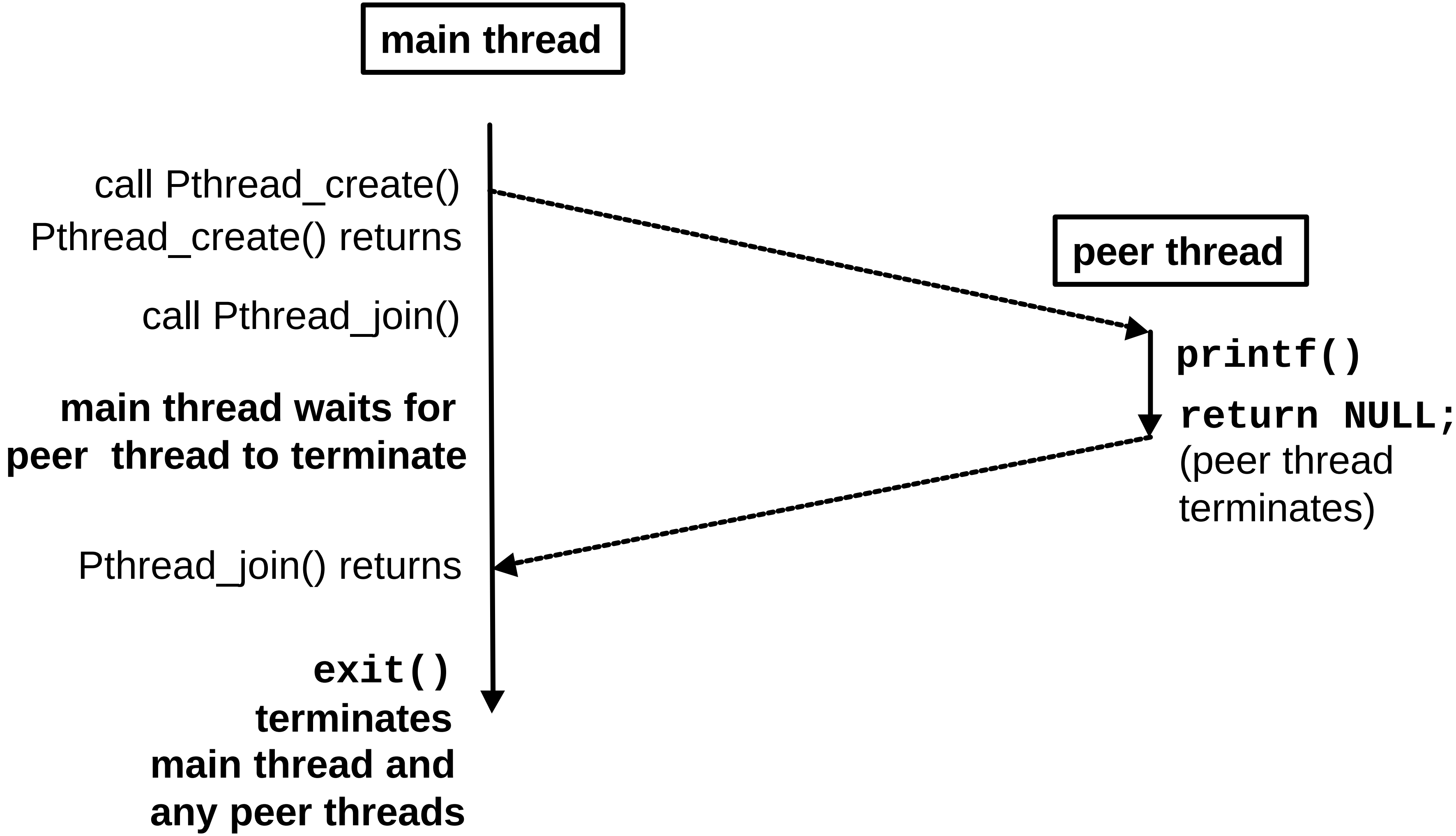
```c
/*
 * hello.c - Pthreads "hello, world" program
 gcc –o hello hello.c -lphtread*/
#include <pthread.h>

void *thread(void *vargp);

int main() {
  pthread_t tid;

  pthread_create(&tid, NULL, thread, NULL);
  pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Execution of “hello, world”



Shared variables in threaded C programs

Question: **Which variables in a threaded C program are *shared variables*?**

- **The answer is not as simple as "global variables are shared" and "stack variables are private".**

Requires answers to the following questions:

- **What is the memory model for threads?**
- **How are variables mapped to memory instances?**
- **How many threads reference each of these instances?**

Threads memory model

Conceptual model:

- **Each thread runs in the context of a process.**
- **Each thread has its own separate *thread context.***
  - Thread ID, stack, stack pointer, program counter, condition codes, and general purpose registers.
- **All threads share the remaining process context.**
  - Code, data, heap, and shared library segments of the process virtual address space.
  - Open files and installed handlers

Operationally, this model is not strictly enforced:

- **While register values are truly separate and protected....**
- **Any thread can read and write the stack of any other thread.**

Mismatch between the conceptual and operation model is a source of confusion and errors.

Example of threads accessing another thread's stack

