# Multithreaded Programming

## Concepts and Practice

Bil Lewis

Lambda Computer Science
Bil@LambdaCS.com
www.LambdaCS.com

Phone/FAX: (650) 328-8952

# Objective

Explain the multithreading paradigm, and all aspects of how to use it in an application

- Cover basic MT concepts
- Review the positions of the major vendors
- Contrast POSIX, UI, Win32,and Java threads
- Explore (most) all issues related to MT
- Look at program and library design issues
- Look at thread-specific performance issues
- Understand MP hardware design
- Examine some POSIX/Java code examples

At the end of this seminar, you should be able to evaluate the appropriateness of threads to your application, and you should be able to use the documentation from your chosen vendor and start writing MT code.

# Table of Contents

# *Basic Concepts*

# A Thread is...

An independent flow of control in a program

A “virtual” CPU

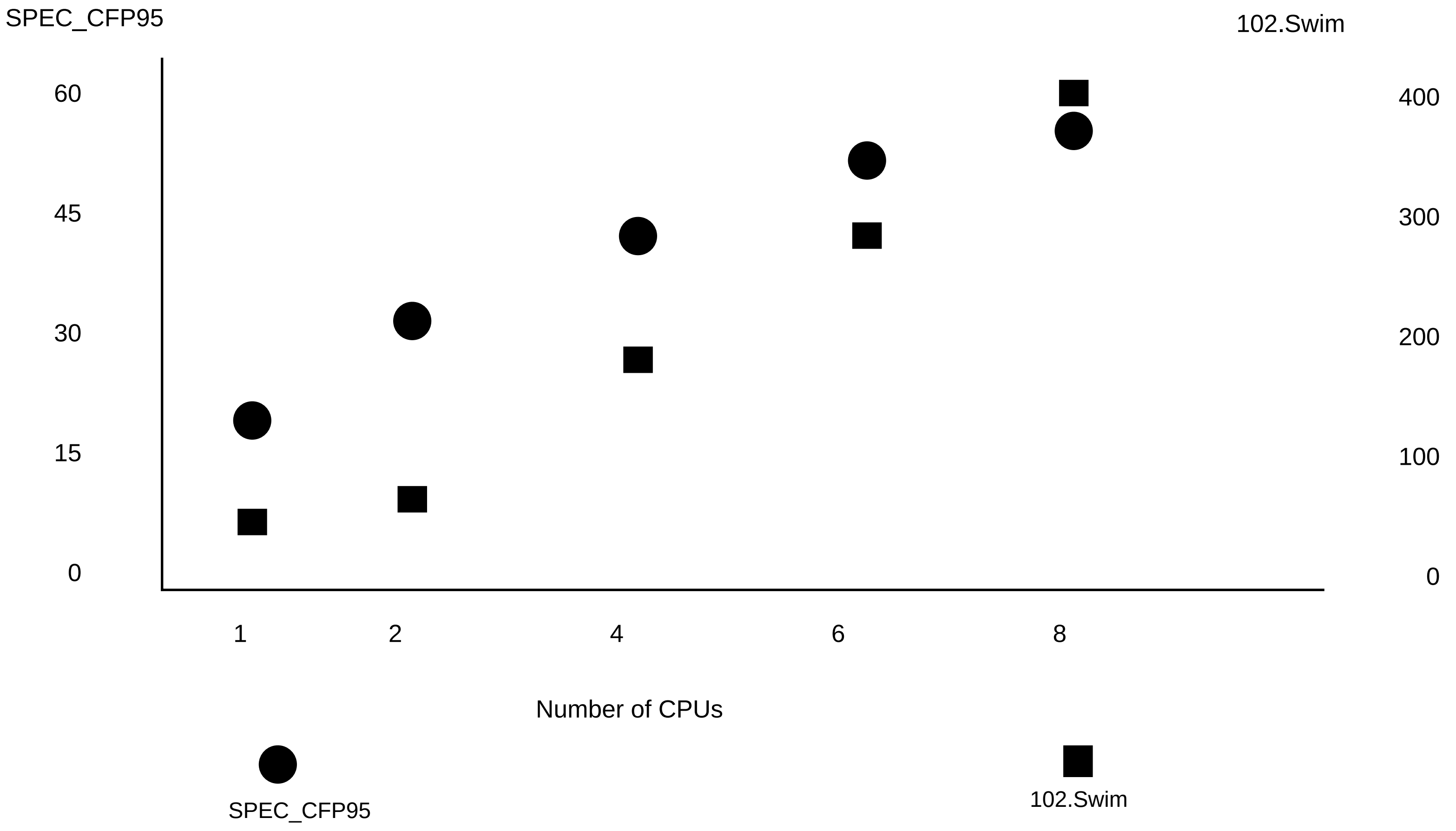
The little green man inside of your computer that reads your code and does the work!

# Background

- Experimental threads have been in labs for years
- Many different libraries were experimented with
- Currently there are four major “native” MT libraries in use:
    - POSIX / UNIX98
    - UI (aka “Solaris threads”)
    - OS/2
    - Win32
    - (Apple with Mach)
- POSIX ratified “Pthreads” in June, 1995
- All major UNIX vendors ship Pthreads

- Java Threads (which are usually built on the native threads)

# MP Hardware Performance



## Performance for Digital’s Alpha Servers (8400 5/625)

# The Value of MT

**Threads can:**

- **Simplify program structure**
- **Exploit parallelism**
- **Improve throughput**
- **Improve responsiveness**
- **Minimize system resource usage**
- **Simplify realtime applications**
- **Simplify signal handling**
- **Enable distributed objects**
- **Compile from a single source across platforms (POSIX, Java)**
- **Run from a single binary for any number of CPUs**

# What to Thread?

- **“Inherently” MT programs**
  **(Uniq/Delphax, programs whose structure is “MTish”)**

- **Parallelizable programs**
  **(NFS, numerics, new programs you need to be *FAST*)**

- **What not to thread**
  **(Icontool, 99.9% programs that are fine as they are)**

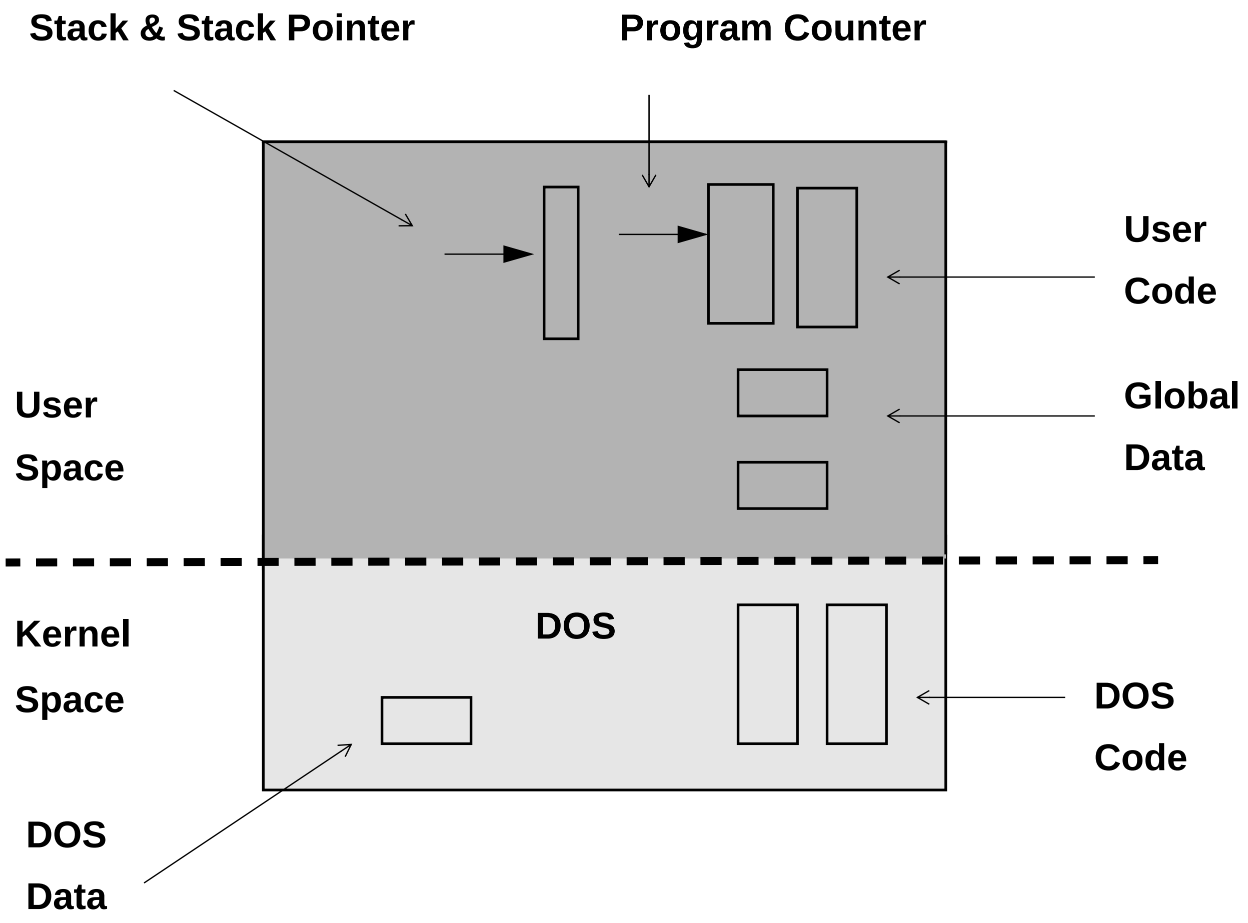
**Despite all of the good aspects of threads, one must remain aware that they *are* a different way of writing programs and that you will quickly find a whole new class of bugs in your programs. And some of those bugs will be *very* nasty.**

# “Proof”

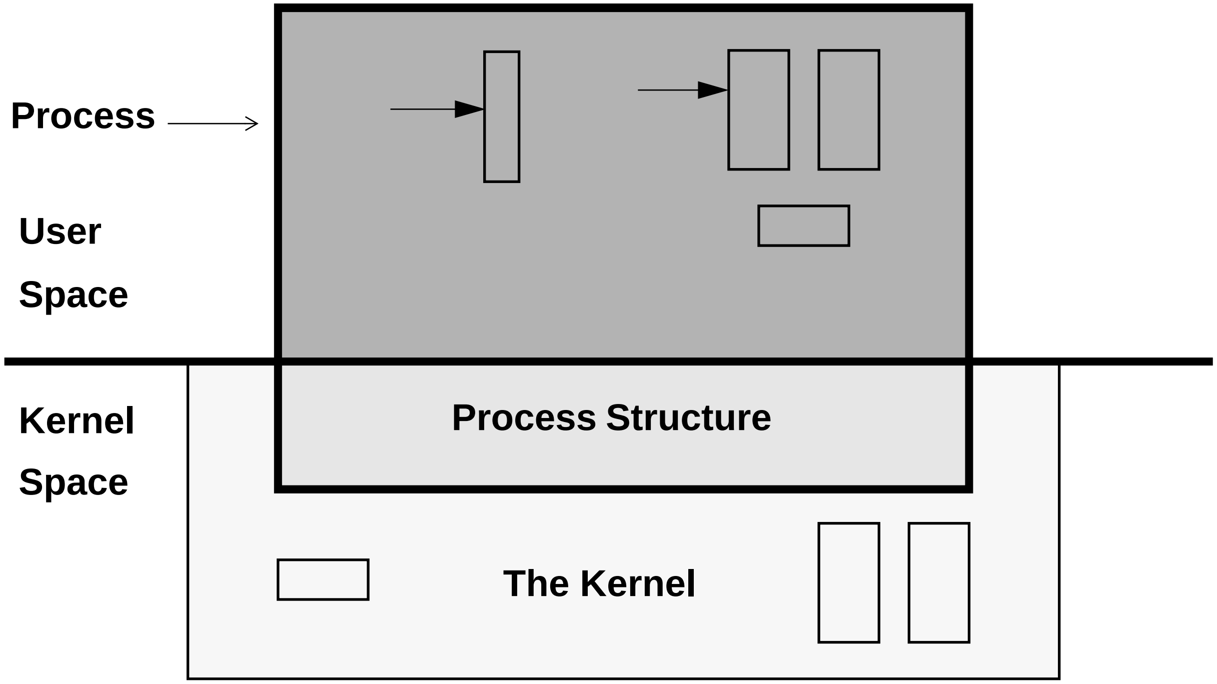
- Solaris 2, Digital UNIX, Unixware, AIX, IRIX, OS/2, NT
- NFS
- SPEC benchmarks
- DSS
- Agfa
- Uniq/Delphax
- MARC Analysis
- Photoshop
- ImagiNation Networks
- Scitex
- Sybase, Oracle, Informix, etc.
- Java
- Distributed Objects

# *Foundations*

# DOS - The Minimal OS

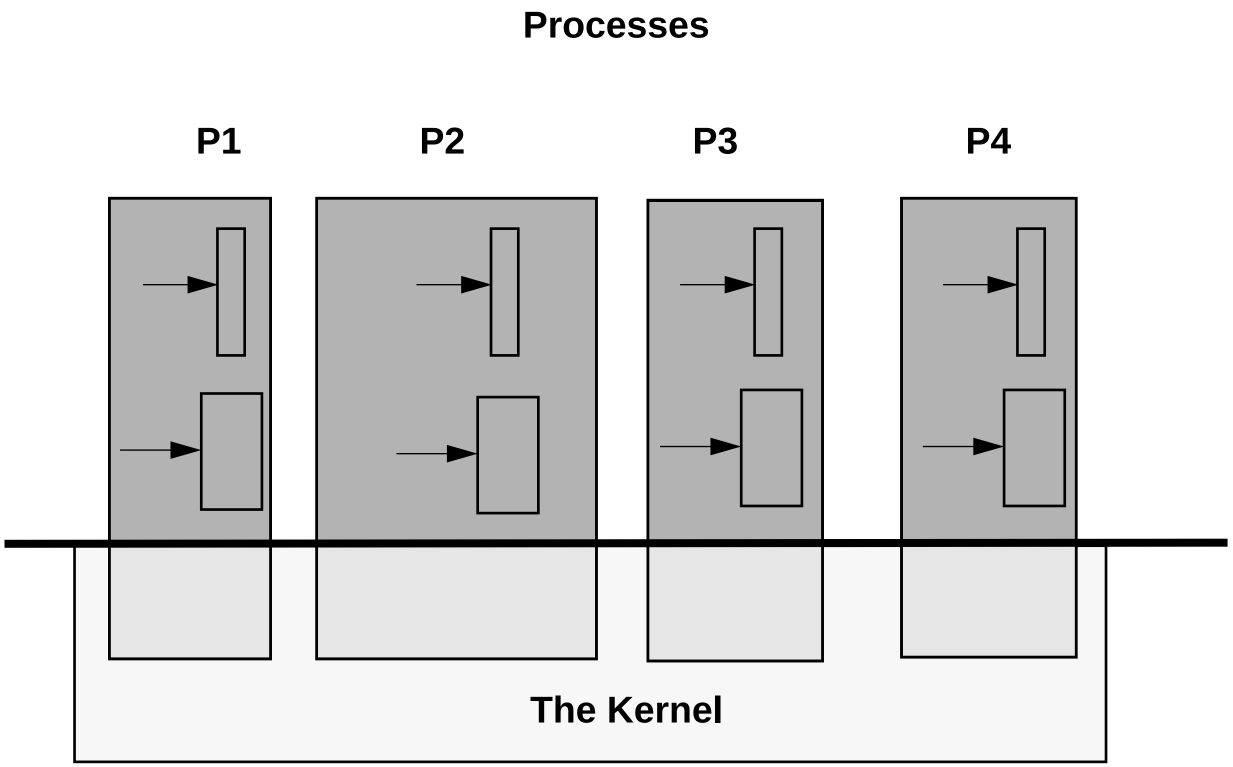


# Multitasking OSs



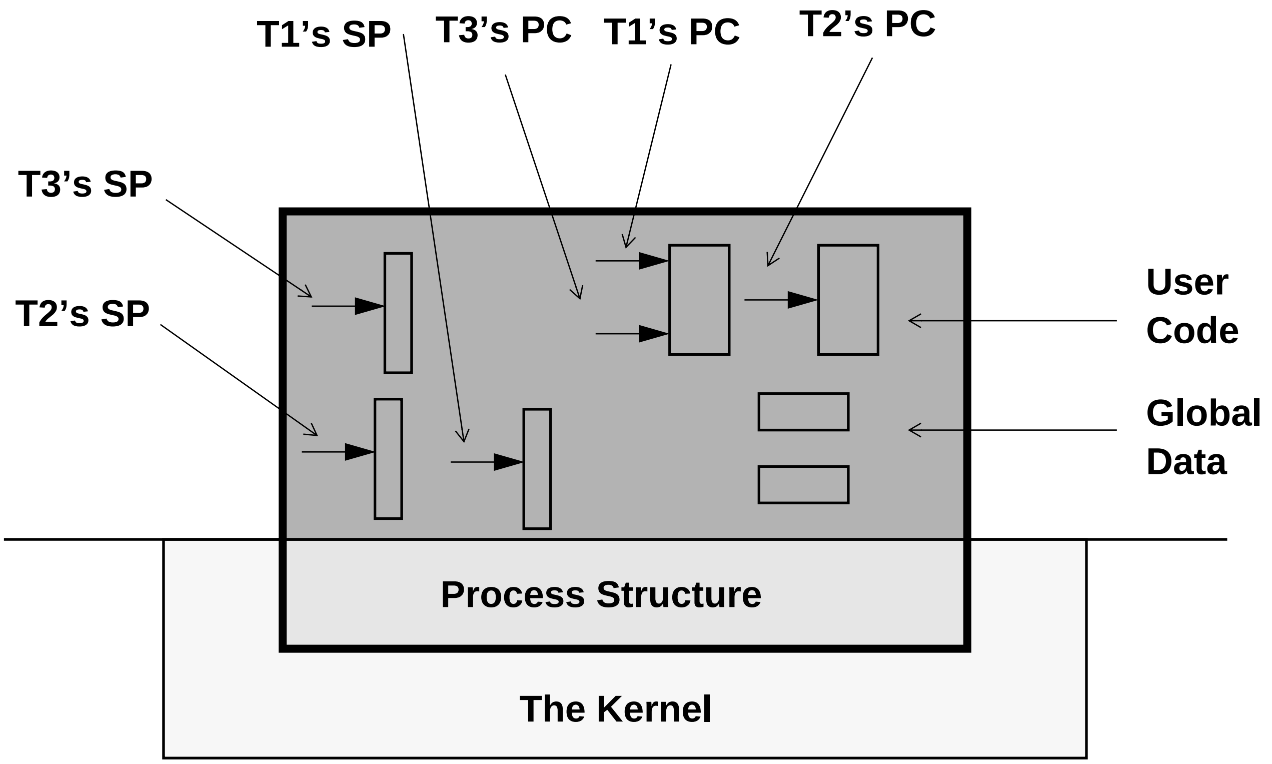
## (UNIX, VMS, MVS, NT, OS/2, etc.)

# Multitasking OSs



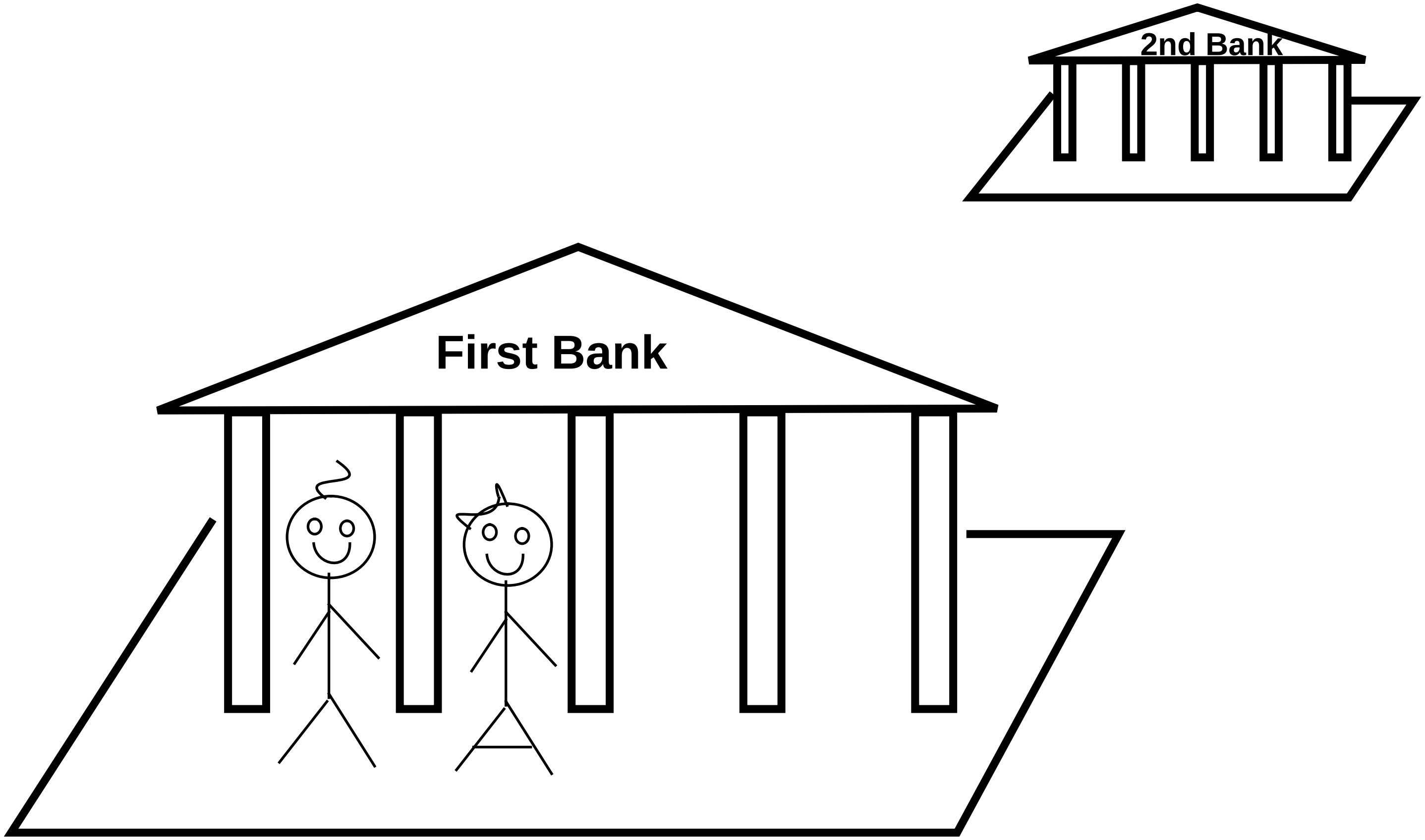
**(Each process is completely independent)**

