CS 105

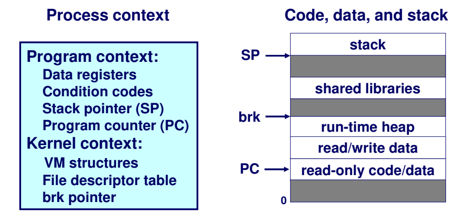
"Tour of the Black Holes of Computing!"

Programming with Threads

Topics
- Threads
- Shared variables
- The need for synchronization
- Synchronizing with semaphores
- Thread safety and reentrancy
- Races and deadlocks

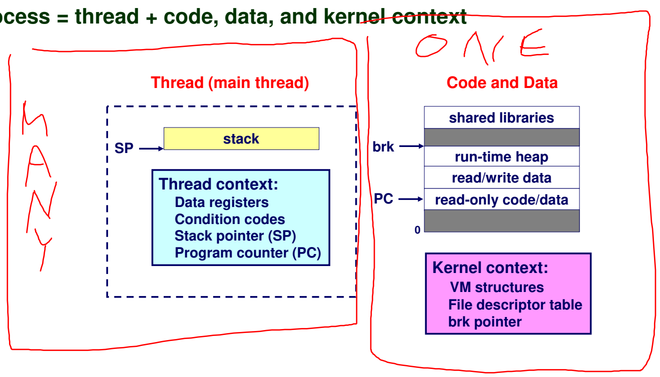
Traditional View of a Process

Process = process context + code, data, and stack



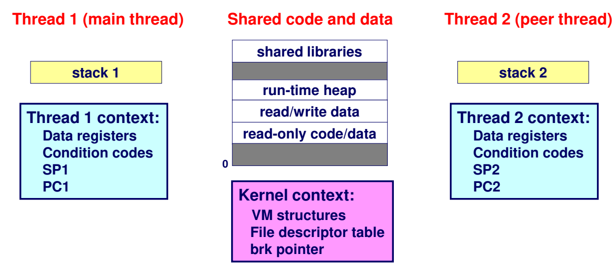
Alternate View of a Process

Process = thread + code, data, and kernel context



A Process With Multiple Threads

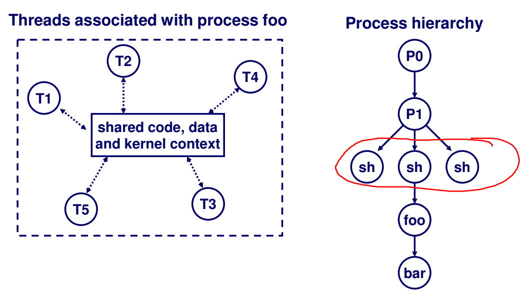
Multiple threads can be associated with a process
- Each thread has its *own* logical control flow (sequence of PC values)
- Each thread *shares* the same code, data, and kernel context
- Each thread has its own thread id (TID)



Logical View of Threads

Threads associated with a process form pool of peers

- Unlike processes, which form tree hierarchy



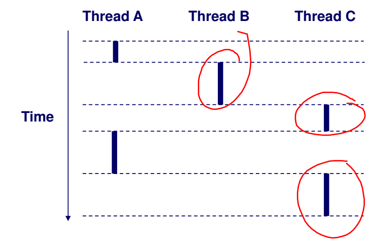
Concurrent Thread Execution

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

Otherwise, they are sequential (same rule as for processes)

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 (maybe on different cores)
- Each is context-switched

How threads and processes are different

- Threads share code and data, processes (typically) do not
- Threads are somewhat cheaper than processes
  - Process control (creating and reaping) is roughly 5–8× as expensive as thread control
  - Linux numbers:
    - » ~160K, 280K, 530K cycles minimum to create and reap a process (three machines)
    - » ~19K, 34K, 100K cycles minimum to create and reap a thread

Posix Threads (Pthreads) Interface

Pthreads: Standard interface for ~60 (!) functions that manipulate threads from C programs