```
...
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
}
```

Peer threads access main thread's stack indirectly through global ptr variable

Mapping variables to memory instances

Global var: **1 instance (`ptr` [data])**

Local automatic vars: **1 instance (`i.m, msgs.m`)**

```
#include <pthread.h>
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

Local automatic var: **2 instances (**
  `myid.p0` [peer thread 0's stack],
  `myid.p1` [peer thread 1's stack]
)

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
}
```

Local static var: **1 instance (`cnt` [data])**

Shared variable analysis

Which variables are shared?

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

Answer: A variable x is shared iff multiple threads reference at least one instance of x. Thus:

- **`ptr`, `cnt`, and `msgs` are shared.**
- **`i` and `myid` are NOT shared.**

Synchronizing with semaphores

Basic Tool for many Synchronization Problems:

Dijkstra's P and V operations on *semaphores.*

- ***semaphore:*** **non-negative integer synchronization variable.**
  - P(s): [`while (s == 0) wait(); s--;`]
    - » Dutch for "Proberen" (test)
  - V(s): [`s++;`]
    - » Dutch for "Verhogen" (increment)
- **OS guarantees that operations between brackets [ ] are executed indivisibly.**
  - **Only one P or V operation at a time can modify s.**
  - **When `while` loop in P terminates, only that P can decrement s.**

Semaphore invariant: *(s >= 0)*

POSIX semaphores (not System V)

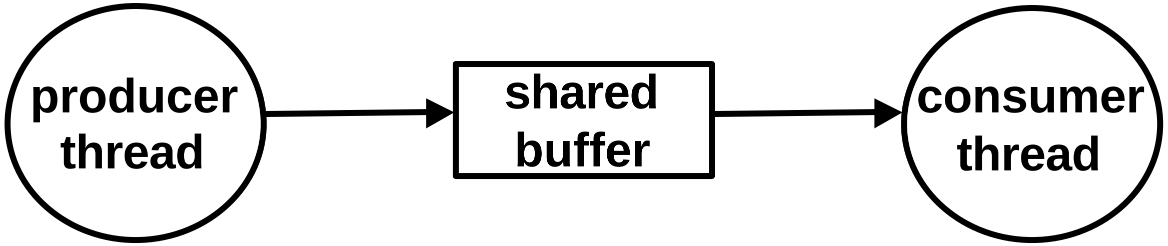
```
/* initialize semaphore sem to value */
/* pshared=0 if thread (for POSIX threads, pshared=1 if process */
/* however pshared=1 Not implemented for LINUX see man sem_init*/

void sem_init(sem_t *sem, int pshared, unsigned int value) {
  if (sem_init(sem, pshared, value) < 0)
    unix_error("Sem_init");
}

/* P operation on semaphore sem */
void P(sem_t *sem) {
  if (sem_wait(sem))
    unix_error("P");
}

/* V operation on semaphore sem */
void V(sem_t *sem) {
  if (sem_post(sem))
    unix_error("V");
}
```

Sharing with POSIX semaphores