# Multithreaded Process



**(Kernel state and address space are shared)**

# A Clever Analogy



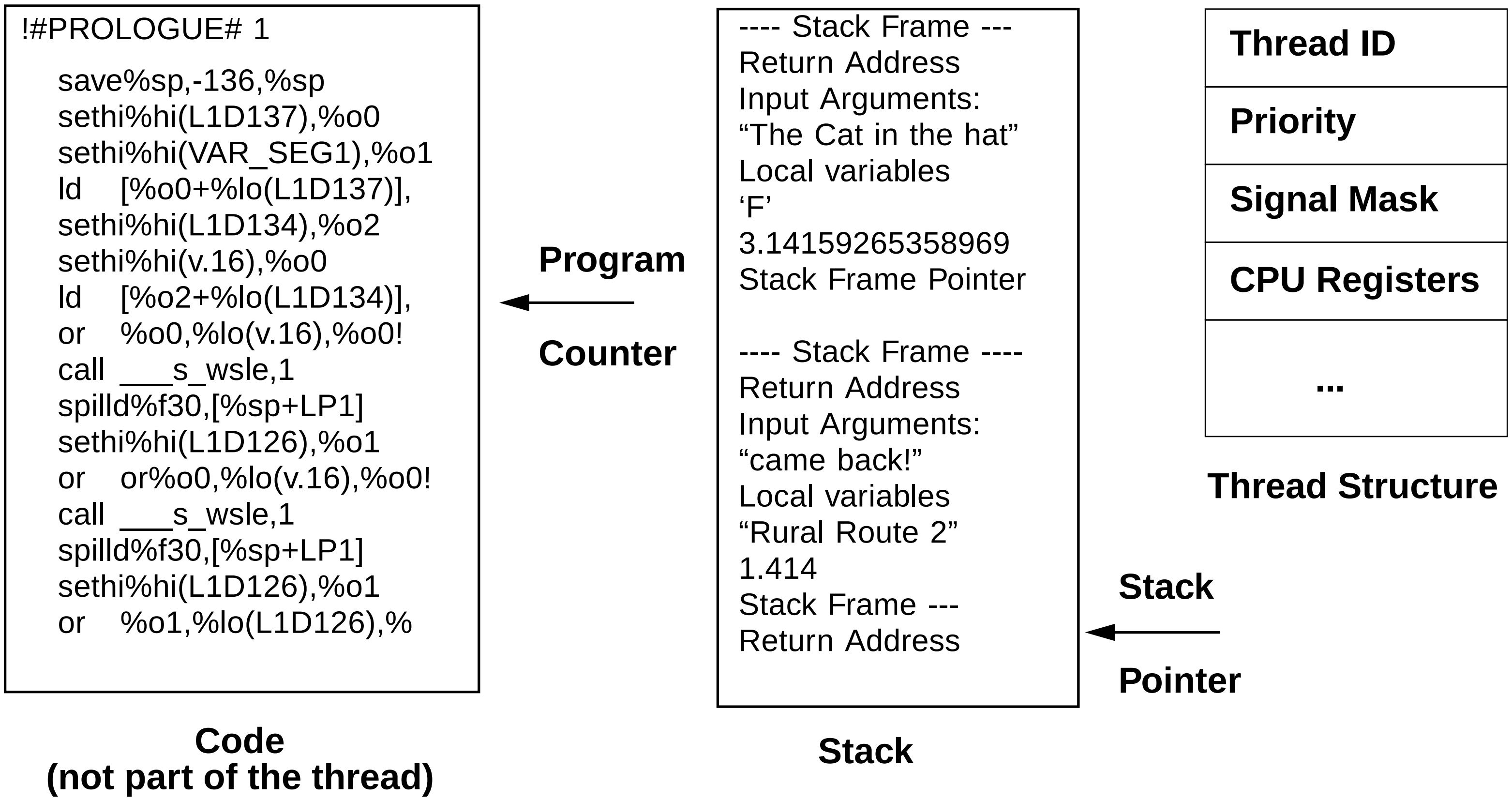
# Implementation vs. Specification

**A *specification* is a description of how things should work. Pthreads and the US constitution are specifications.**

**An *implementation* is an actual thing. The libpthread.so and the United States of America are implementations.**

**Learn from the implementation, write to the specification.**

# Thread Contents



```
!#PROLOGUE# 1
    save%sp,-136,%sp
    sethi%hi(L1D137),%o0
    sethi%hi(VAR_SEG1),%o1
    ld    [%o0+%lo(L1D137)],
    sethi%hi(L1D134),%o2
    sethi%hi(v.16),%o0
    ld    [%o2+%lo(L1D134)],
    or    %o0,%lo(v.16),%o0!
    call  ___s_wsle,1
    spilld%f30,[%sp+LP1]
    sethi%hi(L1D126),%o1
    or    or%o0,%lo(v.16),%o0!
    call  ___s_wsle,1
    spilld%f30,[%sp+LP1]
    sethi%hi(L1D126),%o1
    or    %o1,%lo(L1D126),%
```

**Code**
**(not part of the thread)**

**Program**

**Counter**

---- Stack Frame ---
Return Address
Input Arguments:
“The Cat in the hat”
Local variables
‘F’
3.14159265358969
Stack Frame Pointer

---- Stack Frame ----
Return Address
Input Arguments:
“came back!”
Local variables
“Rural Route 2”
1.414
Stack Frame ---
Return Address

**Stack**

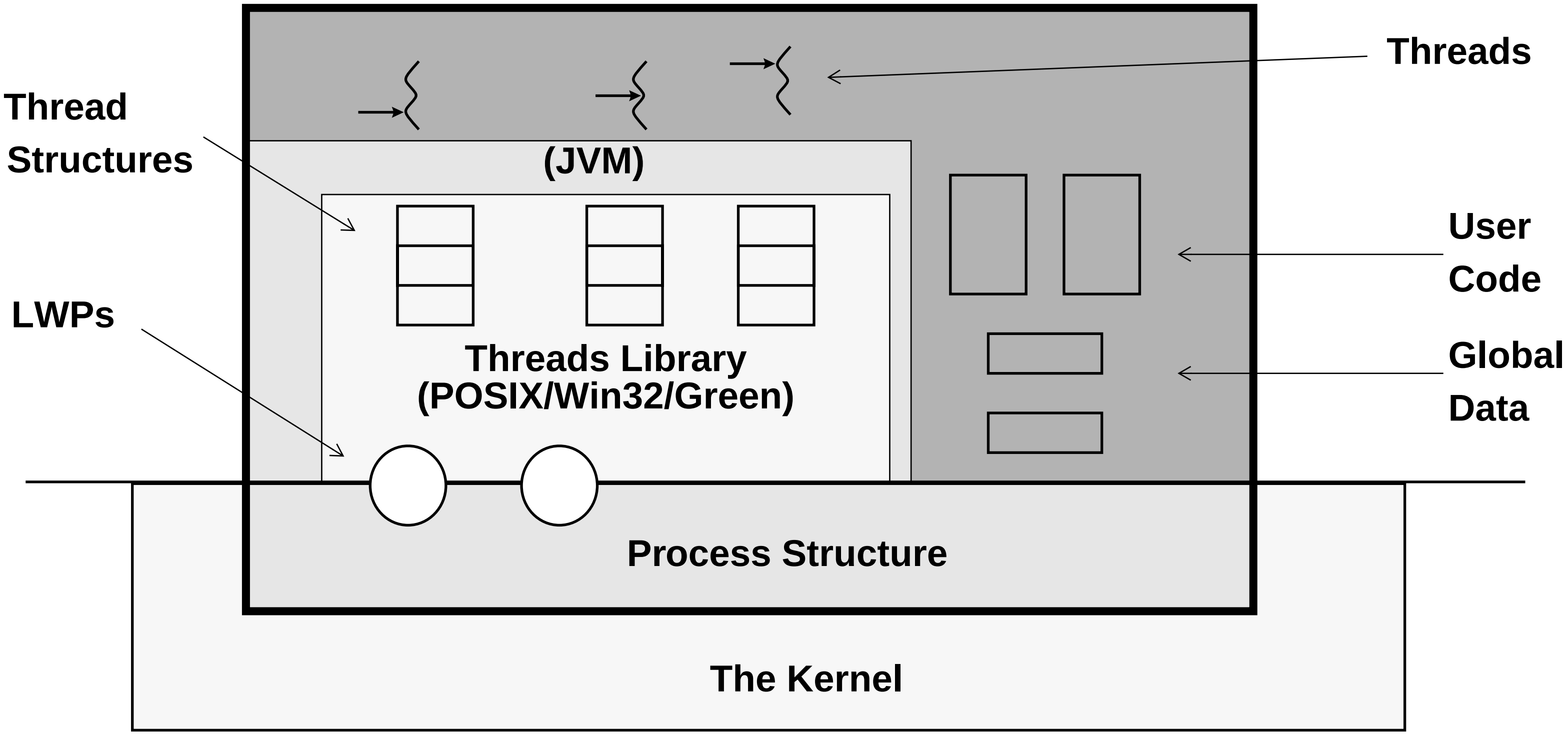
**Stack**

**Pointer**

| |
|---|
| **Thread ID** |
| **Priority** |
| **Signal Mask** |
| **CPU Registers** |
| **...** |

**Thread Structure**

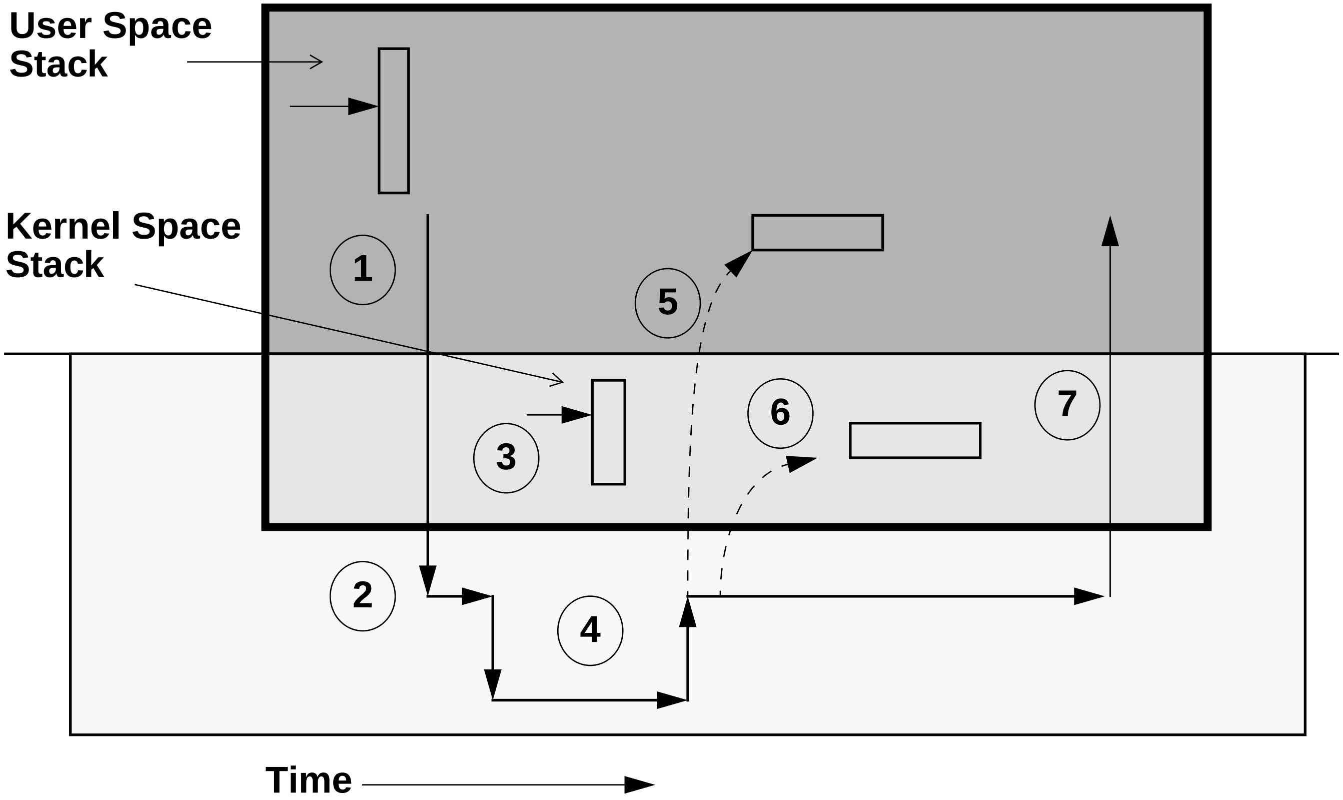
# Solaris Implementation of Pthreads



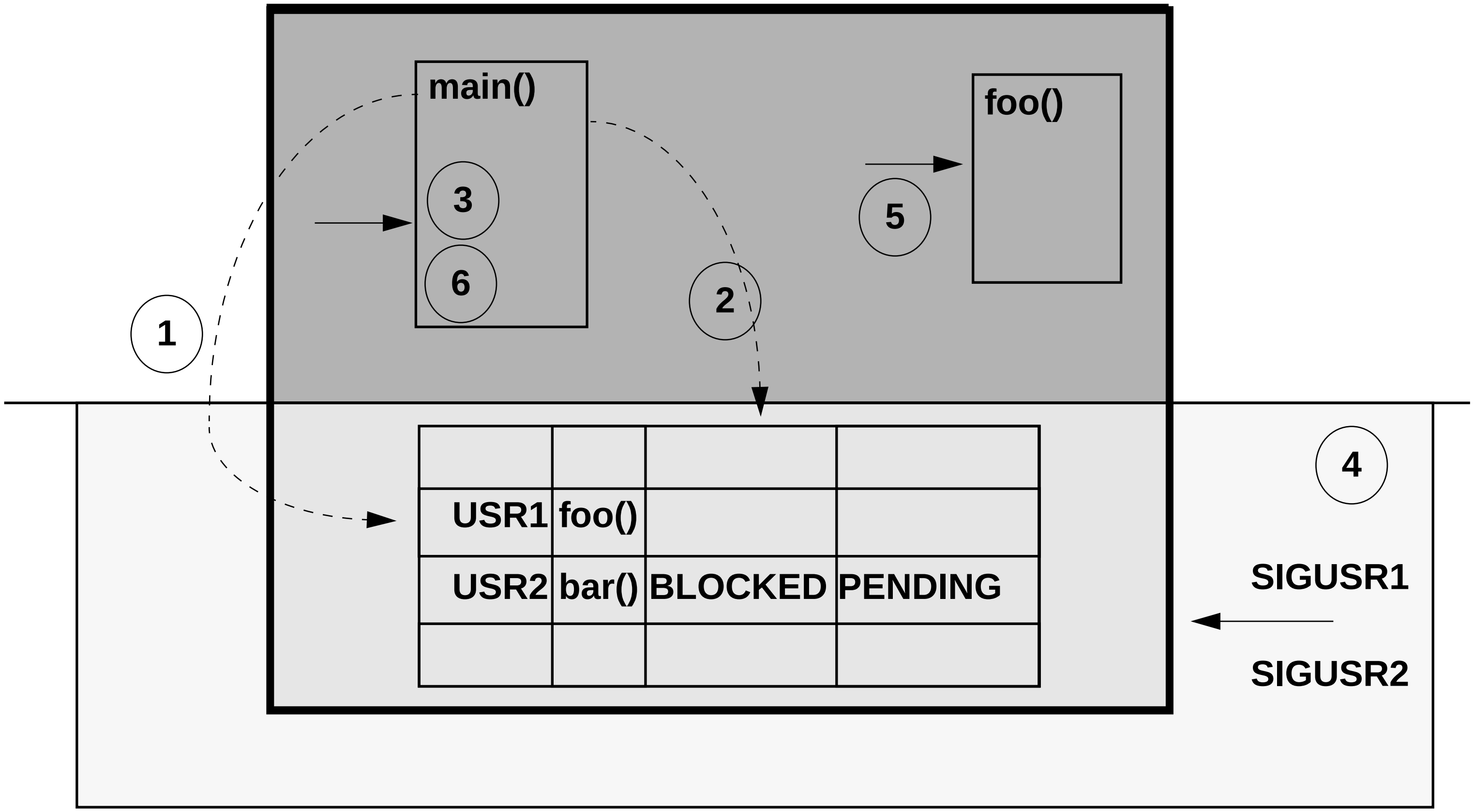
**`libpthread.so` is just a normal, user library!**

**Java threads are part of the JavaVM and similar.**

# How System Calls Work



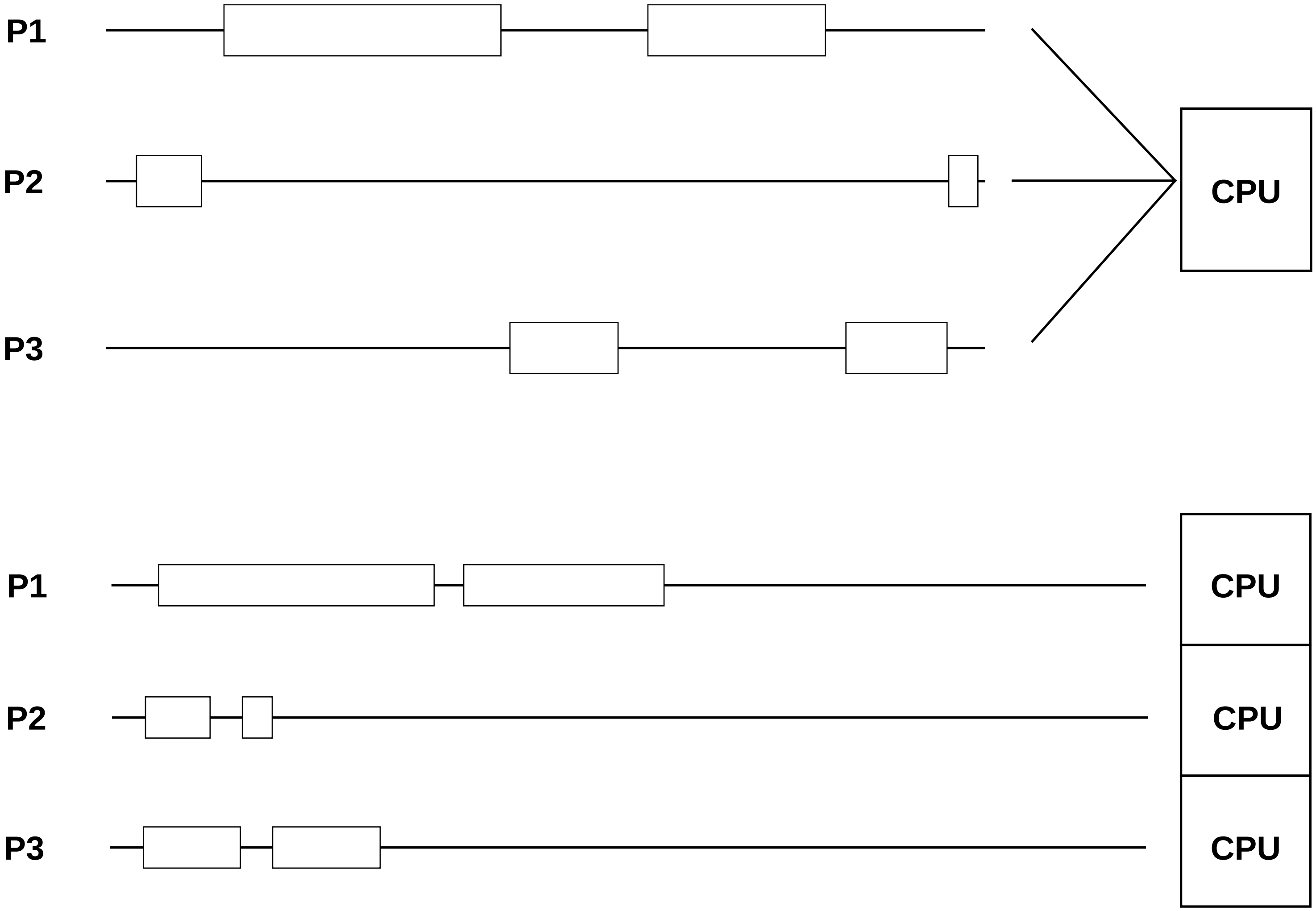
# How Signals Work



| | | | |
|---|---|---|---|
| | | | |
| USR1 | foo() | | |
| USR2 | bar() | BLOCKED | PENDING |
| | | | |

**MT obviates most of the need for signals.**

# Concurrency vs. Parallelism



# Automatic Threading

For a limited set of programs, it is possible for the compiler to infer the degree of data independence and separate out sections of the computation that can be done in parallel.

- MP Fortran - Sun, DEC, SGI, IBM
    - This compiler can take old f77 & f90 code and parallelize loops which do a lot of calculations.
    - Many numeric programs are amenable to MP FTN.
- MP C - Sun, SGI
    - This compiler takes ANSI C code and does the same.
    - Most C programs are too complex to parallelize.
- Ada - Sun, SGI
    - This is not “automatic” per-se, but Ada tasks will map directly to threads.

# Vendor Status

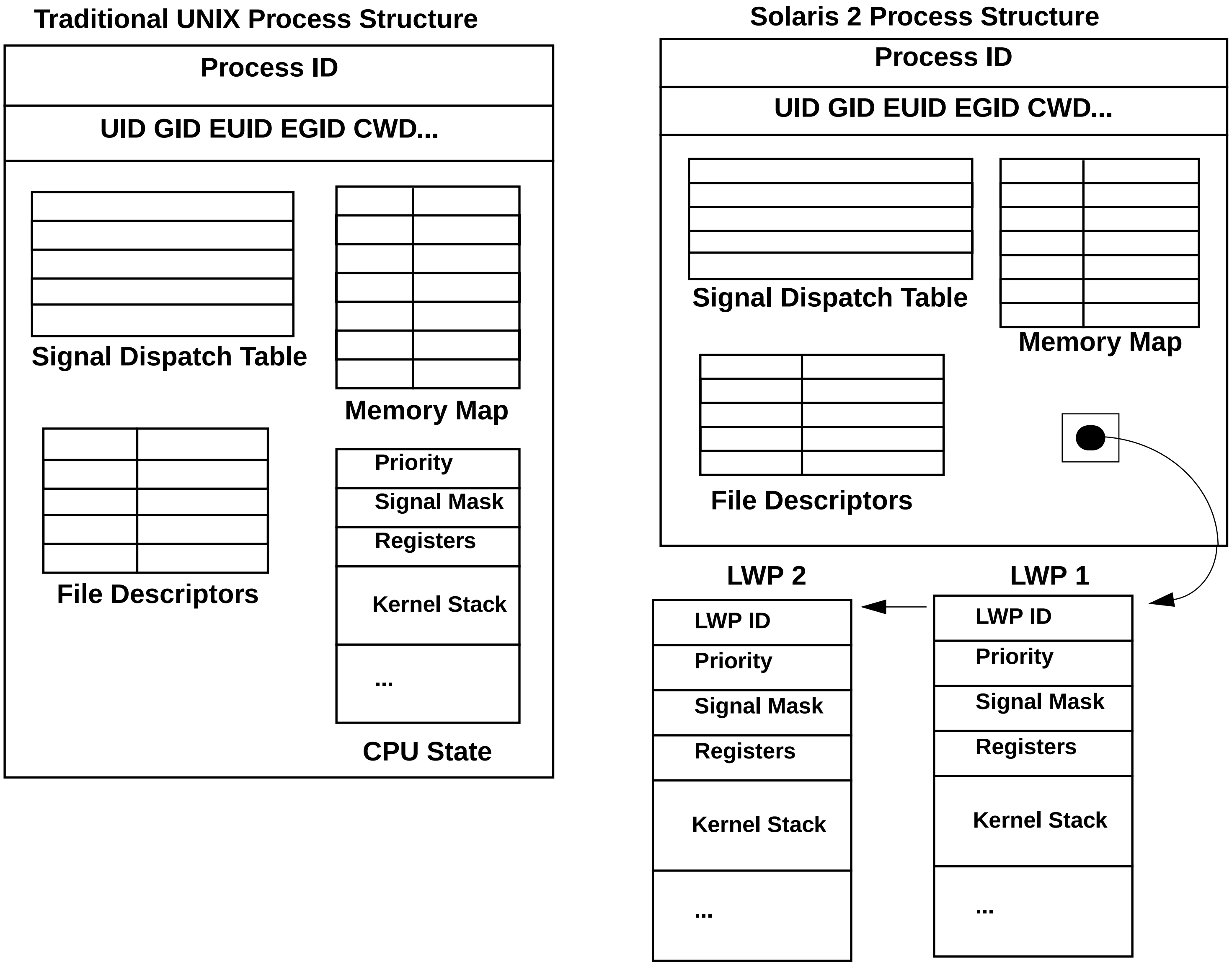
- HP-UX 11.0
    - Full POSIX standard pthreads.
- DEC UNIX 4.0
    - Full POSIX standard pthreads.
- SGI IRIX 6.2 (patch)
    - Full POSIX standard pthreads.
- IBM AIX 4.1.3
    - Full POSIX standard pthreads.
- DG UX
    - Some version of POSIX-style threads. (What?)
- Sun Solaris 2.5
    - Full POSIX standard pthreads.
    - Also full UI threads library.