- Creating and reaping threads
  - `pthread_create, pthread_join`
- Determining your thread ID
  - `pthread_self`
- Terminating threads
  - `pthread_cancel, pthread_exit`
  - `exit` [terminates all threads], `return` [terminates current thread]
- Synchronizing access to shared variables
  - `pthread_mutex_init, pthread_mutex_[un]lock`
  - `pthread_cond_init, pthread_cond_[timed]wait`

The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *howdy(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, howdy, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

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

Thread ID

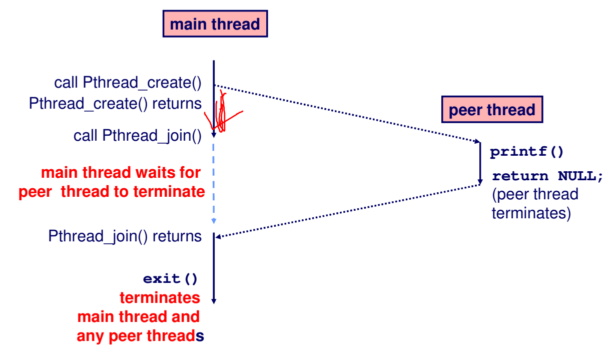
Thread attributes (usually NULL)

*Thread arguments (void *p)*

Thread routine

*Thread return value (void **p)*

Execution of Threaded "hello, world"



Pros and Cons of Thread-Based Designs

+ Threads take advantage of multicore/multi-CPU H/W
+ Easy to share data structures between threads
  - E.g., logging information, file cache
+ Threads are more efficient than processes

– Unintentional sharing can introduce subtle and hard-to-reproduce errors!
  - Ease of data sharing is greatest strength of threads, but also greatest weakness
  - Hard to know what's shared, what's private
  - Hard to detect errors by testing (low-probability failures)

Shared Variables in Threaded C Programs

Question: Which variables in a threaded C program are shared variables?
- Answer not as simple as "global variables are shared" and "stack variables are private"

Definition: **A variable x is *shared* if and only if multiple threads reference some instance of x.**

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 larger 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 remaining process context
  - Code, data, heap, and shared library segments of process virtual address space
  - Open files and installed handlers

Operationally, this model is not strictly enforced:

- Register values are truly separate and protected
- But any thread can read and write the stack of any other thread

Mismatch between conceptual and operational model is a source of confusion and errors

Example Program to Illustrate Sharing

```
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_join omitted
    Pthread_exit(NULL);
}
```

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

    printf("[%d]: %s (svar=%d)\n",
        myid, ptr[myid], ++svar);
    return 0;
}
```

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

Mapping Variable Instances to Memory

Global variables

- *Def:* Variable declared outside of a function
- Virtual memory contains exactly one instance of any global variable

Local variables

- *Def:* Variable declared inside function without `static` attribute
- Each thread stack frame contains one instance of each local variable

Local static variables

- *Def:* Variable declared inside function with the `static` attribute
- Virtual memory contains exactly one instance of any local static variable.

Mapping Vars to Memory Instances

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

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

```
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 svar = 0;

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

Local static var: 1 instance: `svar` [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
`svar`	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`, `svar`, and `msgs` are shared.
- `i` and `myid` are NOT shared.

Synchronizing Threads

Shared variables are handy...

...but introduce the possibility of nasty *synchronization* errors.

`badcnt.c`: An Improperly Synchronized Threaded Program

```
unsigned int cnt = 0; /* shared */

int main()
{
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL,
                   count, NULL);
    Pthread_create(&tid2, NULL,
                   count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt == (unsigned)NITERS*2)
        printf("OK cnt=%d\n",
               cnt);
    else
        printf("BOOM! cnt=%d\n",
               cnt);
    return 0;
}
```

```
/* thread routine */
void *count(void *arg)
{
    int i;
    for (i = 0; i < NITERS; i++)
        cnt++;
    return NULL;
}
```

```
linux> ./badcnt
BOOM! cnt=198841183

linux> ./badcnt
BOOM! cnt=198261801

linux> ./badcnt
BOOM! cnt=198269672
```

Zero?