```
/* goodcnt.c - properly sync'd
counter program */

#include <pthread.h>
#include <semaphore.h>

#define NITERS 10000000

unsigned int cnt; /* counter */
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;

    sem_init(&sem, 0, 1);

    /* create 2 threads and wait */
    ...

exit(0);
}
```

```
/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```

Signaling with semaphores



Common synchronization pattern:

- **Producer waits for slot, inserts item in buffer, and "*signals*" consumer.**
- **Consumer waits for item, removes it from buffer, and "signals" producer.**
  - "signals" in this context has nothing to do with Unix signals

Examples

- **Multimedia processing:**
  - Producer creates MPEG video frames, consumer renders the frames
- **Event-driven graphical user interfaces**
  - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer.
  - Consumer retrieves events from buffer and paints the display.

Producer-consumer

```
…
#define BUFFERSIZE  10

/* 3 threads, one producer,
2 consumers */
pthread_t producer_thread;
pthread_t consumer_thread1,
    consumer_thread2;

/* A protected buffer,
protected with 3 semaphores */

struct prot_buffer {
  sem_t sem_read;
  sem_t sem_write;
  int readpos, writepos;
  char buffer[BUFFERSIZE];
} b;

sem_t mutex;
```

```
int main() {
char str1[] = "consumer1";
char str2[] = "consumer2";

/* There is nothing to read  */
sem_init(&(b.sem_read), 0, 0);
/* But much space for writing */
sem_init(&(b.sem_write),0,BUFFERSIZE-1);
sem_init(&mutex, 0, 1);

pthread_create(&producer_thread, NULL,
(void *) &producer, NULL);
pthread_create(&consumer_thread1, NULL,
(void *) &consumer, (void *) str1);
pthread_create(&consumer_thread2, NULL,
(void *) &consumer, (void *) str2);

 pthread_detach(consumer_thread1);
 pthread_detach(consumer_thread2);
 pthread_join(producer_thread, NULL);
}
```

Producer-consumer (cont)

```
void producer() {
char c;
for(;;) {
 sem_wait(&(b.sem_write));
 sem_wait(&mutex);
 while( '\n' != (c = getchar()))
  {
   if( 'E' == c) {
    pthread_exit(NULL);
   }
   b.buffer[b.writepos] = c;
   b.writepos++;
   if(b.writepos >= BUFFERSIZE)
     b.writepos = 0;
   sem_post(&(b.sem_read));
   sem_post(&mutex);
  }
 }
}
```

```
void consumer(void * str) {
char c;
for(;;) {
  sem_wait(&(b.sem_read));
  sem_wait(&mutex);
  c = b.buffer[b.readpos];
  b.readpos++;

  if(b.readpos >= BUFFERSIZE)
   b.readpos = 0;

  printf(" %s: ", (char *) str);
  putchar(c); putchar('\n');
  sem_post(&(b.sem_write));
  sem_post(&mutex);
 }
}
```

Limitations of semaphores

Semaphores are sound and fundamental, but they have limitations.

- **Difficult to broadcast a signal to a group of threads.**
  - e.g., *barrier synchronization:* no thread returns from the barrier function until every other thread has called the barrier function.
- **Difficult to program (exchange order of the semaphores in the last example !)**
- **Impossible to do timeout waiting.**
  - e.g., wait for at most 1 second for a condition to become true.

Alternatively, we may use Pthreads *safe sharing* and *condition variables.*

Synchronizing with safe sharing and condition variables

Pthreads interface provides two different mechanisms for these functions:

- **Safe sharing: operations on mutex variables**
- **Conditional Signaling: operations on condition variables**

DIFFERENT FROM Hoare’s MONITOR ADT

MONITOR = mutual exclusion on all procedures of the monitor

Basic operations on mutex variables

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr)
```

Initializes a mutex variable (`mutex`) with some attributes (`attr`).

- **attributes are usually NULL.**
- **like initializing a mutex semaphore to 1.**

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Indivisibly waits for `mutex` to be unlocked and then locks it.

- **like `P(mutex)`**

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Unlocks `mutex`.

- **like `V(mutex)`**

Basic operations on cond variables

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *attr)
```

Initializes a condition variable cond with some attributes (`attr`).