# Vendor Status

- IBM OS/2
    - Full OS/2 threads library. Incompatible with POSIX, but designs are portable.
- Microsoft Windows
    - Four slightly different libraries for NT & Windows.
- Apple
    - Future release (1 - 2 years?) of non-POSIX threads
- Linux
    - Two freeware pthread packages
- SCO
    - SCO UNIX ?
    - UNIXWARE: UI Threads (very similar to Solaris Threads) Future Pthreads (when?)

# *Scheduling*

# Kernel Structures

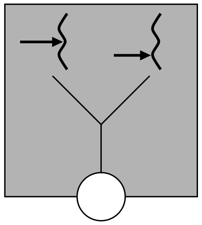


# Light Weight Processes

- **Virtual CPU**
- **Scheduled by Kernel**
- **Independent System Calls, Page Faults, Kernel Statistics, Scheduling Classes**
- **Can run in Parallel**
- **LWPs are an *Implementation Technique* (i.e., think about *concurrency*, not LWPs)**
- ***LWP* is a Solaris term, other vendors have different names for the same concept:**
    - **DEC: Lightweight threads vs. heavyweight threads**
    - **Kernel threads vs. user threads (also see “kernel threads” below)**

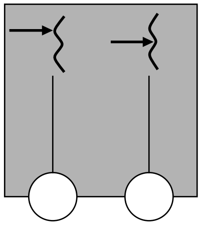
**(SUNOS 4.1 had a library called the “LWP” library. No relation to Solaris LWPs)**

# Scheduling
# Design Options



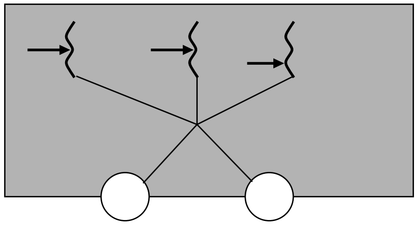
**M:1**

**HP-UX 10.20**
**(via DCE)**
**Green Threads**

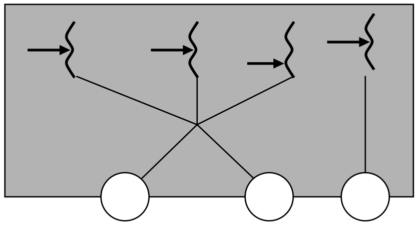


**1:1**

**Win32, OS/2, AIX 4.0**

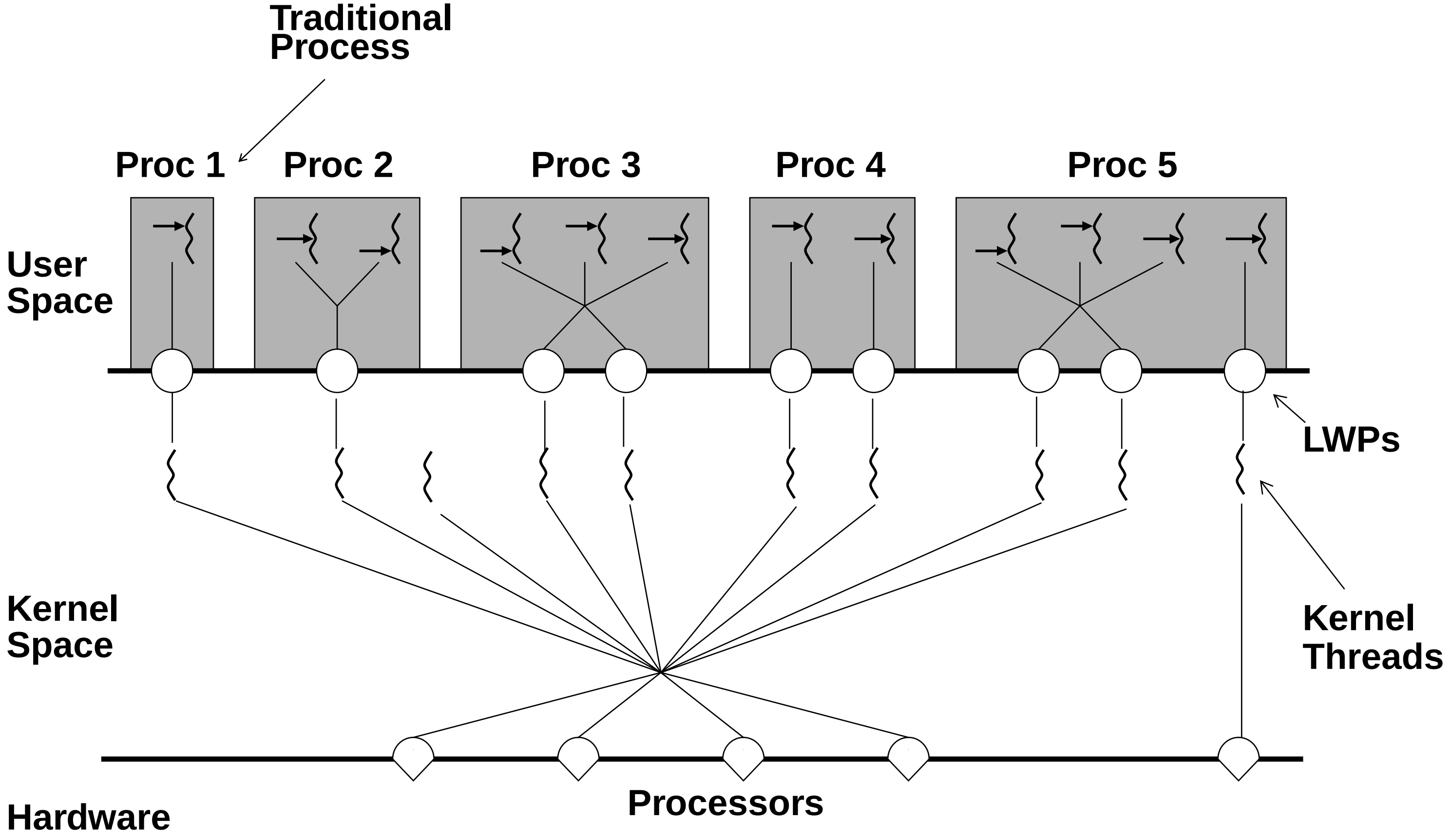


**M:M (strict)**



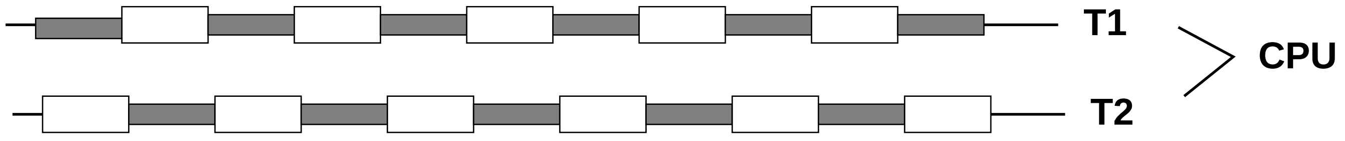
**2-Level (aka: M:M)**
**Solaris, DEC, IRIX, HP-UX 10.30, AIX 4.1**

# Solaris Two-Level Model

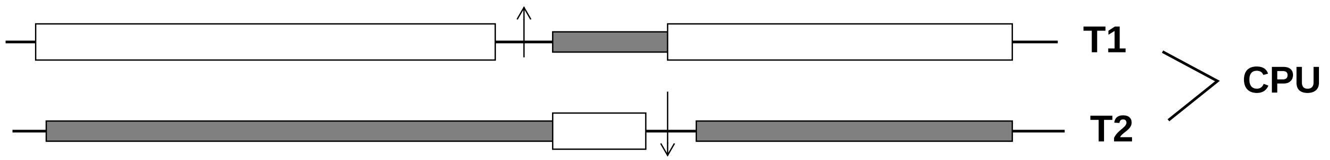


**“Kernel thread” here means the threads that the OS is built with. On HP, etc. it means LWP.**

# Time Sliced Scheduling
# vs.
# Strict Priority Scheduling



Typical Time Sliced Scheduling



Typical Strict Priority Scheduling

□ Working ▨ Sleeping ↑ Request ↓ Reply

# Inter-Thread Dependencies

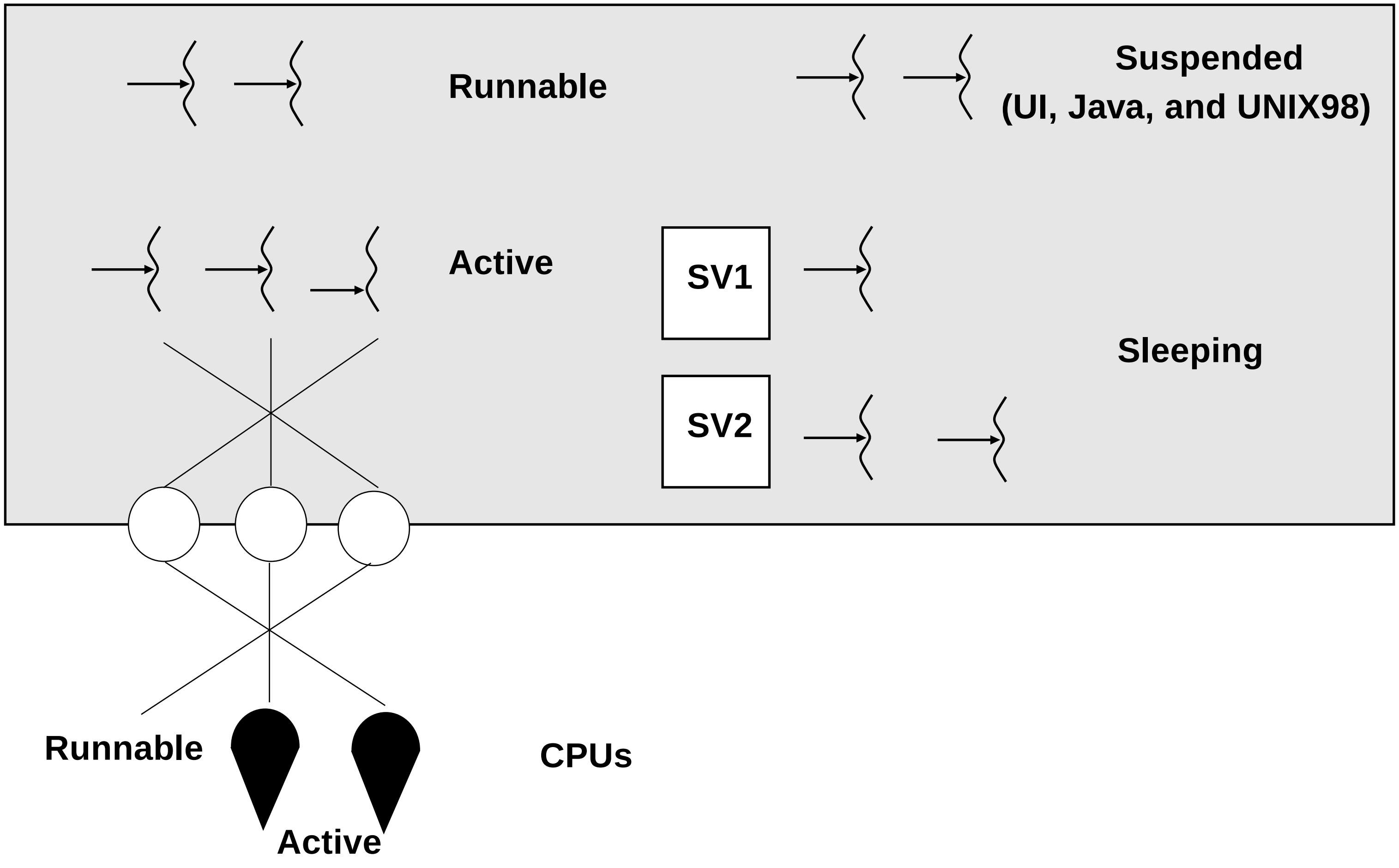
- Independent
  - Two users running DOOM:
    Each will use 100% of the CPU
  - Requires time slicing for “fairness”
- Dependent
  - A text editor and a file system:
    Each needs the other to do something
  - Needs no time slicing
- Reality is a mixture
- Scheduling is an unsolved problem

# System Scope “Global” Scheduling
# vs.
# Process Scope “Local” Scheduling

- **Global means that the kernel is scheduling threads (aka “bound threads”, 1:1, System Scope)**
    - **Requires significant overhead**
    - **Allows time slicing**
    - **Allows real-time scheduling**

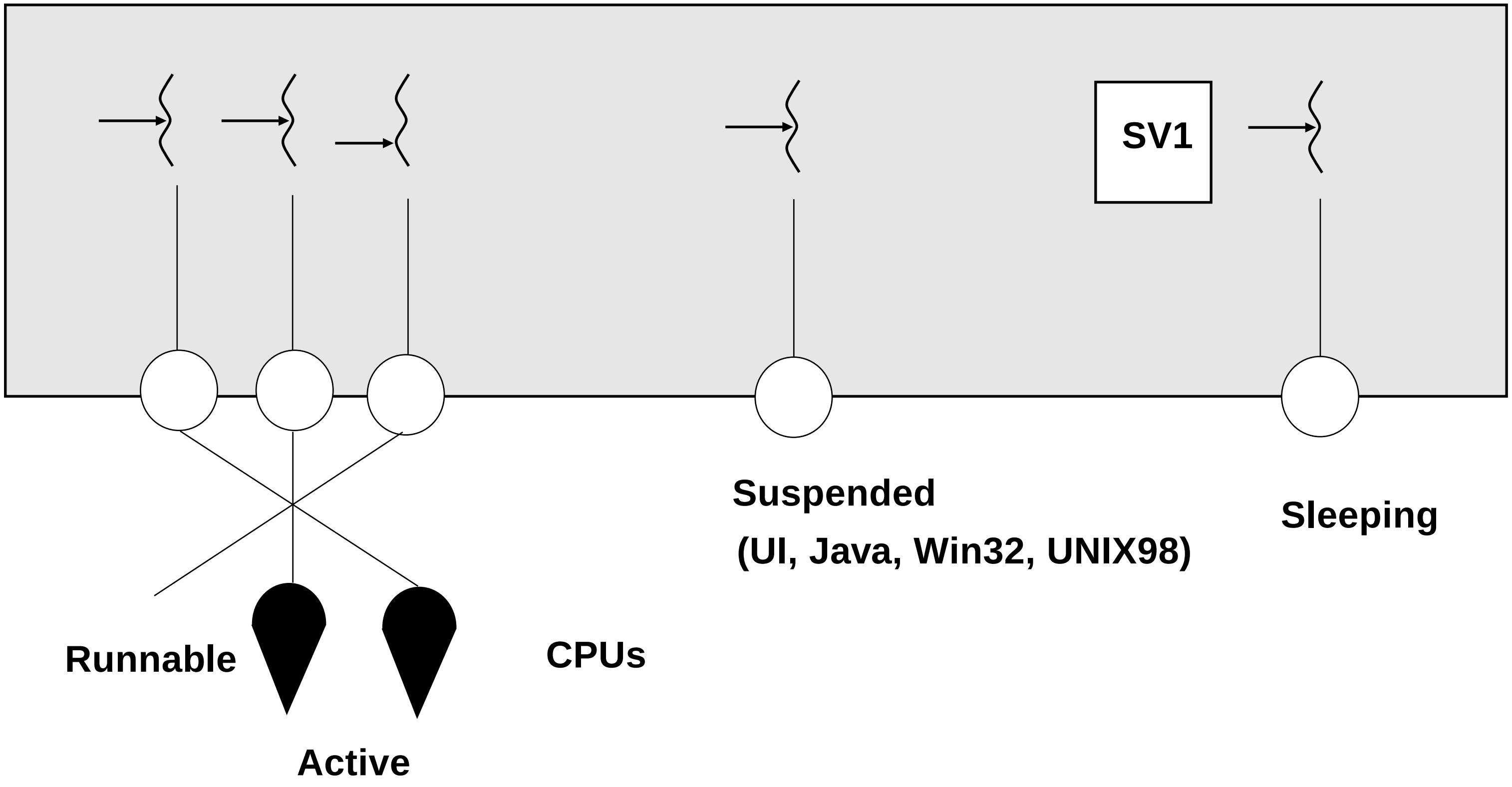
- **Local means the library is scheduling threads (aka “unbound threads”, M:M, Process Scope)**
    - **Very fast**
    - **Strict priority only, no time slicing (time slicing is done on Digital UNIX, not Solaris)**

# Local Scheduling States



**(UI, Java, and POSIX only)**

# Global Scheduling States



**(UI, Java, POSIX, Win32)**

# Scheduler Performance

## Creation Primitives

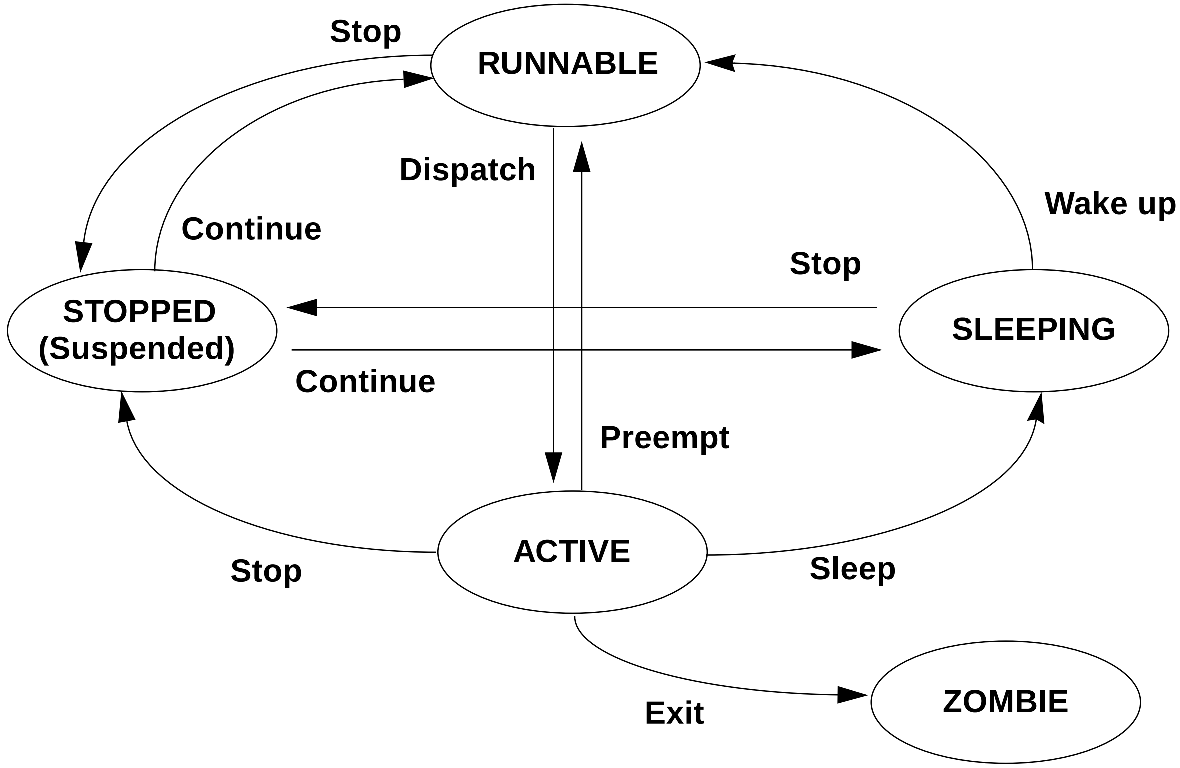
| | POSIX<br>uSecs | Java 2<br>uSecs |
|---|---|---|
| Local Thread | 330 | 2,600 |
| Global Thread | 720 | n/a |
| Fork | 45,000 | |

## Context Switching

| | POSIX<br>uSecs | Java 2<br>uSecs |
|---|---|---|
| Local Thread | 90 | 125 |
| Global Thread | 40 | n/a |
| Fork | 50 | |

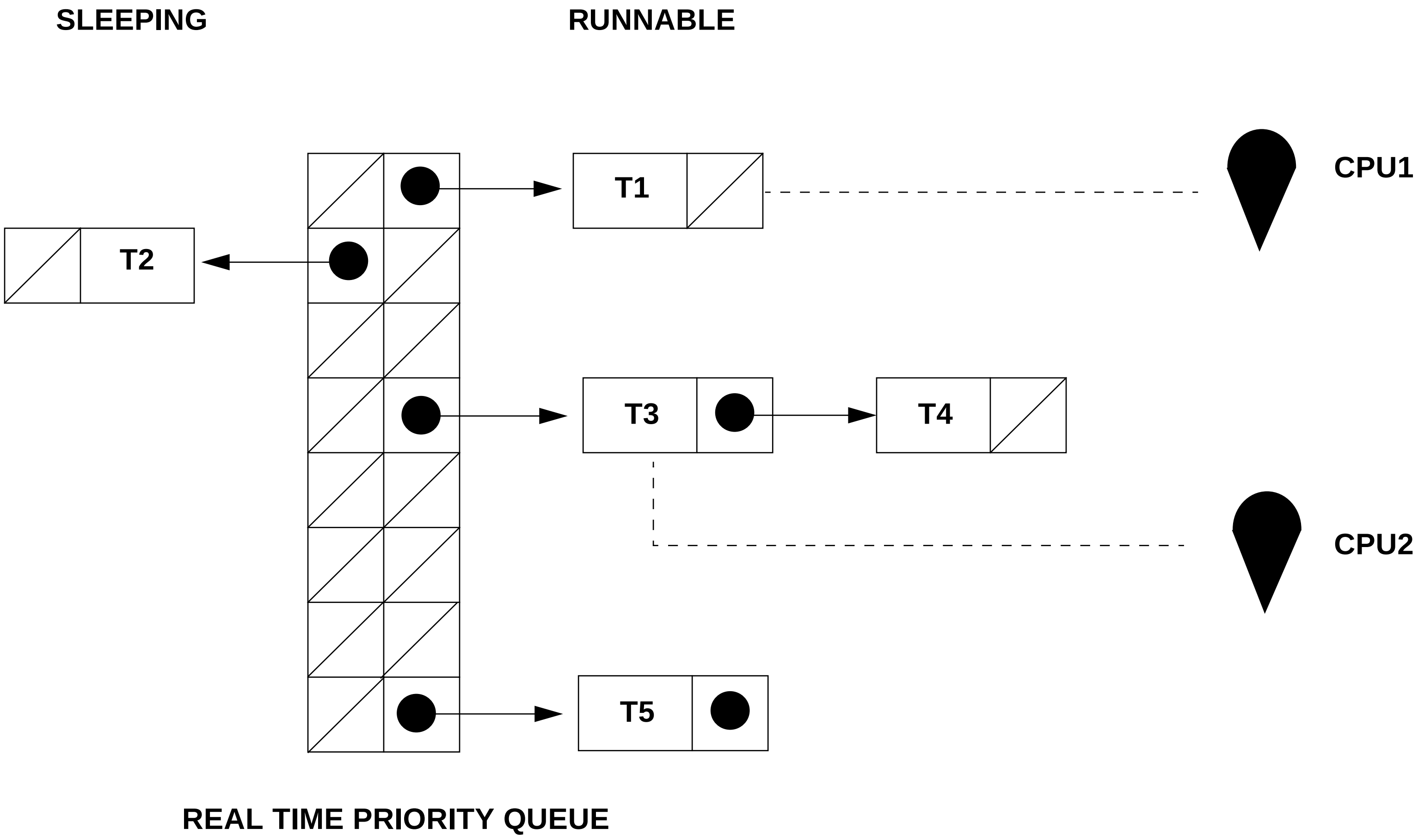
**110MHz SPARCstation 4 (This machine has a SPECint of about 20% of the fastest workstations today.) running Solaris 2.5.1.**