`cnt` should be 200,000,000.
What went wrong?!

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (i = 0; i < NITERS; i++)
    cnt++;
```

Asm code for thread i

```
    movl $100000000, %edx
.L2:
    movl cnt(%rip), %eax
    addl $1, %eax
    movl %eax, cnt(%rip)
    subl $1, %edx
    jne  .L2
```

H_i : Head
L_i : Load `cnt`
U_i : Update `cnt`
S_i : Store `cnt`
T_i : Tail

critical

Concurrent Execution

Key idea: **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**

- I_i denotes that thread i executes instruction I
- $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

Thread 1 critical section

Thread 2 critical section

Concurrent Execution (cont)

Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

Concurrent Execution (cont)

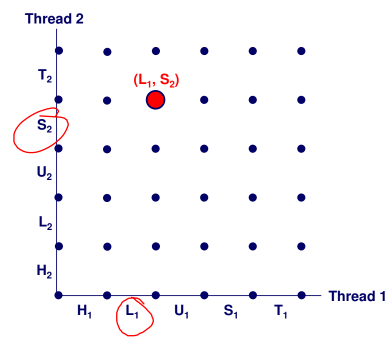
How about this ordering?

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1			0
1	L_1	0		
2	H_2			
2	L_2		0	
2	U_2		1	
2	S_2		1	1
1	U_1	1		
1	S_1	1		1
1	T_1			1
2	T_2			1

Oops!

We can analyze the behavior using a *progress graph*

Progress Graphs



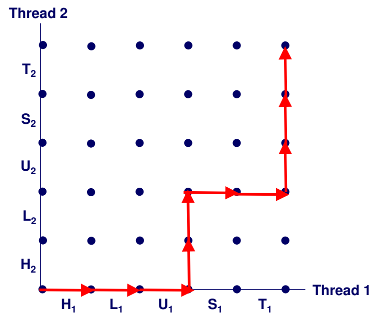
Progress graph **depicts discrete *execution state space* of concurrent threads**

Each axis corresponds to sequential order of instructions in a thread

Each point corresponds to a possible *execution state* ($Inst_1$, $Inst_2$)

E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2

Trajectories in Progress Graphs

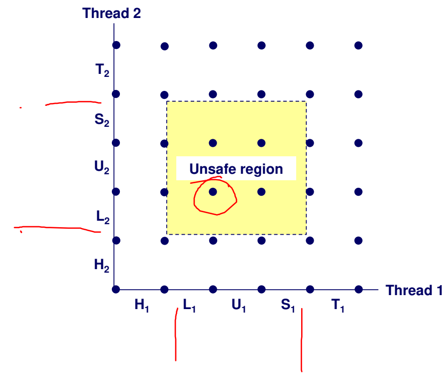


A *Trajectory* is sequence of legal state transitions that describes one possible concurrent execution of the threads

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions

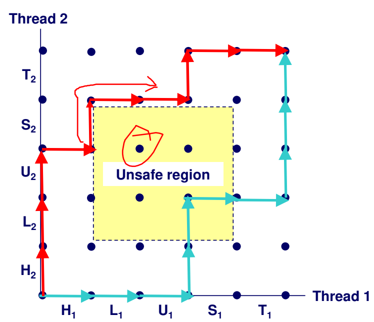


L, U, and S form a *critical section* with respect to the shared variable `cnt`

Instructions in critical sections (w.r.t. to some shared variable) should not be interleaved

Sets of states where such interleaving occurs form *unsafe regions*

Safe and Unsafe Trajectories



Def: A trajectory is *safe* iff it doesn't enter any part of an unsafe region

Claim: A trajectory is correct (w.r.t. `cnt`) iff it is safe

Races

Race happens when program correctness depends on one thread reaching point *x* before another thread reaches point *y*

```
void *thread(void *vargp);

/* a threaded program with a race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

Enforcing Mutual Exclusion

Question: **How can we guarantee a safe trajectory?**

Answer: We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.