- **attributes are usually NULL.**

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Indivisibly waits for the signaling on the condition variable cond together with a locked `mutex`. (it releases `mutex`).

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Signaling (wake-up) all threads waiting for on the condition variable cond. (use of pthread_cond_signal to wake up one).

Producer-consumer with signaling

```
#define non_empty 0
#define non_full  1

/* 3 Threads:one is the producer, other are consumers */
pthread_t producer_thread;
pthread_t consumer_thread1, consumer_thread2;

/* State of the buffer is full or empty */
pthread_cond_t buffer_full;
pthread_cond_t buffer_empty;

/* Only a thread holding this mutex is allowed to access the
buffer */
pthread_mutex_t mutex;

/* Who is allowed to access the buffer, the producer or the
consumers? */
int state;

/* This buffer is shared between the threads (1-slot buffer)
*/
char *buffer;
```

Producer-consumer with signaling (cont)

```
void main() {
char str1[] = "consumer1"; char str2[] = "consumer2";

  /* The thread must start working */
  state = non_full;
  buffer = (char *) malloc(1000 * sizeof(char));

  /* Initialize the 2 States (conditions) */
  pthread_cond_init(&buffer_full, NULL);
  pthread_cond_init(&buffer_empty, NULL);
  pthread_mutex_init(&mutex, NULL);

  /* And create the 3 threads */
  pthread_create(&producer_thread, NULL, (void *)
     &producer, NULL);
  pthread_create(&consumer_thread1, NULL, (void *)
     &consumer, (void *) str1);
  pthread_create(&consumer_thread2, NULL, (void *)
     &consumer, (void *) str2);
...
}
```

Producer-consumer (cont)

```
void producer() {
  for(;;) {
    pthread_mutex_lock(&mutex);
    // !=non_full = full = non_empty
    while(state != non_full)
     pthread_cond_wait(
       &buffer_empty,&mutex);
    // Rewrite buffer
    buffer = gets(buffer);

    if(0==strcmp(buffer, "end")) {
      pthread_mutex_unlock(&mutex);
      pthread_exit(NULL);
    }
    /* The buffer is full now,
    so tell the consumers */
    state = non_empty;
    pthread_mutex_unlock(&mutex);
    pthread_cond_broadcast(
     &buffer_full);
  }
}
```

```
void consumer(void * str) {

  for(;;) {
    pthread_mutex_lock(&mutex);

    while(state != non_empty)
     pthread_cond_wait
     (&buffer_full, &mutex);

    printf("  %s = %s\n",
    (char *) str, buffer);

    state = non_full;

    pthread_mutex_unlock
    (&mutex);
    pthread_cond_signal
    (&buffer_empty);
  }
}
```

Thread-safe functions

Functions called from a thread must be *thread-safe.*

We identify four (non-disjoint) classes of thread-unsafe functions:

- **Class 1: Failing to protect shared variables.**
- **Class 2: Relying on persistent state across invocations.**
- **Class 3: Returning a pointer to a static variable.**
- **Class 4: Calling thread-unsafe functions.**

Thread-unsafe functions

Class 1: Failing to protect shared variables.

- **Fix: use Pthreads lock/unlock functions or P/V operations.**
- **Issue: synchronization operations will slow down code.**
- **Example: `goodcnt.c`**

Thread-safe functions (cont)

Class 2: Relying on persistent state across multiple function invocations.