# Scheduling States



**"Stopped" (aka "Suspended") is only in Win32, Java, and UI threads, not POSIX. No relation to the java method `thread.stop()`**

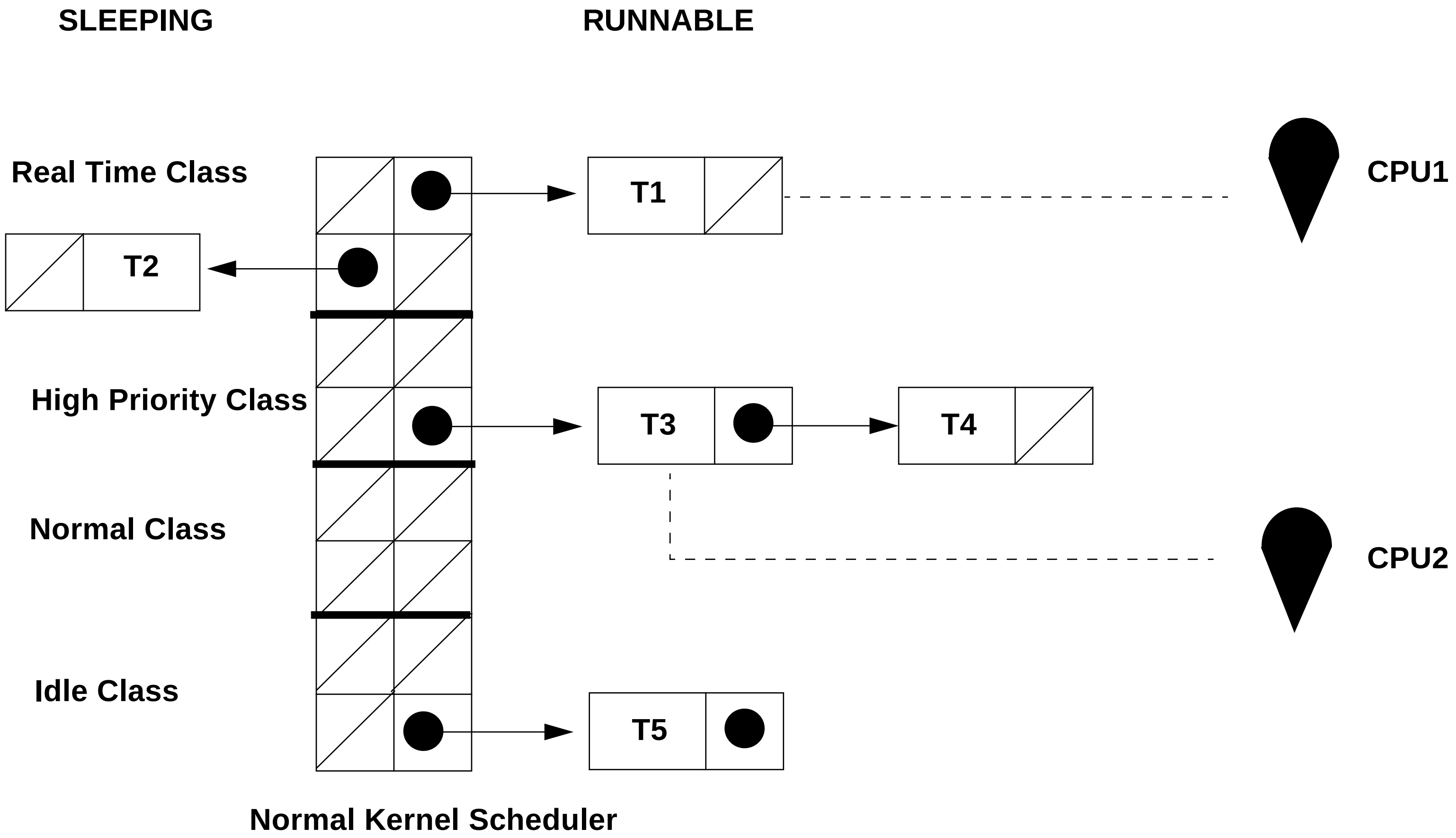
# Realtime Global POSIX Scheduling



**Strict FIFO Priority does no Timeslicing, Round Robin does.**

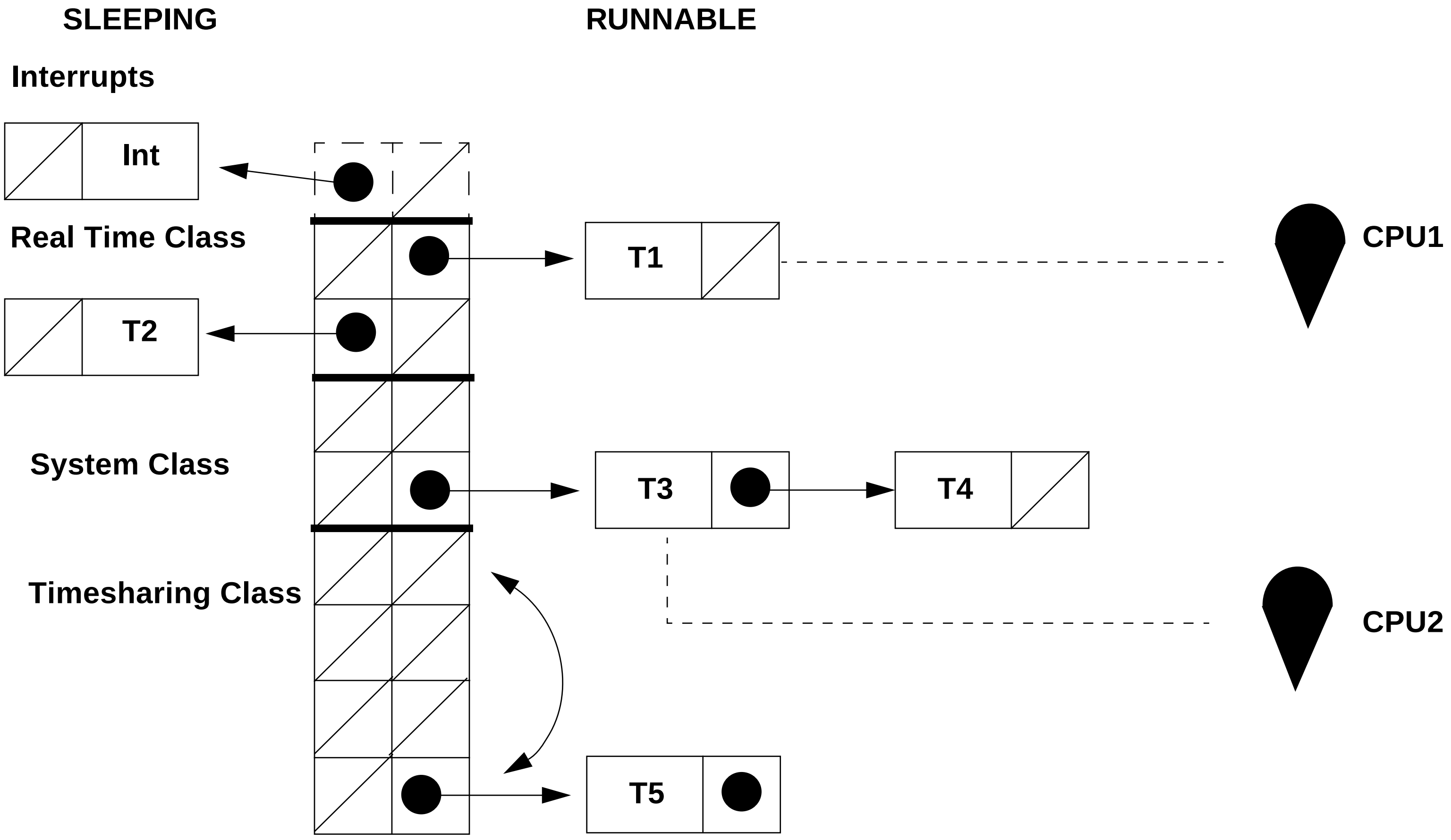
**(This is *optional* in POSIX, and not implemented in many OSs yet.)**

# NT Scheduling



## All Round Robin (timeslicing)

# Solaris Scheduling



Normal Solaris Kernel Scheduler (60 priority levels in each class)

**RT and system classes do RR, TS does demotion/promotion.**

# Solaris Kernel Level Scheduling

| | | | |
|---|---|---|---|
| Real Time Class Priority | Time Slice | | |
| 59 | 10 | | |
| 58 | 10 | | |
| ... | ... | | |
| 0 | 100 | | |
| System Class Priority | Time Slice | | |
| 59 | 10 | | |
| 58 | 10 | | |
| ... | ... | | |
| 0 | 10 | | |
| Timesharing Class Priority | Time Slice | If I/O Interrupted | If Time Slice Completed |
| 59 | 10 | 59 | 49 |
| 58 | 10 | 59 | 48 |
| ... | ... | ... | ... |
| 0 | 100 | 10 | 0 |

**All these entries may be redefined by the super user (`dispadmin`).**

**`nice`**

# POSIX Priorities

For realtime threads, POSIX requires that at least 32 levels be provided and that the highest priority threads always get the CPUs (as per previous slides).

For non-realtime threads, the meaning of the priority levels is not well-defined and left to the discretion of the individual vendor. On Solaris with bound threads, the priority numbers are ignored.

***You will probably never use priorities.***

# Java Priorities

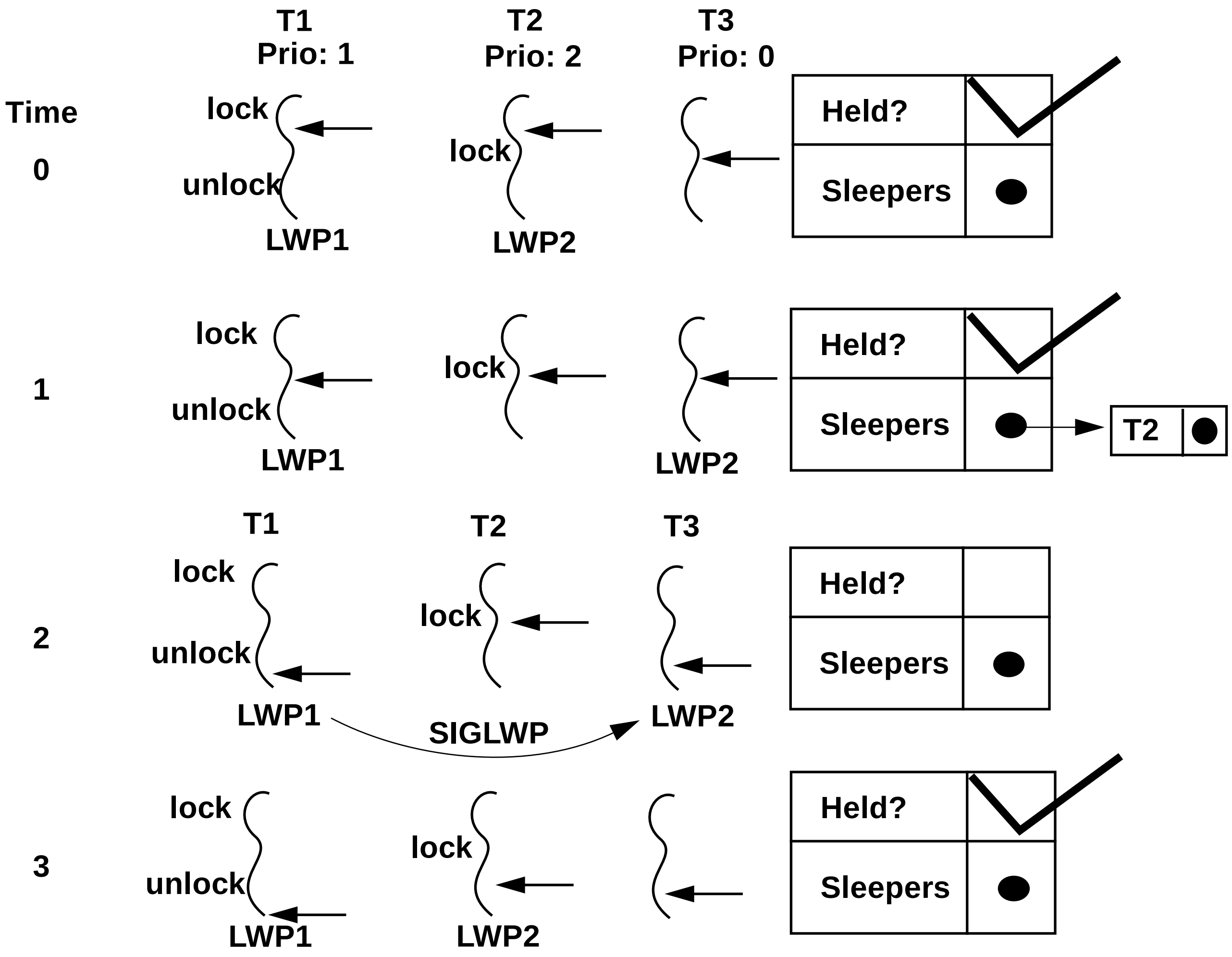
**The Java spec requires that 10 levels be provided, the actual values being integers between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`. The default level for a new thread is `Thread.NORM_PRIORITY`.**

**Threads may examine or change their own or others' priority level with `thread.getPriority()` and `thread.setPriority(int)`. Obviously, `int` has to be with in the range `Thread.MIN_PRIORITY`, `Thread.MAX_PRIORITY.` It may be further restricted by the thread's group via `threadGroup.getMaxPriority()` and `threadGroup.setMaxPriority(int).`**

**On Solaris these are mapped to 1, 10, and 5, although this is not required. These numbers are not propagated down to the LWPs and are therefor meaningless when there are enough LWPs. For Green threads, the priority levels are absolute.**

***You will probably never use priorities.***

# Local Context Switching



**Solaris uses SIGLWP, Digital UNIX uses a different method. The results are the same.**

# Preemption

The process of rudely interrupting a thread (or LWP) and forcing it to relinquish its LWP (or CPU) to another.

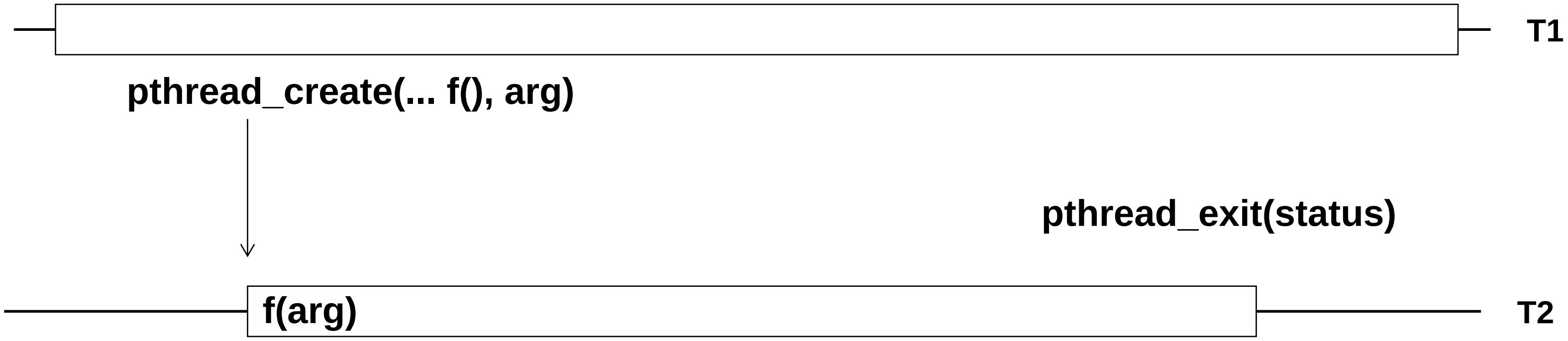
CPU2 cannot change CPU3’s registers directly. It can only issue a hardware interrupt to CPU3. It is up to CPU3’s interrupt handler to look at CPU2’s request and decide what to do. (Ditto for LWPs and signals.)

Higher priority threads always preempt lower priority threads.

Preemption != Time Slicing

All of the libraries are preemptive.

# Thread Life Cycle



T1

pthread_create(... f(), arg)

pthread_exit(status)

f(arg)

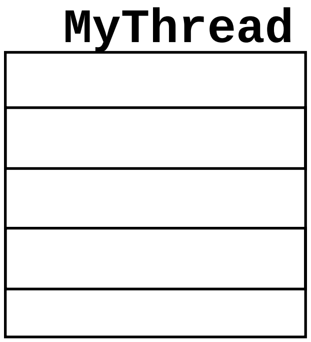
T2

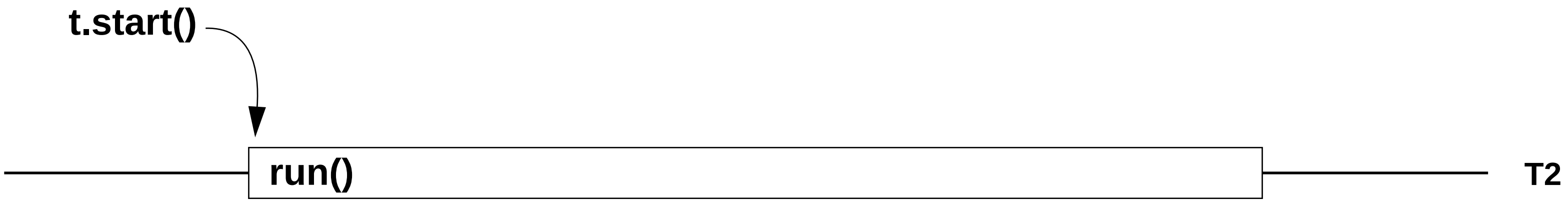
```
POSIX                   Win32                  Java

main()                  main()                 main()
{...                    {...                   { MyThread t;
pthread_create(f, arg)  CreateThread(f, arg)
...                     _beginthread(f, arg)     t = new MyThread();
}                       _xbeginthread(f,arg)     t.start();
                        }                      }

void *f(void *arg)      DWORM f(DWORD arg)     class MyThread
                                                     extends Thread
{...                    {...                   {
pthread_exit(status);   ExitThread(status);      public void run()
}                       _endthread(status);      {...
                        _xendthread(status);      return()
                                                 }
                                               }
```

# Runnable vs. Thread

It is possible to subclass `Thread` and define a `run()` method for it:

```
public MyThread extends Thread
{
     public void run()
     {do_stuff();}
}


MyThread t = new MyThread();

t.start();
```

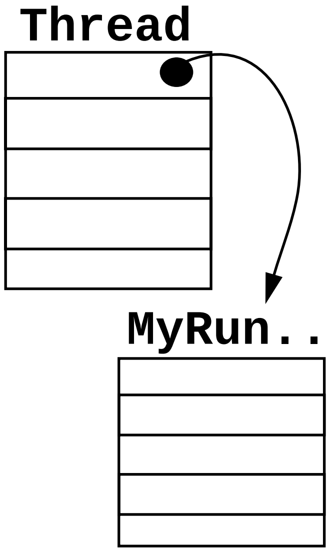




# Runnable vs. Thread

**But it is (always ?) better to define a `Runnable` and have a stock Thread run it:**

```
public MyRunnable implements Runnable
{
     public void run()
     {do_stuff();}
}
```



```
Thread t = new Thread(new MyRunnable());

t.start();
```

**Logically speaking, we are not changing the nature of the thread, so we shouldn’t be subclassing it. Plus the fact, now we can subclass something more useful if we want. (No multiple inheritance in Java.)**

# Self Starting Threads

**It is also possible to have a thread start running as soon as construction is complete:**

```
public MyRunnable implements Runnable
{
     public MyRunnable()
     {new Thread(this).start();}

     public void run()
     {do_stuff();}
}
```

**But I don’t think this gains you anything (you save one line of code) and subclassing gets rather hairy.**

***Don’t do this.***

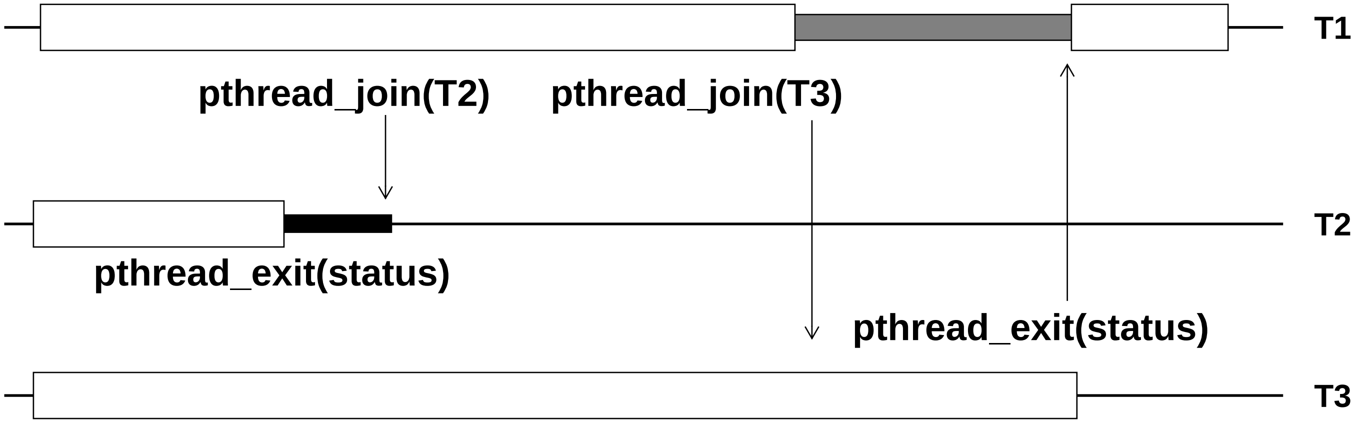
# Restarting a Thread

In Java, a `Thread` object is just an object that happens to have a pointer to the actual thread (stack, kernel structure, etc.). Once a thread has exited (“stopped”), it is gone. It is not “restartable” (whatever that means!).

A `Runnable`, on the other hand, may be used for as many threads as you like. Just don’t put any instance variables into your `Runnable`.

We like to view `Threads` as the engines for getting work done, while the `Runnable` is the work to be done.

# Waiting for a Thread to Exit



```
POSIX                Win32                       Java

pthread_join(T2);    WaitForSingleObject(T2)     T2.join();
```

**There is no special relation between the creator of a thread and the waiter. The thread may die before or after join is called. You may join a thread only once.**

# EXIT vs. THREAD_EXIT

The normal C function `exit()` always causes the process to exit. That means all of the process -- All the threads.

The thread exit functions:

| | |
|---|---|
| POSIX: | `pthread_exit()` |
| Win32: | `ExitThread()` and `endthread()` |
| Java: | `thread.stop()` (vs. `System.exit()`) |
| UI: | `thr_exit()` |

all cause only the calling thread to exit, leaving the process intact and all of the other threads running. (If no other (non-daemon) threads are running, then `exit()` will be called.)

Returning from the initial function calls the thread exit function implicitly (via “falling off the end” or via `return()`).