- i.e., need to guarantee *mutually exclusive access* to critical regions

Classic solution:

- Semaphores (Edsger Dijkstra)

Other approaches

- Mutex and condition variables (Pthreads—ringbuf lab)
- Monitors (Java)

Pthread Mutexes

Part of Posix pthreads package

Only one thread can *hold* a given mutex at one time

- Mutex is associated with specific critical region or shared variable(s)
- Can use multiple mutexes to control different critical regions

pthread_mutex_lock:

- "Grabs" given mutex and returns
- If some other thread already has mutex, waits until it's free

pthread_mutex_unlock:

- "Releases" mutex and makes it available to other threads
- If any threads are waiting for mutex, wakes one up *at random* and gives mutex to it

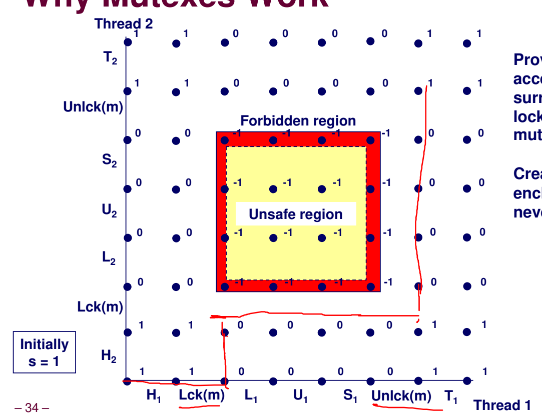
Sharing With Pthread Mutexes

```
/* goodcnt.c - properly sync'd
counter program */
#include <pthread.h>
#define NITERS 10000000

unsigned int cnt; /* counter */
pthread_mutex_t mutex; /* lock */

int main()
{
    pthread_t tid1, tid2;

    pthread_mutex_init(&mutex, NULL);
    /* create 2 threads and wait */
    ...

    if (cnt == (unsigned)NITERS*2)
        printf("OK cnt=%d\n", cnt);
    else
        printf("BOOM! cnt=%d\n", cnt);
    return 0;
}
```

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

    for (i = 0; i < NITERS; i++) {
        pthread_mutex_lock(&mutex);
        cnt++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

Why not just put lock/unlock around the whole loop?

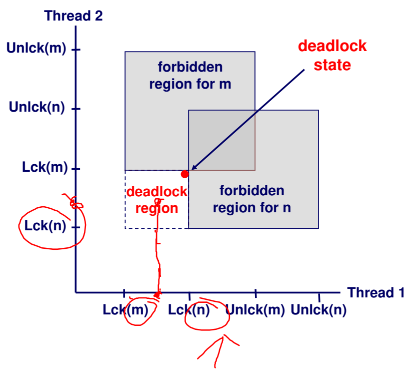
Why Mutexes Work



Provide mutually exclusive access to shared variable by surrounding critical section with lock and unlock operations on mutex named `m`

Creates *forbidden region* that encloses unsafe region and is never touched by any trajectory

Deadlock



Locking introduces potential for *deadlock:* waiting for a condition that will never be true.

Any trajectory that enters *deadlock region* will eventually reach *deadlock state*, waiting for either `m` or `n` to become nonzero.

Other trajectories luck out and skirt deadlock region.

Unfortunate fact: deadlock is often non-deterministic (thus hard to detect).

Synchronization With Pthread Conditions

Often need more than just mutual exclusion

- Thread B wants to wait for thread A to do something (X)
- Simple approach: mutex, "Did A do X?", release mutex, loop
  - Called "polling"
  - Wasteful of CPU
- Better approach: pthread conditions
  - B says "Wait for A to tell me about X"
  - A says "I did X"
  - B continues

Pthread condition variables

- One special variable per thing that can happen (e.g., "x_happened")
- Also need associated mutex
- Thread B must grab mutex (we'll see why in a moment), then calls pthread_cond_wait
  - Process of waiting *releases* mutex, pauses until X happens, then *re-grabs* mutex
- Thread A simply calls pthread_cond_signal
  - No need to hold mutex (but OK if you do)
  - IMPORTANT: If nobody is waiting, the signal is lost!