- **The `my_read()` function called by `readline()` buffers input in a `static` array.**

```
ssize_t
my_read(int fd, char *ptr)
{
    static int read_cnt = 0;
    static char *read_ptr,
    static char *read_buf[MAXLINE];
    ...
}
```

- **Fix: Rewrite function so that caller passes in all necessary state.**

```
ssize_t
my_read_r(Rline *rptr, char *ptr)
{
   ...
}
```

Thread-safe functions (cont)

Class 3: Returning a pointer to a `static` variable.

```
struct hostent
*gethostbyname(char name)
{
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

- **Fixes:**
  - **1. Rewrite so caller passes pointer to `struct`.**
    - » **Issue: Requires changes in caller and callee.**

```
hostp = Malloc(...));
gethostbyname1_r(name, hostp);
```

  - **2. "Lock-and-copy"**
    - » **Issue: Requires only simple changes in caller (and none in callee)**
    - » **However, caller must free memory.**

```
struct hostent
*gethostbyname_ts(char *name)
{
  struct hostent *q = Malloc(...);
  Pthread_mutex_lock(&mutex);
  p = gethostbyname(name);
  *q = *p;
  Pthread_mutex_unlock(&mutex);
  return q;
}
```

Thread-safe functions

Class 4: Calling thread-unsafe functions.

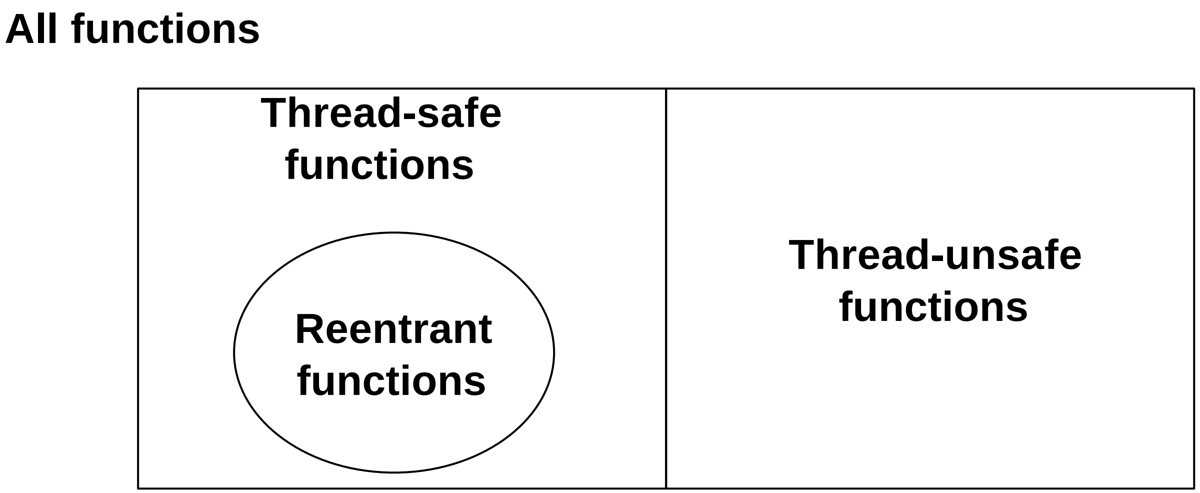
- **Calling one thread-unsafe function makes an entire function thread-unsafe.**
  - **Since `readline()` calls the thread-unsafe `my_read()` function, it is also thread_unsafe.**

- **Fix: Modify the function so it calls only thread-safe functions**
  - **Example: `readline_r()` is a thread-safe version of `readline()` that calls the thread-safe `my_read_r()` function.**

Reentrant functions

A function is *reentrant* iff it accesses NO shared variables when called from multiple threads.

- **Reentrant functions are a proper subset of the set of thread-safe functions.**



- **NOTE: The fixes to Class 2 and 3 thread-unsafe functions require modifying the function to make it reentrant.**

Thread-safe library functions

All functions in the Standard C Library (at the back of your K&R text) are thread-safe.

Most Unix system calls are thread-safe, with a few exceptions:

Thread-unsafe function	Class	Reentrant version
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(none)
localtime	3	localtime_r
rand	2	rand_r

Since linux libc version 6.0 all C Libraries are thread safe

Threads summary

Threads provide another mechanism for writing concurrent programs.

Threads are growing in popularity

- **Somewhat cheaper than processes.**
- **Easy to share data between threads.**

However, the ease of sharing has a cost:

- **Easy to introduce subtle synchronization errors (-> Solution in Distributed Systems I next summer !!!)**
- **Tread carefully with threads!**

For more info:

- **D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997.**