Returning from `main()` calls `_exit()` implicitly. (C, C++, not Java)

# `stop()` is Deprecated in JDK 1.2

**As called from the thread being stopped, it is equivalent to the thread exit functions. As called from other threads, it is equivalent to POSXI asynchronous cancellation. As it is pretty much impossible to use it correctly, stop has been proclaimed officially undesirable.**

**If you wish to have the current thread exit, you should have it return (or throw an exception) up to the run() method and exit from there. The idea is that the low level functions shouldn't even know if they're running in their own thread, or in a "main" thread. So they should never exit the thread at all.**

# `destroy()` was Never Implemented

**So don't use it.**

# POSIX Thread IDs are *Not* Integers

**They are defined to be opaque structures in all of the libraries. This means that they cannot be cast to a `(void *)`, they can not be compared with ==, and they cannot be printed.**

**In implementations, they CAN be...**

- **Solaris: `typedef unsigned int pthread_t`**
    - **main thread: 1; library threads: 2 & 3; user threads 4, 5...**
- **IRIX: `typedef unsigned int pthread_t`**
    - **main thread: 65536; user threads ...**
- **Digital UNIX: ?**
- **HP-UX: `typedef struct _tid{int, int, int} pthread_t`**

**I define a print name in thread_extensions.c:**

```
thread_name(pthread_self()) -> “T@123”
```

# Thread IDs are *Not* Integers

```
Good Programmer                  Bad Programmer

if (pthread_equal(t1, t2)) ...   if (t1 == t2) ...


int foo(pthread_t tid)           int foo(void *arg)
{...}                            {...}

foo(pthread_self());             foo((void *) pthread_self());


pthread_t tid = pthread_self();


printf(“%s”, thread_name(tid));  printf(“t@%d”, tid);


                                 #define THREAD_RWLOCK_INITIALZER \
        ??!                        {PTHREAD_MUTEX_INITIALIZER, \
                                     NULL_TID}
```

# Java Thread Names

A Java `Thread` is an `Object`, hence it has a `toString()` method which provides an identifiable, printable name for it.

You may give a thread your own name if you so wish:

```
Thread t1 = new Thread(“Your Thread”);
Thread t2 = new Thread(MyRunnable, “My Thread”);

System.out.println(t2 + “ is running.”);
==> Thread[My Thread,5,] is running.
System.out.println(t2.getName() + “ is running.”);
==> My Thread is running.
```

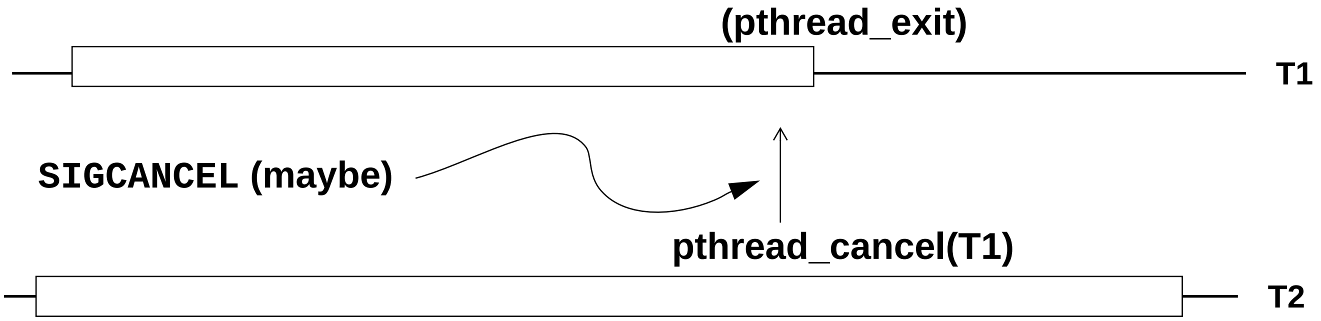
# Win32 TIDs & Thread Handles

Win32 uses two methods of referring to threads. The TID is process-local referent, the handle is a system-wide kernel referent. You can do some things with a TID and other things with a handle!?

```
handle = CreateThread(NULL, 0, fn, arg, 0, &tid);
```

# Cancellation

Cancellation is the means by which a thread can tell another thread that it should exit.



| POSIX | Win32 | Java |
|---|---|---|
| `pthread_cancel(T1);` | `TerminateThread(T1)` | `T1.stop()` |

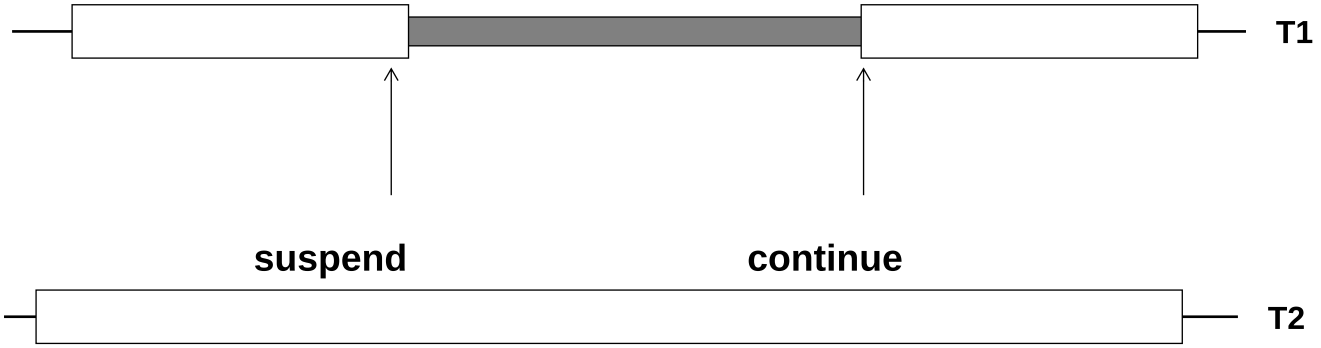
There is no special relation between the killer of a thread and the victim.

# Returning Status

- **POSIX and UI**
    - **A *detached* thread cannot be waited for (“joined”). It cannot return status.**
    - **An *undetached (“joinable”)* thread must be waited for (“joined”), and can return status.**
- **Java**
    - **Any thread can be waited for (*except* main!)**
    - **No thread can return status.**
- **Win32**
    - **Any thread can be waited for.**
    - **Any thread can return status.**

***Waiting for threads or returning status is not recommended.***

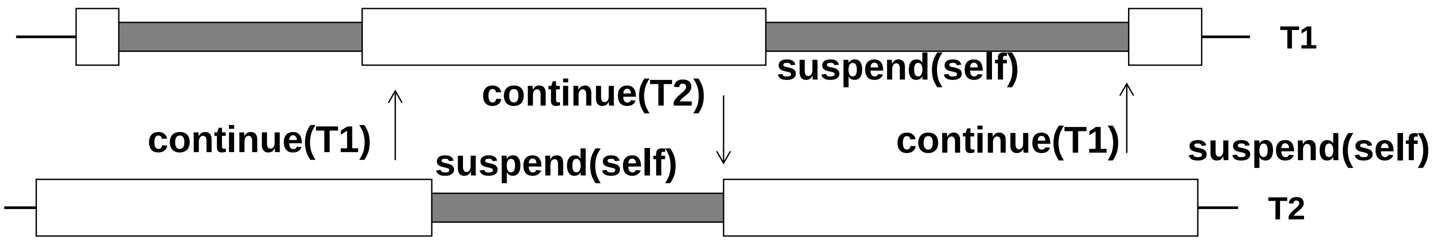
# Suspending a Thread



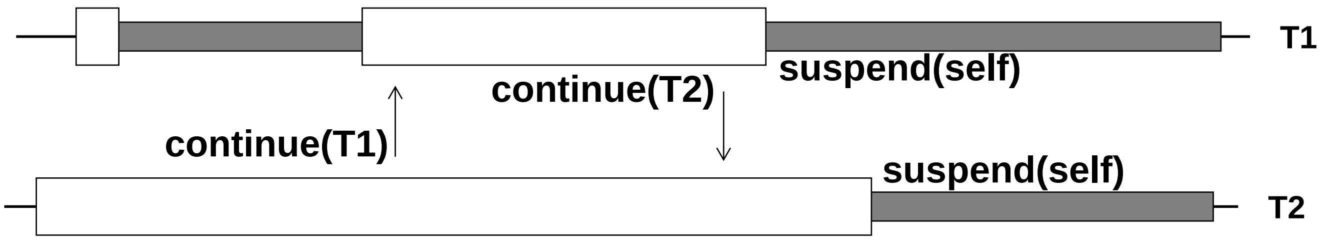
```
UI                      Win32                   Java

{...                    {...                    {...
thr_suspend(T1);        SuspendThread(T1);      T1.suspend();
...                     ...                     ...
thr_continue(T1);       ResumeThread(T1);       T1.resume();
...                     ...                     ...
}                       }                       }
```

- POSIX and UNIX98 do not define suspension
- You probably never want to use this!

# Suspending a Thread (Bad!)



DESIRED BEHAVIOR



POSSIBLE BEHAVIOR

```
T1                                T2

while(1)                          while(1)
  { work();                         { work();
    thr_continue(T2);                 thr_continue(T1);
    thr_suspend(self);                thr_suspend(self);
  }                                 }
```

# Proposed Uses of Suspend/Continue

- Garbage Collectors
- Debuggers
- Performance Analyzers
- Other Tools?
- NT Services specify that a service should be suspendable.

***Be Careful!***

# sched_yield()
# Thread.yield()

- **Will relinquish the LWP (or CPU for bound threads) to anyone who wants it (at the same priority level, both bound and unbound).**
- **Must NOT be used for correctness!**
- **Probably not useful for anything but the oddest programs (and then only for “efficiency” or “fairness”.**
- **The JVM on some platforms (e.g., Solaris with Green threads) may require it under some circumstances, such as computationally bound threads.) This will change.**
- **Never Wrong, but...**

***Avoid `sched_yield()` `Thread.yield()` !***

# Dæmon Threads

**A dæmon is a normal thread in every respect save one: Dæmons are not counted when deciding to exit. Once the last non-dæmon thread has exited, the entire application exits, taking the dæmons with it.**

**Dæmon Threads exist only in UI, and Java threads.**

**UI:**

```
thr_create(..., THR_DAEMON,...);
```

**Java:**

```
void thread.setDaemon(boolean on);

boolean thread.isDaemon();
```

**Avoid using Dæmons!**

# Do NOT Think about Scheduling!

- Think about Resource Availability
- Think about Synchronization
- Think about Priorities

# Example Programs and Exercises

| | | | |
|---|---|---|---|
| Login: | student | thread{1-10} | javat{1-10} |
| Password: | 4software | letmein | EtAvraa |

(you may run either OpenLook or CDE)

To build all programs (POSIX):

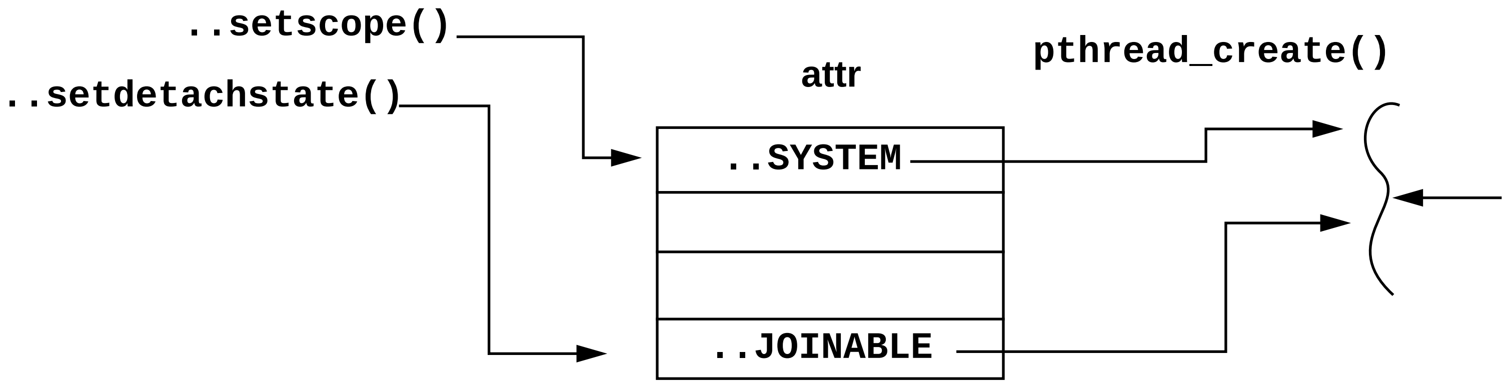
```
% cd programs/Pthreads
% make
% workshop &      // You must run workshop from this directory to
                   get the local dynamic libraries.
```

To see all programs (Java):

```
% cd programs/Java
% jws &
% xemacs &
```

# Attribute Objects

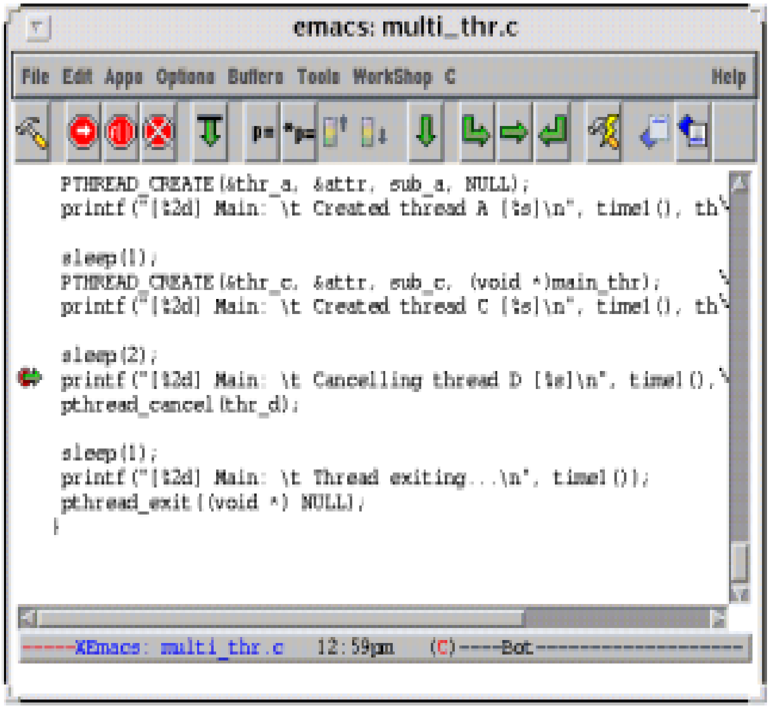
```
pthread_attr_t attr;
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER, lock2;

pthread_mutex_init(&lock2, NULL);
PTHREAD_ATTR_INIT(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS); */

pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
/* pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); */

PTHREAD_CREATE(&tid_d, &attr, sub_d, NULL);
```



# Wrappers

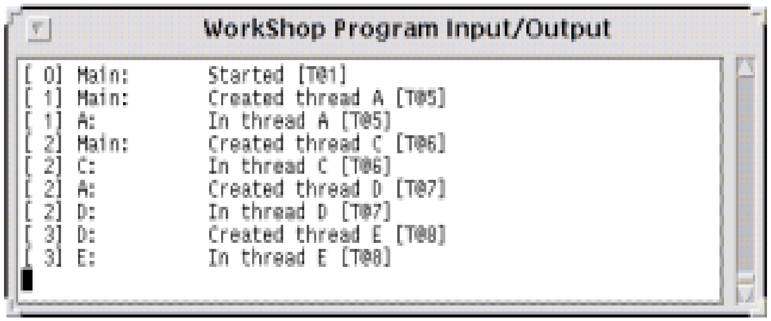
```
int PTHREAD_CREATE(pthread_t *new_thread_ID,
   const pthread_attr_t *attr,
   void * (*start_func)(void *), void *arg)
{int err;
 pthread_t tid;

 if (err = pthread_create(new_thread_ID, attr, start_func, arg))
   {printf("%s\n", strerror(err));
    abort();
  }

}



int PTHREAD_ATTR_INIT(pthread_attr_t *a)
...

int PTHREAD_CONDATTR_INIT(pthread_condattr_t *a)
...

int PTHREAD_MUTEXATTR_INIT(pthread_mutexattr_t *a)
...

int SEM_INIT(sem_t *sem, int pshared, unsigned int value)
...
```

# Workshop Debugger (4.0)

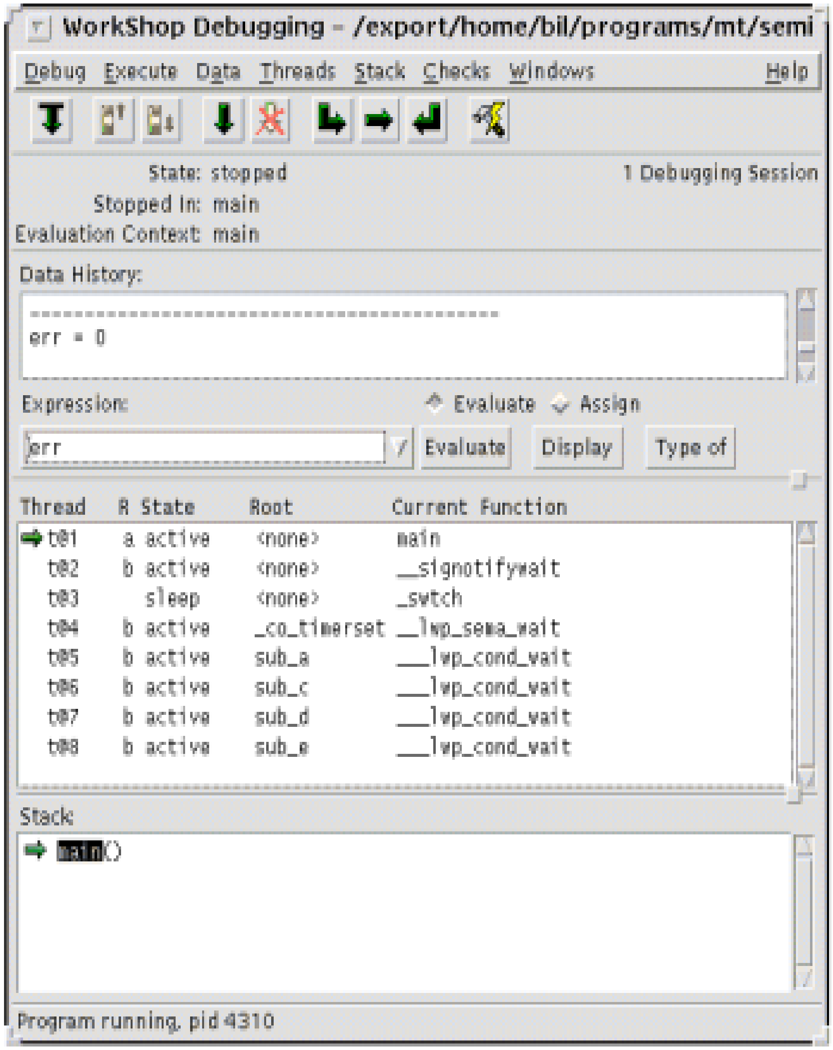


```
PTHREAD_CREATE(&thr_a, &attr, sub_a, NULL);
printf("[%2d] Main: \t Created thread A [%s]\n", time1(), th\

sleep(1);
PTHREAD_CREATE(&thr_c, &attr, sub_c, (void *)main_thr);     \
printf("[%2d] Main: \t Created thread C [%s]\n", time1(), th\

sleep(2);
printf("[%2d] Main: \t Cancelling thread D [%s]\n", time1(),\
pthread_cancel(thr_d);

sleep(1);
printf("[%2d] Main: \t Thread exiting...\n", time1());
pthread_exit((void *) NULL);
}
```

```
-----XEmacs: multi_thr.c    12:59pm    (C)----Bot--------------------
```



```
[ 0] Main:      Started [T@1]
[ 1] Main:      Created thread A [T@5]
[ 1] A:         In thread A [T@5]
[ 2] Main:      Created thread C [T@6]
[ 2] C:         In thread C [T@6]
[ 2] A:         Created thread D [T@7]
[ 2] D:         In thread D [T@7]
[ 3] D:         Created thread E [T@8]
[ 3] E:         In thread E [T@8]
```



WorkShop Debugging - /export/home/bil/programs/mt/semi
Debug Execute Data Threads Stack Checks Windows Help

State: stopped
1 Debugging Session
Stopped In: main
Evaluation Context: main

Data History:

```
------------------------------------------
err = 0
```

Expression: ◆ Evaluate ◇ Assign

err

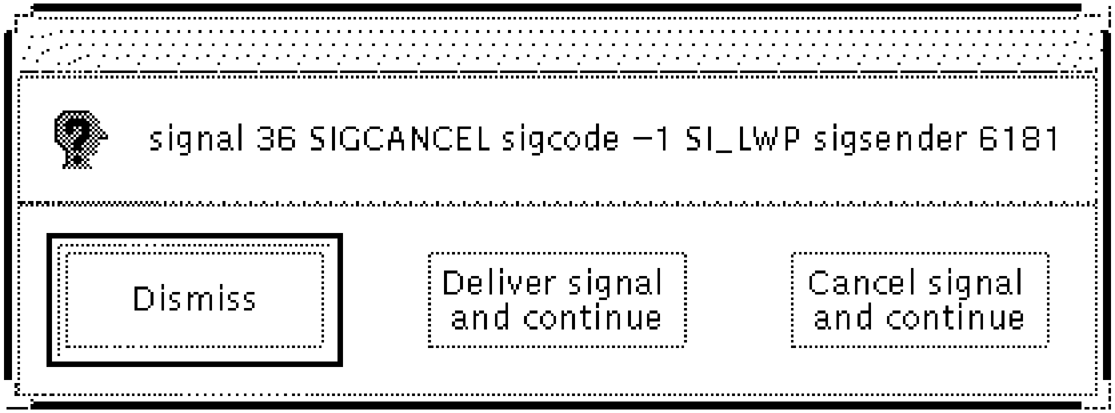
```
Thread    R State    Root          Current Function
t@1       a active   <none>        main
t@2       b active   <none>        __signotifywait
t@3         sleep    <none>        _swtch
t@4       b active   _co_timerset  __lwp_sema_wait
t@5       b active   sub_a         ___lwp_cond_wait
t@6       b active   sub_c         ___lwp_cond_wait
t@7       b active   sub_d         ___lwp_cond_wait
t@8       b active   sub_e         ___lwp_cond_wait
```

Stack:

main()

Program running, pid 4310

# Debugger and Signals



signal 36 SIGCANCEL sigcode −1 SI_LWP sigsender 6181

**Cancellation in Solaris is accomplished by sending SIGCANCEL.**

**By default, the debugger will catch this signal (and some others) and break. Normally you'll have the debugger deliver the signal and continue.**

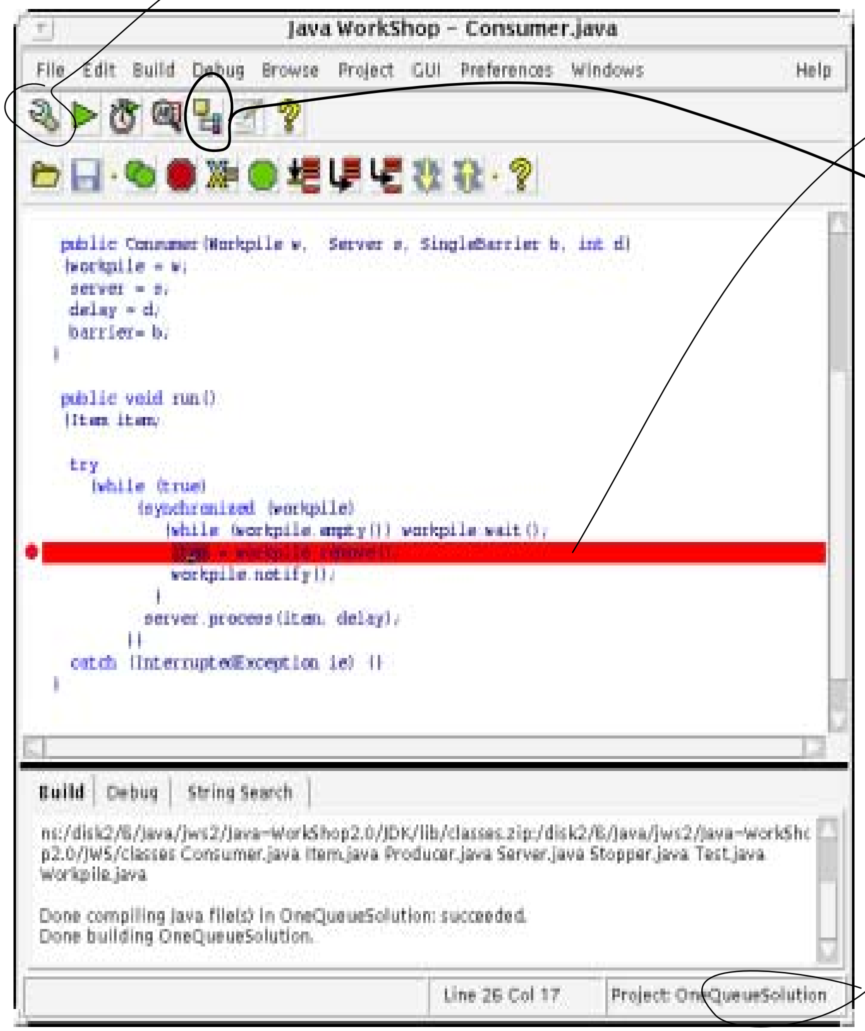
**To tell the debugger not to break on the signal, you can bring up a command window: Windows->Dbx Commands**