Pthread Waiting

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
...
int some_other_variable;
...

  pthread_mutex_lock(&mutex);
  while (!some condition on some variables)
    pthread_cond_wait(&condition, &mutex);
  do something with those variables--we hold the mutex!
  pthread_mutex_unlock(&mutex);
```

critical!

Decision to wait is based on "outside" variables (example coming)

- *Must check condition while holding a mutex*
- Decision to wait must be made *atomically*
  - Otherwise, could decide to wait, then other thread could signal before we *actually* wait
  - Remember signals are lost if nobody is waiting

Must re-check condition after being awoken

- Possible that another thread got mutex first and changed status

Pthread Synchronization (Sender)

```
int nsleeps = 0;

void* sender(void* data)
{
    int i;

    for (i = 0;  i < NPASSES;  i++) {
        sleep(1);
        printf("Sender slept %d time(s)\n", i + 1);
        pthread_mutex_lock(&mutex);
        ++nsleeps;
        pthread_mutex_unlock(&mutex);
        pthread_cond_signal(&slept);
    }
    return NULL;
}
```

Pthread Synchronization (Receiver)

```
void* receiver(void* data)
{
    int total_sleeps = 1;

    while (1) {
        pthread_mutex_lock(&mutex);
        if (total_sleeps >= NPASSES) {
            pthread_mutex_unlock(&mutex);
            printf("\tReceiver saw %d total sleeps\n", total_sleeps);
            return NULL;
        }
        while (nsleeps < total_sleeps) {
            pthread_cond_wait(&slept, &mutex);
        }
        pthread_mutex_unlock(&mutex);   awake
        printf("\tReceiver saw sleep number %d...", total_sleeps);
        ++total_sleeps;
        if (nsleeps < total_sleeps) {
            int sleep_time = random() % 4;
            printf("sleeping %d second(s)\n", sleep_time);
            sleep(sleep_time);
        }
        else
            printf("continuing\n");
    }
}
```


Thread Safety

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 pointer to static variable
- Class 4: Calling thread-unsafe functions

Thread-Unsafe Functions

Class 1: Failing to protect shared variables

- Fix: Use pthread mutex lock and unlock operations
- Issue: Synchronization operations will slow down code
- Example: `goodcnt.c`

Thread-Unsafe Functions (cont)

Class 2: Relying on persistent state across multiple function invocations

- Random number generator relies on static state
- Fix: Rewrite function so that caller passes in all necessary state

```
/* rand - return bad pseudo-random integer on 0..32767 */
static unsigned int next = 1;

int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

LUS

Thread-Unsafe Functions (cont)

Class 3: Returning pointer to `static` variable

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

Fixes:

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

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

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

Why outside the mutex?

```
struct hostent
*gethostbyname_ts(char *p)
{
  struct hostent *q = Malloc(...);
  pthread_mutex_lock(&mutex);
  p = gethostbyname(name);
  *q = *p;    /* copy */
  pthread_mutex_unlock(&mutex);
  return q;
}
```


Thread-Unsafe Functions

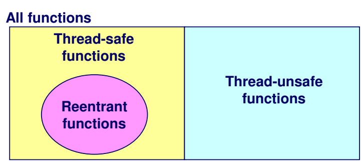
Class 4: Calling thread-unsafe functions

- Calling one thread-unsafe function makes an entire function thread-unsafe
- Fix: Modify the function so it calls only thread-safe functions

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 (only first fix for Class 3 is reentrant)

Thread-Safe Library Functions

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

- Examples: `malloc`, `free`, `printf`, `scanf`

All Unix system calls are thread-safe

Library calls that aren't thread-safe:

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`

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**
- **Tread carefully with threads!**

For more info:

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