**and tell it to ignore SIGCANCEL:**

```
(dbx) ignore SIGCANCEL
```

**You can also put this into your dbx.rc file.**

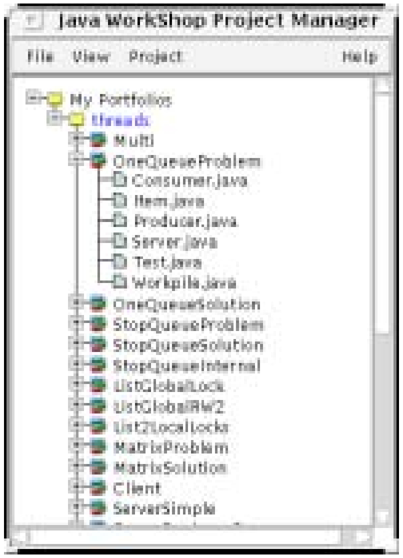
# Java Tools (Sun’s JWS 2.0)


Java WorkShop – Consumer.java
File Edit Build Debug Browse Project GUI Preferences Windows Help
Build Debug String Search

```
public Consumer(Workpile w,  Server s, SingleBarrier b, int d)
 {workpile = w;
  server = s;
  delay = d;
  barrier= b;
}

public void run()
 {Item item;

  try
    {while (true)
         {synchronized (workpile)
            {while (workpile.empty()) workpile.wait();
             item = workpile.remove();
             workpile.notify();
            }
          server.process(item, delay);
         }}
  catch (InterruptedException ie) {}
}
```

```
ns:/disk2/6/java/jws2/java-WorkShop2.0/JDK/lib/classes.zip:/disk2/6/java/jws2/java-WorkSho
p2.0/JWS/classes Consumer.java Item.java Producer.java Server.java Stopper.java Test.java
Workpile.java

Done compiling java file(s) in OneQueueSolution: succeeded.
Done building OneQueueSolution.
```

Line 26 Col 17 | Project: OneQueueSolution

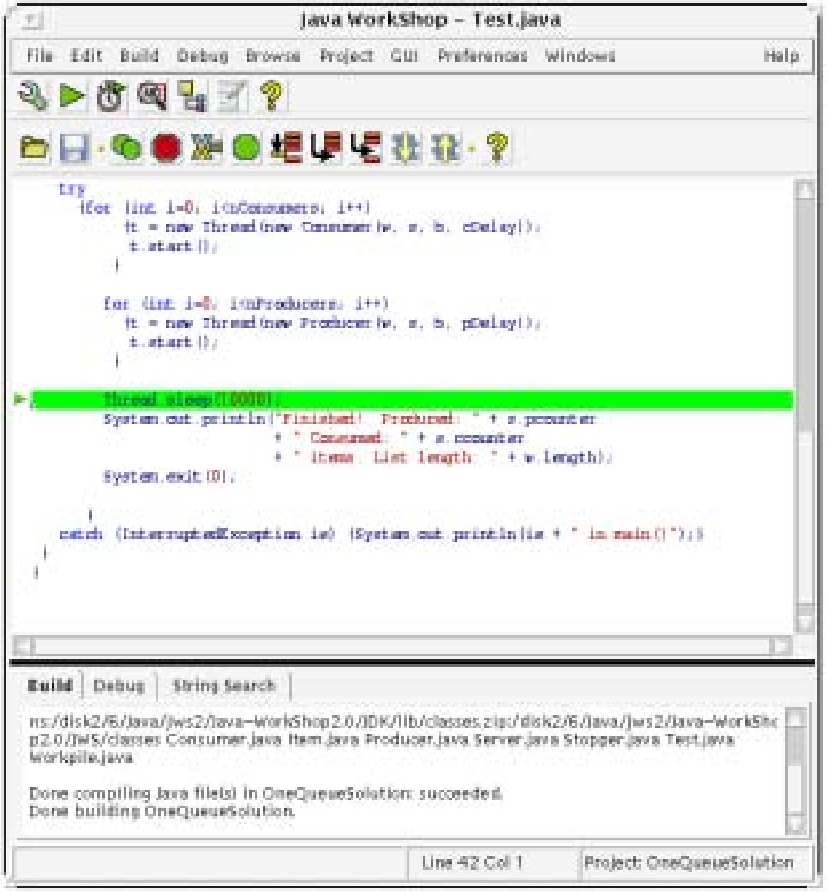


**Compile changed files**

**Red:::: Breakpoint on this line**
**Green: Stopped at Breakpoint**
**Purple: Cannot show code inside**
**(no source available)**

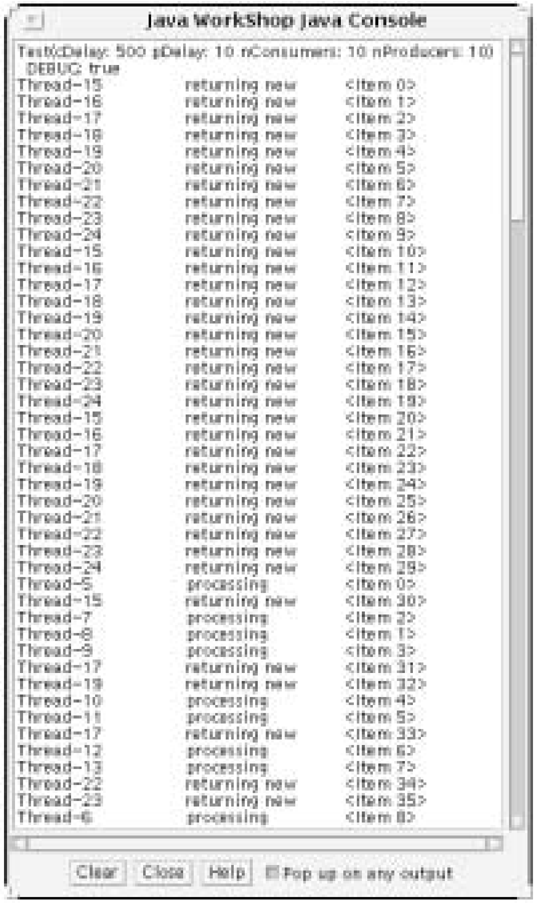
**NB: Which Project**

# Java Tools (Sun’s JWS 2.0)


Java WorkShop - Test.java
File Edit Build Debug Browse Project GUI Preferences Windows Help
Build Debug String Search
Line 42 Col 1
Project: OneQueueSolution

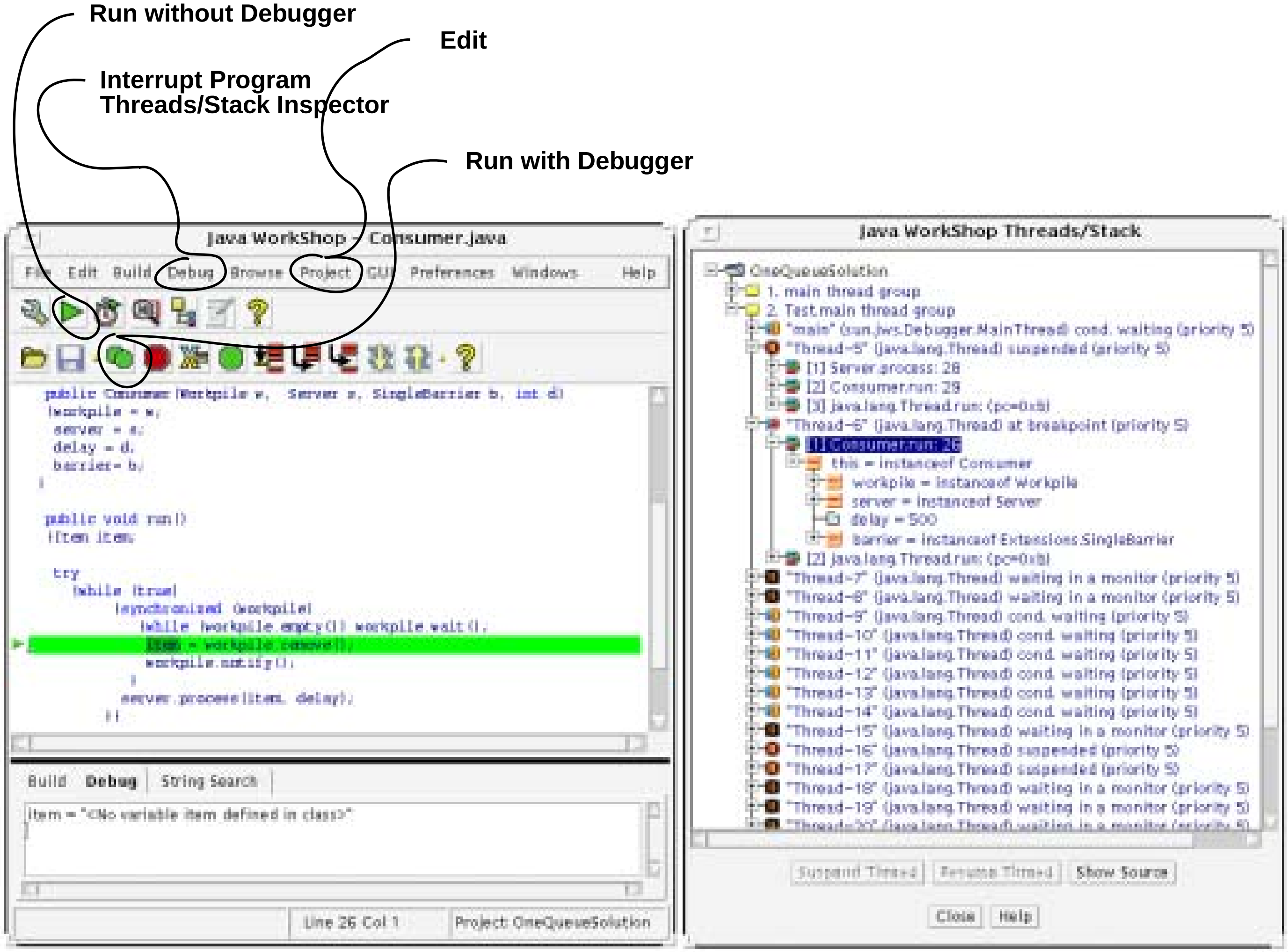
```
    try
      {for (int i=0; i<nConsumers; i++)
           {t = new Thread(new Consumer(w, s, b, cDelay));
            t.start();
           }

         for (int i=0; i<nProducers; i++)
           {t = new Thread(new Producer(w, s, b, pDelay));
            t.start();
           }

         Thread.sleep(10000);
         System.out.println("Finished!  Produced: " + s.pcounter
                           + " Consumed: " + s.ccounter
                           + " items. List length: " + w.length);
         System.exit(0);

       }
    catch (InterruptedException ie) {System.out.println(ie + " in main()");}
   }
 }
```

```
ns:/disk2/6/Java/jws2/Java-WorkShop2.0/JDK/lib/classes.zip:/disk2/6/Java/jws2/Java-WorkShc
p2.0/JWS/classes Consumer.java Item.java Producer.java Server.java Stopper.java Test.java
Workpile.java

Done compiling Java file(s) in OneQueueSolution: succeeded.
Done building OneQueueSolution.
```



```
Test(cDelay: 500 pDelay: 10 nConsumers: 10 nProducers: 10)
  DEBUG: true
Thread-15          returning new      <Item 0>
Thread-16          returning new      <Item 1>
Thread-17          returning new      <Item 2>
Thread-18          returning new      <Item 3>
Thread-19          returning new      <Item 4>
Thread-20          returning new      <Item 5>
Thread-21          returning new      <Item 6>
Thread-22          returning new      <Item 7>
Thread-23          returning new      <Item 8>
Thread-24          returning new      <Item 9>
Thread-15          returning new      <Item 10>
Thread-16          returning new      <Item 11>
Thread-17          returning new      <Item 12>
Thread-18          returning new      <Item 13>
Thread-19          returning new      <Item 14>
Thread-20          returning new      <Item 15>
Thread-21          returning new      <Item 16>
Thread-22          returning new      <Item 17>
Thread-23          returning new      <Item 18>
Thread-24          returning new      <Item 19>
Thread-15          returning new      <Item 20>
Thread-16          returning new      <Item 21>
Thread-17          returning new      <Item 22>
Thread-18          returning new      <Item 23>
Thread-19          returning new      <Item 24>
Thread-20          returning new      <Item 25>
Thread-21          returning new      <Item 26>
Thread-22          returning new      <Item 27>
Thread-23          returning new      <Item 28>
Thread-24          returning new      <Item 29>
Thread-5           processing         <Item 0>
Thread-15          returning new      <Item 30>
Thread-7           processing         <Item 2>
Thread-8           processing         <Item 1>
Thread-9           processing         <Item 3>
Thread-17          returning new      <Item 31>
Thread-19          returning new      <Item 32>
Thread-10          processing         <Item 4>
Thread-11          processing         <Item 5>
Thread-17          returning new      <Item 33>
Thread-12          processing         <Item 6>
Thread-13          processing         <Item 7>
Thread-22          returning new      <Item 34>
Thread-23          returning new      <Item 35>
Thread-6           processing         <Item 8>
```

# Java Tools (Sun’s JWS 2.0)



```
public Consumer(Workpile w,  Server s, SingleBarrier b, int d)
{workpile = w;
 server = s;
 delay = d;
 barrier= b;
}

public void run()
{Item item;

 try
   {while (true)
       {synchronized (workpile)
          {while (workpile.empty()) workpile.wait();
           item = workpile.remove();
           workpile.notify();
          }
        server.process(item, delay);
      }}
```

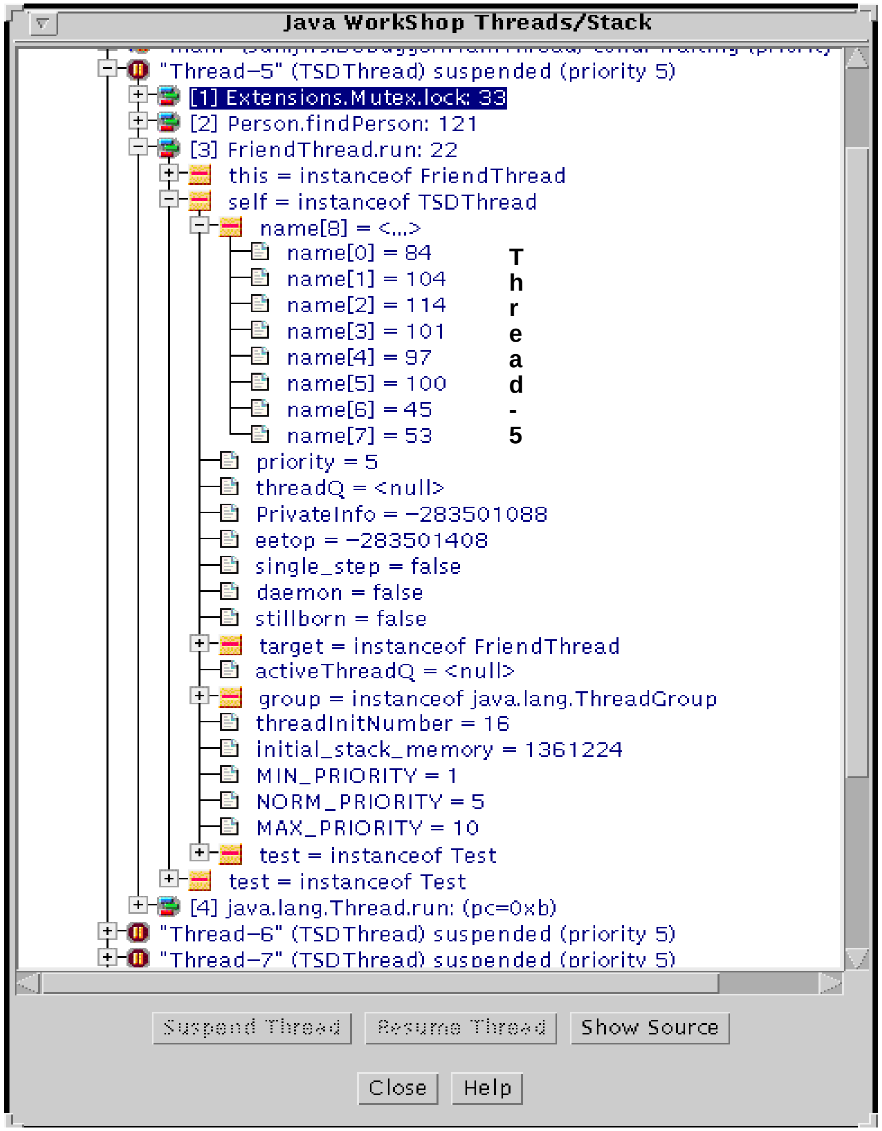
```
item = "<No variable item defined in class>"
```

Line 26 Col 1

Project: OneQueueSolution

```
OneQueueSolution
  1. main thread group
  2. Test.main thread group
    "main" (sun.jws.Debugger.MainThread) cond. waiting (priority 5)
    "Thread-5" (java.lang.Thread) suspended (priority 5)
      [1] Server.process: 28
      [2] Consumer.run: 29
      [3] java.lang.Thread.run: (pc=0xb)
    "Thread-6" (java.lang.Thread) at breakpoint (priority 5)
      [1] Consumer.run: 26
        this = instanceof Consumer
          workpile = instanceof Workpile
          server = instanceof Server
          delay = 500
          barrier = instanceof Extensions.SingleBarrier
      [2] java.lang.Thread.run: (pc=0xb)
    "Thread-7" (java.lang.Thread) waiting in a monitor (priority 5)
    "Thread-8" (java.lang.Thread) waiting in a monitor (priority 5)
    "Thread-9" (java.lang.Thread) cond. waiting (priority 5)
    "Thread-10" (java.lang.Thread) cond. waiting (priority 5)
    "Thread-11" (java.lang.Thread) cond. waiting (priority 5)
    "Thread-12" (java.lang.Thread) cond. waiting (priority 5)
    "Thread-13" (java.lang.Thread) cond. waiting (priority 5)
    "Thread-14" (java.lang.Thread) cond. waiting (priority 5)
    "Thread-15" (java.lang.Thread) waiting in a monitor (priority 5)
    "Thread-16" (java.lang.Thread) suspended (priority 5)
    "Thread-17" (java.lang.Thread) suspended (priority 5)
    "Thread-18" (java.lang.Thread) waiting in a monitor (priority 5)
    "Thread-19" (java.lang.Thread) waiting in a monitor (priority 5)
    "Thread-20" (java.lang.Thread) waiting in a monitor (priority 5)
```

# Java Tools



```
"Thread-5" (TSDThread) suspended (priority 5)
  [1] Extensions.Mutex.lock: 33
  [2] Person.findPerson: 121
  [3] FriendThread.run: 22
    this = instanceof FriendThread
    self = instanceof TSDThread
      name[8] = <...>
        name[0] = 84
        name[1] = 104
        name[2] = 114
        name[3] = 101
        name[4] = 97
        name[5] = 100
        name[6] = 45
        name[7] = 53
      priority = 5
      threadQ = <null>
      PrivateInfo = -283501088
      eetop = -283501408
      single_step = false
      daemon = false
      stillborn = false
      target = instanceof FriendThread
      activeThreadQ = <null>
      group = instanceof java.lang.ThreadGroup
      threadInitNumber = 16
      initial_stack_memory = 1361224
      MIN_PRIORITY = 1
      NORM_PRIORITY = 5
      MAX_PRIORITY = 10
      test = instanceof Test
    test = instanceof Test
  [4] java.lang.Thread.run: (pc=0xb)
"Thread-6" (TSDThread) suspended (priority 5)
"Thread-7" (TSDThread) suspended (priority 5)
```

Thread-5

# POSIX Function Prototypes

```
error pthread_create(&tid, &attribute, my_sigwaiter, arg);

error pthread_join(tid, &status);

pthread_exit(status);

tid pthread_self();

int pthread_equal(tid1, tid2);

error pthread_cancel(tid);

sched_yield();
```

# Java Function Prototypes

```
Thread thread = new Thread();
Thread thread = new Thread(String name);
Thread thread = new Thread(Runnable runnable);
Thread thread = new Thread(Runnable runnable, String name);
boolean thread.isAlive();
void thread.start();
void thread.run();
void runnable.run();

void thread.setName(String name);
String thread.getName();

void thread.join();
void thread.join(long milliseconds);
void thread.join(long milliseconds, int nanoseconds);

void thread.stop();

Thread t = Thread.currentThread();

Thread.sleep(long milliseconds);
Thread.sleep(long milliseconds, int nanoseconds);

void thread.stop();

void thread.yield();
```

# *Synchronization*

# Unsynchronized Shared Data is a Formula for Disaster

```
              Thread 1                                   Thread 2

temp = Your.BankBalance;               temp = Your.BankBalance;
dividend = temp * InterestRate;        newBalance = deposit + temp;
newBalance = dividend + temp;          Your.BankBalance = newBalance;
Your.BankBalance = newBalance;
```

**Synchronization is not just an MP issue. It is not even strictly an MT issue!**

# Atomic Actions

- **An action which must be started and completed with no possibility of interruption.**
  - **A machine instruction could need to be atomic. (not all are!)**
  - **A line of code could need to be atomic. (not all are)**
  - **An entire database transaction could need to be atomic.**

- **All MP machines provide at least one complex atomic instruction (*test-and-set*), from which you can build anything.**
- **A section of code which you have written to be atomic is a *Critical Section.***

# The Synchronization Instruction

- **All machines need an atomic *test and set* instruction which will:**
    - **Load (or compare) the current value of a word in main memory**
    - **and set that word to a known value**
    - ***atomically***
- **SPARC (v7, v8, v9) has the instruction `ldstub` which will read a byte from memory and write all ones into it, atomically.**
- **This is the basis for all synchronization variables (inc. Java).**
- **SPARC v9 (UltraSPARC) also implements CAS (*compare and swap if equal)*, which can be used to do tricky things.**
- **DEC Alpha and MIPS use `LockedLoad` which is logically equivalent to `CAS`. (And even more tricky!)**

# Pseudo Code for Mutex

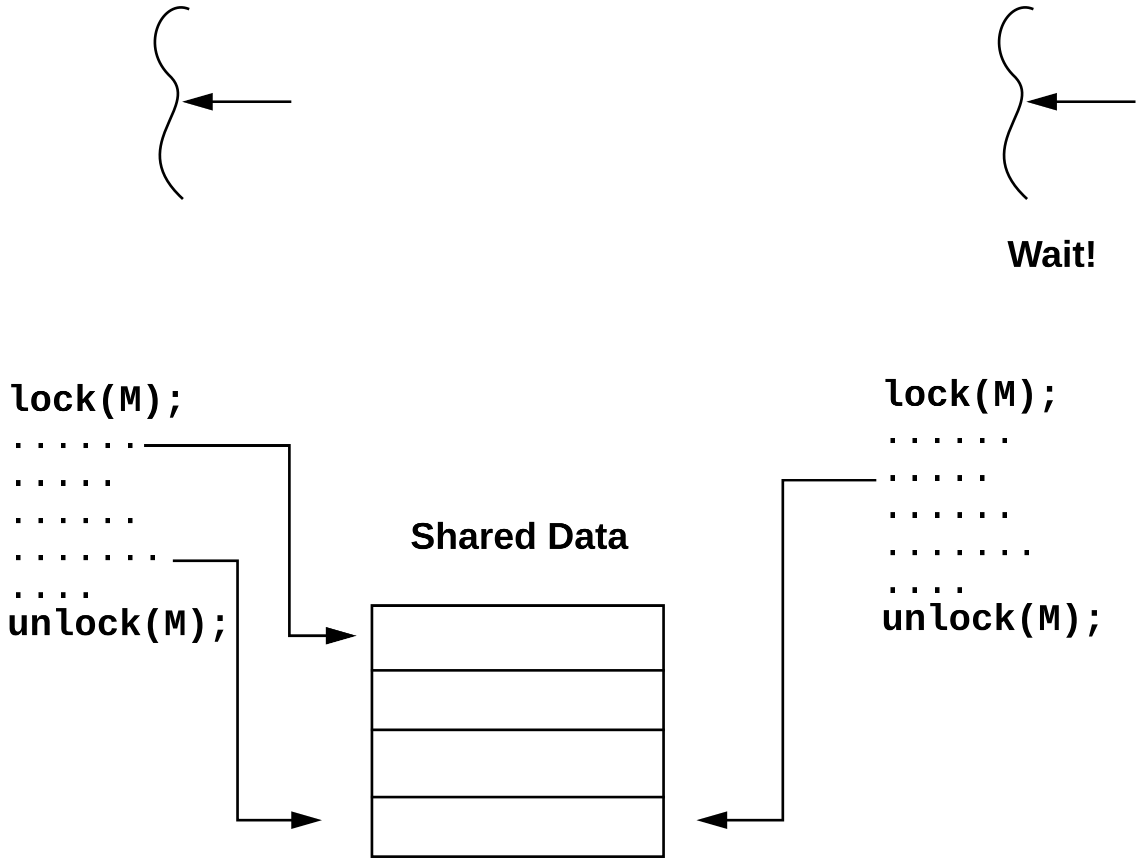
```
try:     ldstub address, register
         cmp register, 0
         beq got_it
         call go_to_sleep
         jmp try
got_it:  return
```

## *“Mutual Exclusion” Lock*

**All synchronization in all languages on all machines for every OS is based on this basic mutex code.** ***All synchronization.***

**You should make yourself 100% comfortable with this operation.**

# Critical Sections
# (Good Programmer)



Wait!

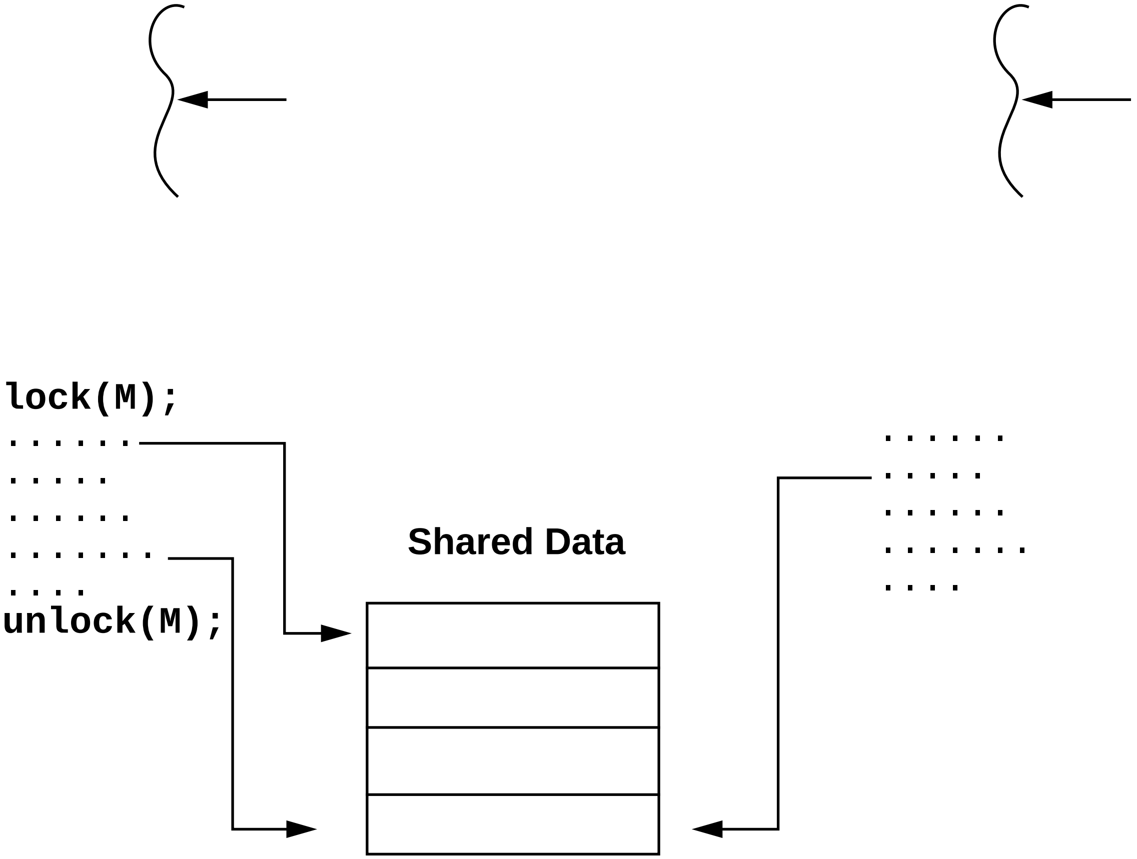
```
lock(M);
......
.....
......
.......
....
unlock(M);
```

Shared Data

```
lock(M);
......
.....
......
.......
....
unlock(M);
```

**(Of course you must know *which* lock protects *which* data!)**

# Critical Sections
# (Bad Programmer)



```
lock(M);
......
.....
......
.......
....
unlock(M);
```

```
......
.....
......
.......
....
```

# Lock Shared Data!

- Globals
- Shared data structures
- Static variables
  (really just lexically scoped global variables)

**Wrong** **Right**

```
foo(int j)            foo(int j)
{ static int i = 0;   { static int i = 0;
                        static pthread_mutex_t m=P*_MUTEX_INITIALIZER

  i+=j;                 pthread_mutex_lock(&m);
  return(i);            i+=j;
}                       j=i;
                        pthread_mutex_unlock(&m);
                        return(j);
                      }
```

# Synchronization Variables

Each of the libraries implement a set of “synchronization variables” which allow different threads to coordinate activities. The actual variables are just normal structures in normal memory*. The functions that operate on them have a little bit of magic to them.

UI: Mutexes, Counting Semaphores, Reader/Writer Locks, Condition Variables, and Join.

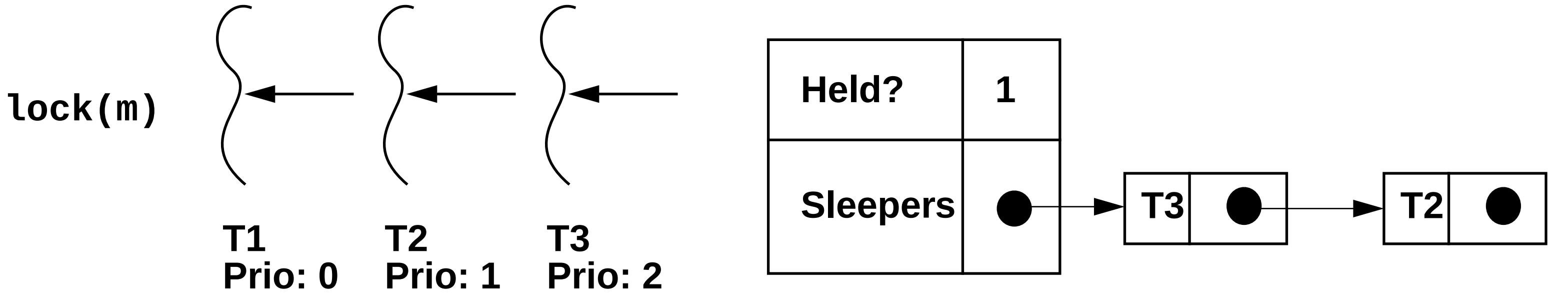
POSIX: Mutexes, Counting Semaphores, Condition Variables, and Join.

Java: Synchronized, Wait/Notify, and Join.

Win32: Mutexes, Event Semaphores, Counting Semaphores, “Critical Sections.” and Join (WaitForObject).

* In an early machine, SGI actually built a special memory area.

# Mutexes

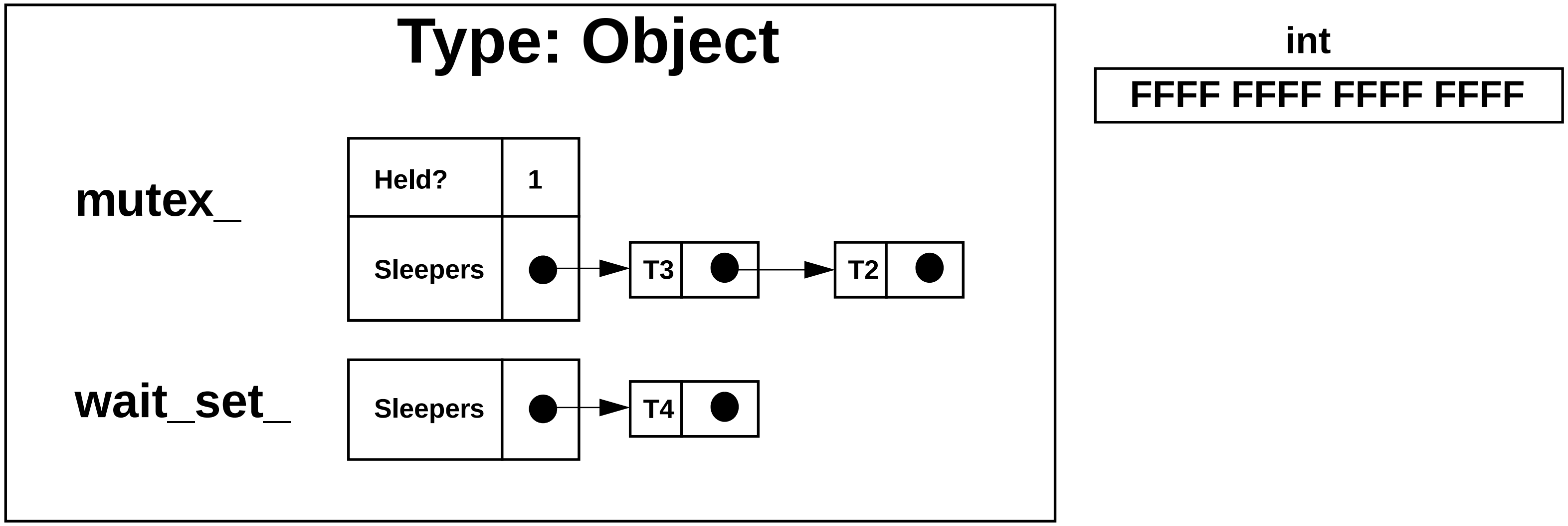


| POSIX | Win32 | Java |
|---|---|---|
| `pthread_mutex_lock(&m)` | `WaitForSingleObject(m)` | `synchronized(obj)` |
| `...` | `...` | `{...` |
| `pthread_mutex_unlock(&m)` | `ReleaseMutex(m)` | `}` |

- **POSIX and UI: Owner not recorded, Block in priority order, Illegal unlock not checked at runtime.**
- **Win32: Owner recorded, Block in FIFO order**
- **Java: Owner recorded (not available), No defined blocking order, Illegal Unlock impossible**

# Java Locking

The class Object (and thus everything that subclasses it -- e.g., everything except for the primitive types: int, char, etc.) have a hidden mutex and a hidden “wait set”:



# Java Locking

The hidden mutex is manipulated by use of the `synchronized` keyword:

```
public MyClass() {
int count=0;

  public synchronized void increment()
  {count++;}

}
```

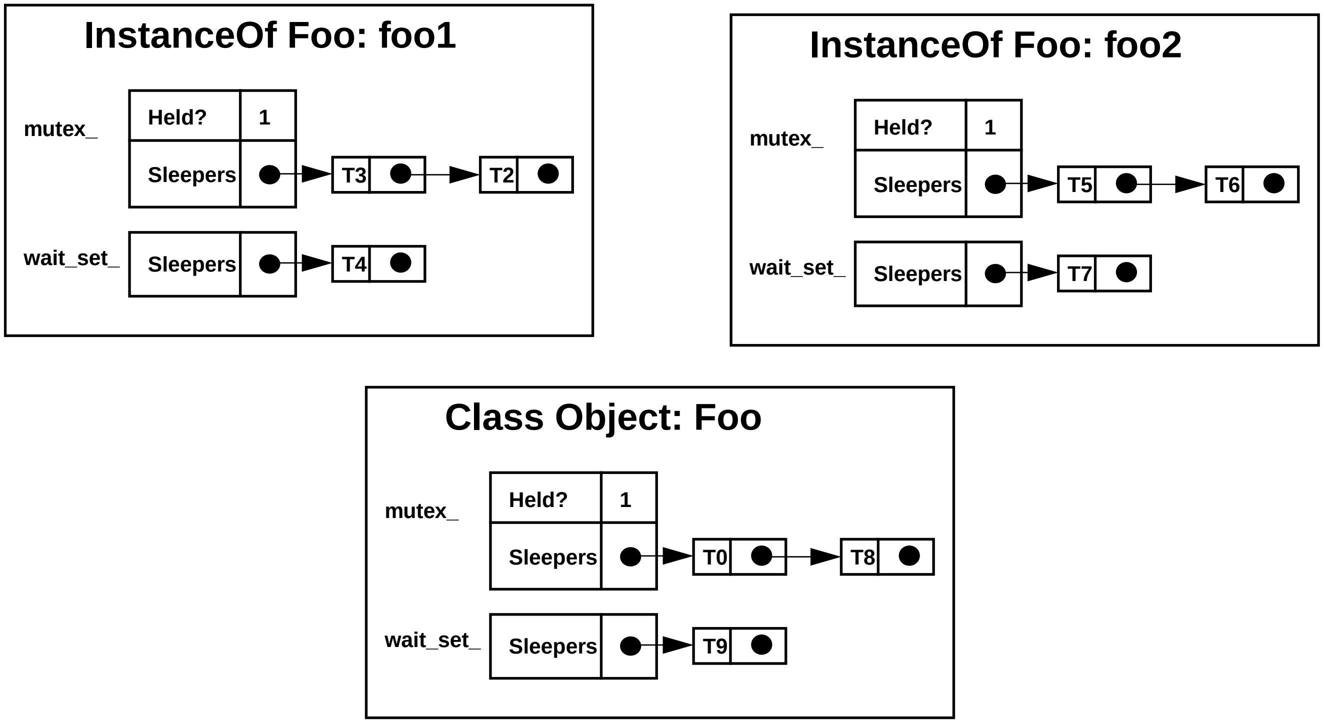
or explicitly using the object:

```
public MyClass() {
int count=0;

  public void increment() {
    synchronized (this)
      {count++;}
  }
}
```

Thus in Java, you don’t have to worry about unlocking. That’s done for you! Even if the thread throws an exception or exits.

# Each Instance Has Its Own Lock

**Thus each instance has its own lock and wait set which can be used to protect.... anything you want to protect! (Normally you’ll be protecting data in the current object.) Class Objects are Objects...**



InstanceOf Foo: foo1

mutex_ Held? 1

Sleepers → T3 → T2

wait_set_ Sleepers → T4

InstanceOf Foo: foo2

mutex_ Held? 1

Sleepers → T5 → T6

wait_set_ Sleepers → T7

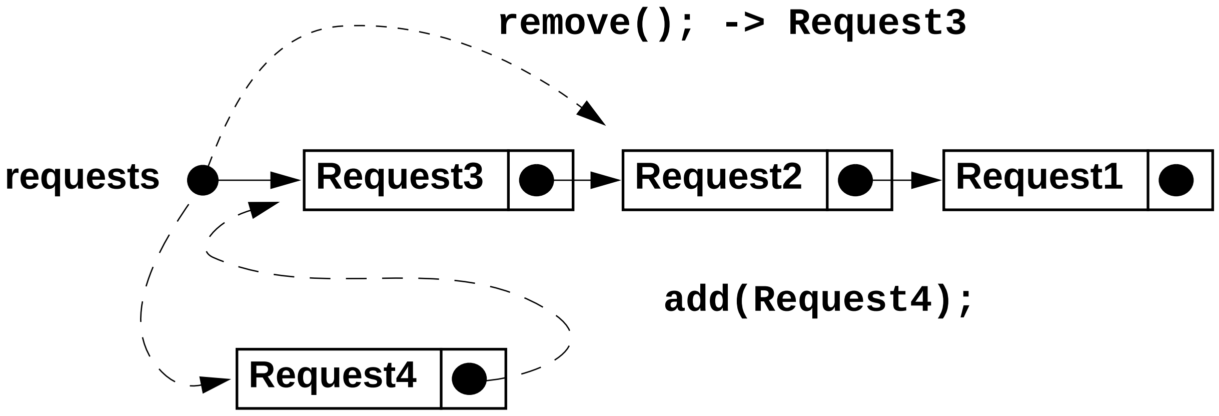
Class Object: Foo

mutex_ Held? 1

Sleepers → T0 → T8

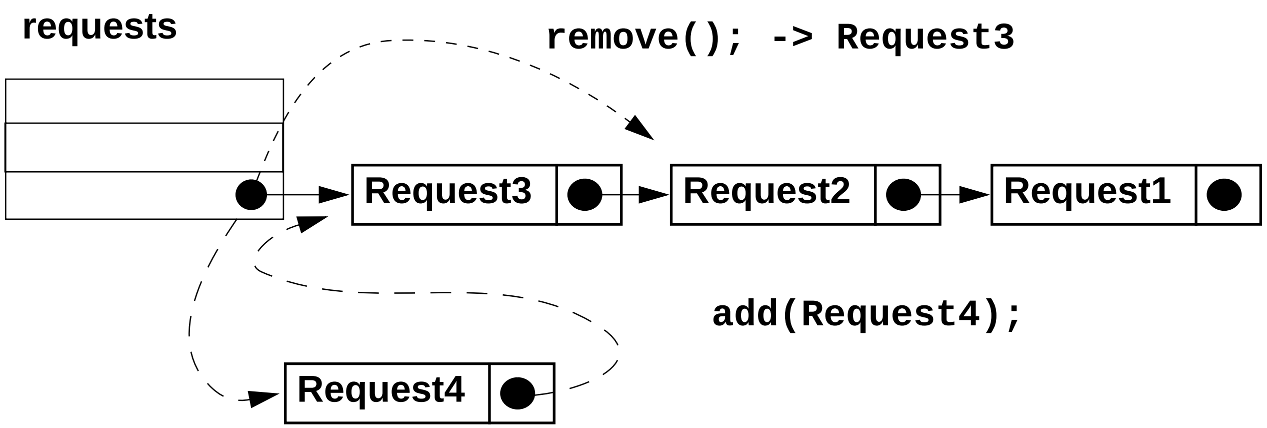
wait_set_ Sleepers → T9

# Using Mutexes



```
Thread 1                            Thread 2

add(request_t *request)
{ pthread_mutex_lock(&req_lock);
  request->next = requests;         request_t *remove()
  requests = request;               { pthread_mutex_lock(&req_lock);
  pthread_mutex_unlock(&req_lock)     ...sleeping...
}
                                      request = requests;
                                      requests = requests->next;
                                      pthread_mutex_unlock(&req_lock);
                                      return(request);
                                    }
```

# Using Java Locking



requests

remove(); -> Request3

Request3

Request2

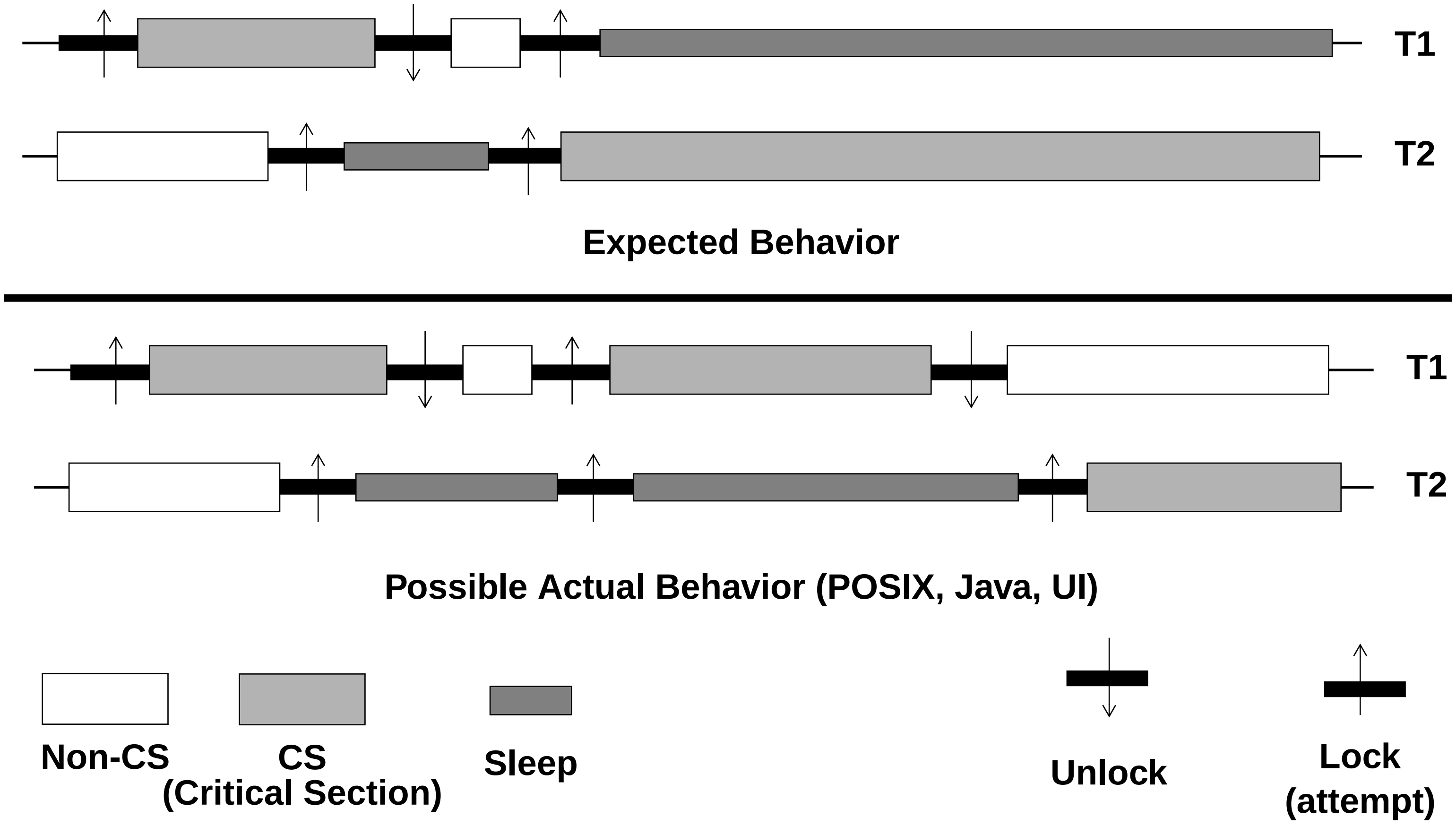
Request1

add(Request4);

Request4

```
Thread 1                               Thread 2

add(Request request) {
  synchronized(requests)
  {request.next = requests;            Request remove() {
   requests = request;                   synchronized(requests)
  }                                      ...sleeping...
}
                                         {request = requests;
                                          requests = requests.next;
                                         }
                                         return(request);
                                       }
```

# Mutex Execution Graph



T1

T2

Expected Behavior

T1

T2

Possible Actual Behavior (POSIX, Java, UI)

Non-CS

CS
(Critical Section)

Sleep

Unlock

Lock
(attempt)

**The blocked thread may not get the mutex in POSIX, Java, and UI.**

**Typical lock/unlock: ~2µs; Critical Section: 10+ µs; Wo > 1ms**
**Typical JavaVM synchronized(o){}: ~4µs**

# Win32 Lingo

In POSIX we refer to a mutex being “locked” or “owned” and “unlocked” or “unowned”. A semaphore has a value (0 or more). A condition variable just is.

In Win32, they use slightly different (and perhaps confusing) terminology. Everything is either “signaled” or “unsignaled”. Signaled means that `WaitForSingleObject(object)` will not block, unsignaled means that it will. Hence:

A signaled mutex is an unlocked mutex.

A signaled semaphore is a semaphore with a value > 0.

A signaled event object has no equivalent in POSIX (`pthread_condwait()` always blocks).

# The Class Lock

For global variables (class “static” variables), you may need to use an unrelated object just to have a lock to protect it:

```
public class Foo
{ static int count = 0;
  static Object o = new Object();

  public void inc(int i)
  {synchronized (o){count = count + i;}}
}
```

Or you may use the Class itself:

```
public class Foo
{ static int count = 0;

  public void inc(int i)
  {synchronized (Foo.class){count = count + i;}} // cf: getClass()
}
```

Static synchronized methods also use the class lock:

```
public class Foo
{ static int count = 0;

  static public synchronized void inc(int i)
  {count = count + i;}
}
```

# Static Methods of Subclasses

The `this` object of a static method (if it were defined) would always be the class in which the method was defined, not the subclass by which it was called. (Catch that?)

In other words, this code is correct:

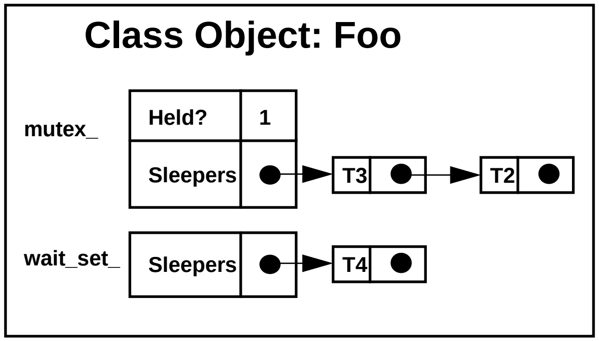
```
public class Foo
{ static int count = 0;

  static public synchronized void inc(int i)
  {count = count + i;}
}

public class SubFoo extends Foo
{...}                // No changes to count or inc here



  Thread 1                                      Thread 2

SubFoo.inc(1);                                Foo.inc(1);
```

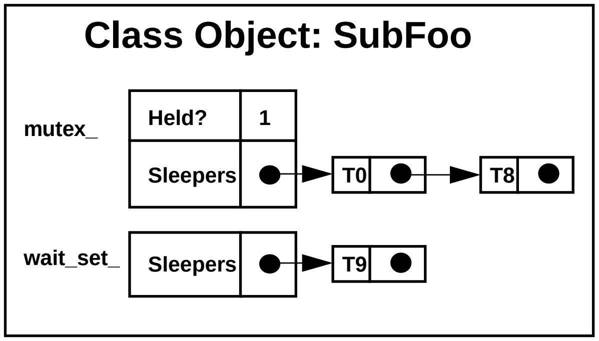
Both calls will lock the lock on the class object Foo, not SubFoo. Both will increment the same `count` too.

# Don’t Get the Wrong Class Lock

**The trick comes if you try to call `getClass()`, which will get the class of the current object (of course). If this happens to be a subclass of the class you’re interested it, you would get the wrong lock:**

```
public class Foo {
  static int count = 0;

public void inc(int i) {
  synchronized(this.getClass()) {
    count++;
  }
}

public class SubFoo extends Foo
{...}
```





```
  Thread 1                  Thread 2

SubFoo.inc(1);          Foo.inc(1);
```

# Subclasses May Change Declaration

**A method declared `synchronized` in a class, may be specialized in the subclass and NOT declared `synchronized`:**

```
public class Foo()
{...
  public synchronized void frob()
  {...}
}

public class Bar() extends Foo
{...
  public void frob()
  {...}
}
```

***Don’t do that!***

# pthread_mutex_trylock()

**There are special circumstances when you do not want to go to sleep, waiting for the mutex to become free. You don’t have to.**

**POSIX**

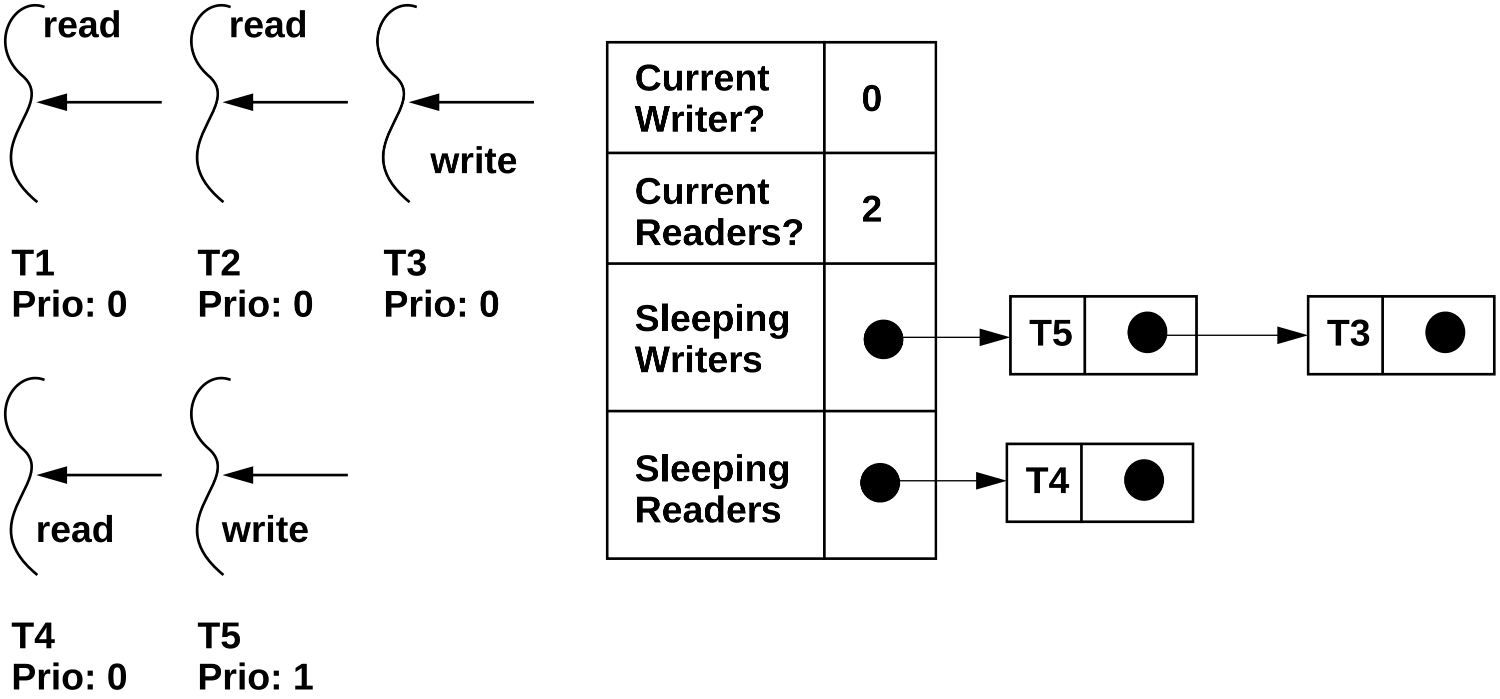
```
if(pthread_mutex_trylock(&m)
   ==EBUSY)

...
```

**Win32**

```
if(WaitForSingleObject(m,timeout)
    == WAIT_TIMEOUT)

  ...
```

**Nothing similar in Java using synchronized objects. You could build equivalent functionality (see “Sometimes Java Needs Mutexes” slide 138), but don’t.**

# Readers/Writer Locks



T1 Prio: 0 — read

T2 Prio: 0 — read

T3 Prio: 0 — write

T4 Prio: 0 — read

T5 Prio: 1 — write

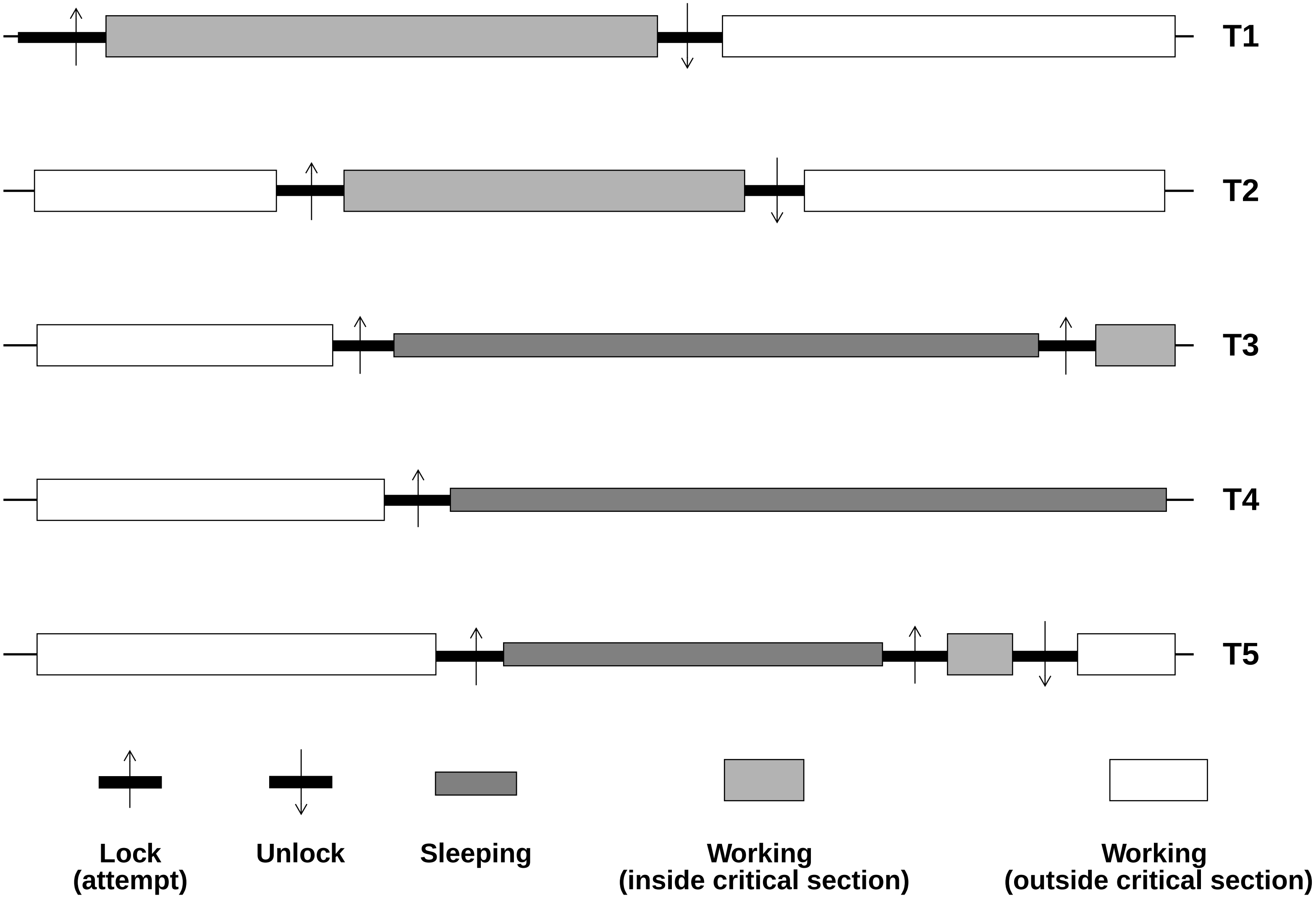
| | |
|---|---|
| Current Writer? | 0 |
| Current Readers? | 2 |
| Sleeping Writers | ● → T5 ● → T3 ● |
| Sleeping Readers | ● → T4 ● |

- Only defined in UI threads & UNIX98, easily built in the others.
- Sample code in thread_extensions.c / Extensions.java

# Using RWLocks

```
void *give_friends_raise(person_t *friends)
{
  while(friends != NULL)
    {thread_rw_rdlock(&people_lock);
     p = find_person(friends->first, people);
     thread_rw_unlock(&people_lock);
     give_raise(p);
     friends = friends->next;
   }}

public void run() {
  while (friends != null) {
     rwlock.readLock();
     p = findPerson(friends.first, people);
     rwlock.unlock();
     p.giveRaise();
     friends = friends.next;
    }}
```

# RW Lock Execution Graph



T1

T2

T3

T4

T5

Lock (attempt)

Unlock

Sleeping

Working (inside critical section)

Working (outside critical section)

# RW Locks are Rarely Used

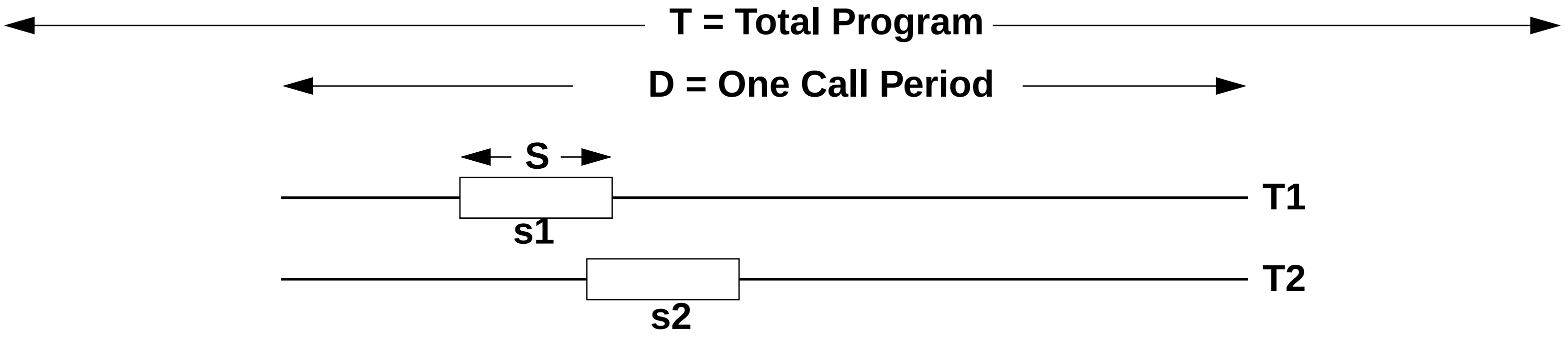
- Requires very large critical sections
- Requires many reads and very few writes
- Requires contention for the lock

- Rule of thumb:
  - > 100 reads per write (?)
  - Critical section > 100µs
  - Read contention > 10%

- E.g., unsorted phonebook lookup

- If you are writing high-performance code, write it both ways and compare. (Then tell me. I’d like to know!)

# RW Analysis



From the figure, you can see that contention for the critical section (S) exists whenever s1 and s2 overlap (S >> M). The probability of contention is thus 2S/D. The total number of times the section will be locked per thread is (T/D) = L. The expected number of contentions is therefore (2S/D) * L = C, while the expected number of noncontention locks is L - C = N. The cost of blocking and then unblocking for a mutex is 30M (empirically determined), where M is the cost of calling mutex_lock() successfully. The expected cost of running the critical sections of the program is:

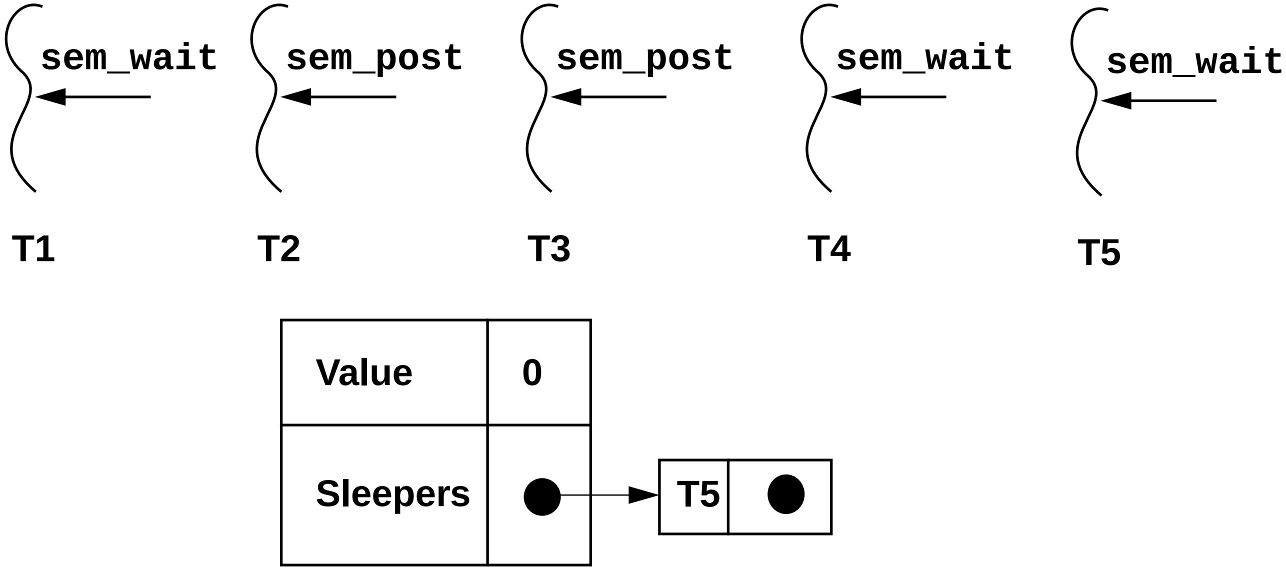
**with mutexes:** T(m) = (N * (M + S)) + (C * (30 * M + S))
(The first term is the cost of the noncontention locks, the second is the cost of the contention locks.)

**with RW locks:** T(r) = (N + C) * (3 * M + S)

The question now becomes “When is T(r) < T(m)?”

A bit of algebra gives us the answer: “When S/D > 4%.”

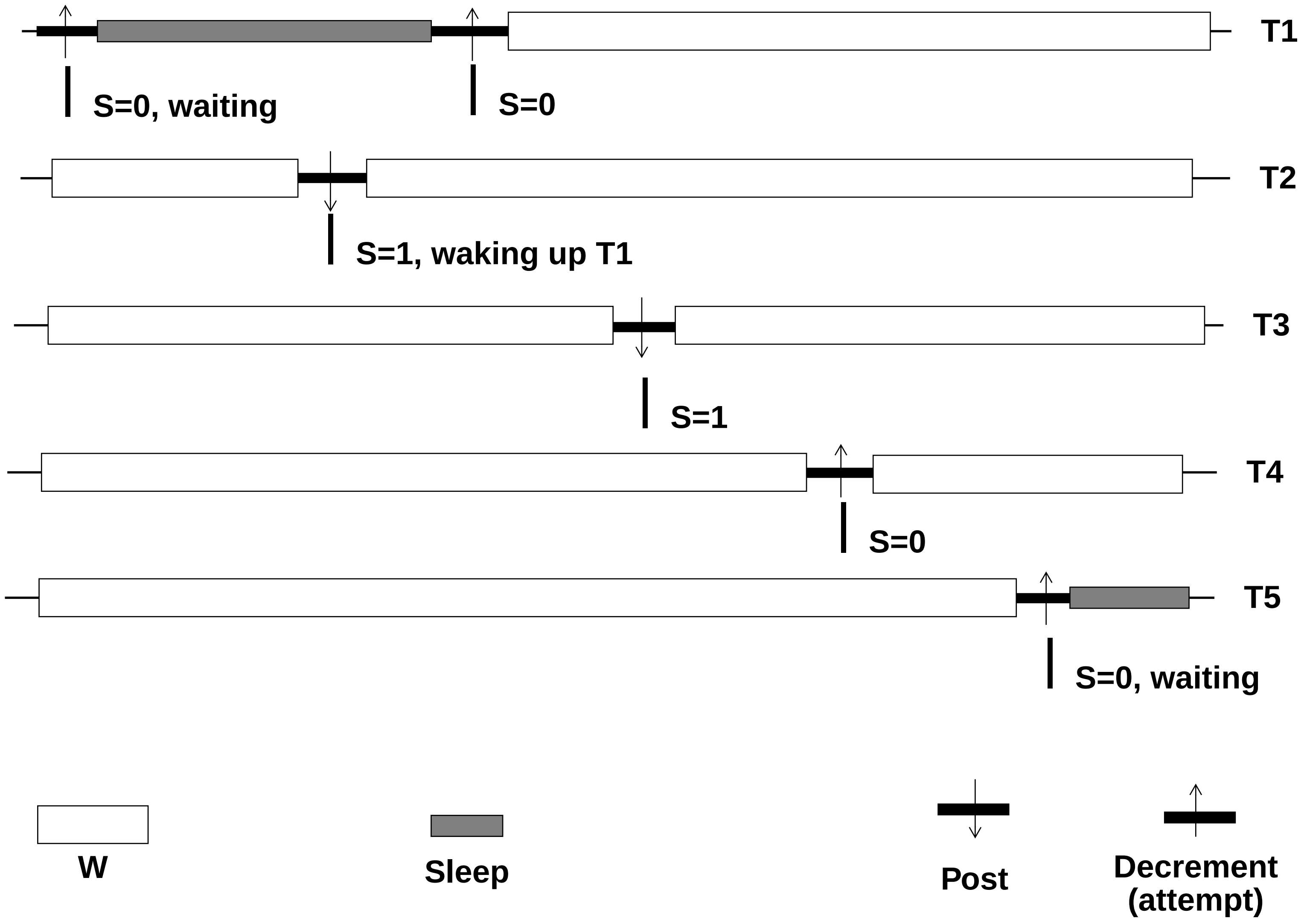
# Counting Semaphores



```
POSIX                 Win32                              Java (Buildable)

{...                  {...                               {...
sem_wait(&s);         WaitForSingleObject(s,...);        s.semWait();
...                   ...                                ...
sem_post(&s);         ReleaseSemaphore(s,...);           s.semPost();
}                     }                                  }
```

**There is no concept of an “owner” for semaphores.**

# Semaphore Execution Graph



# Inserting Items Onto a List

```
request_t *get_request()
{request_t *request;
  request = (request_t *) malloc(sizeof(request_t));
  request->data = read_from_net();
  return(request)
}

void process_request(request_t *request)
{ process(request->data);
  free(request);
}

producer()
{request_t *request;
 while(1)
  {request = get_request();
   add(request);
   sem_post(&requests_min);
  }
}

consumer()
{request_t *request;
 while(1)
  {while(sem_wait(&requests_min) == -1) {}   /* Do nothing if EINTR */
   request = remove();
   process_request(request);
  }
}
```

# Inserting Items Onto a List (Java)

```
public class Consumer implements Runnable {
public void run() {
  while (true) {
           workpile.semaphore.semWait()
           request = workpile.remove();
           server.process(request);
    }
}
}
```

```
public class Producer implements Runnable {
public void run() {
  while (true) {
    request = server.get();
    workpile.add(request);
    workpile.semaphore.semPost();
    }
}
}
```

# EINTR

In POSIX and UI, semaphores and condition variables can be interrupted by either a signal (or a call to `fork()`, UI), in which case they will return a value to indicate this. UI semaphores and CVs return `EINTR`. POSIX CVs just return `0,` but you’re retesting anyway. POSIX semaphores return `-1` and set `errno` to `EINTR*`.

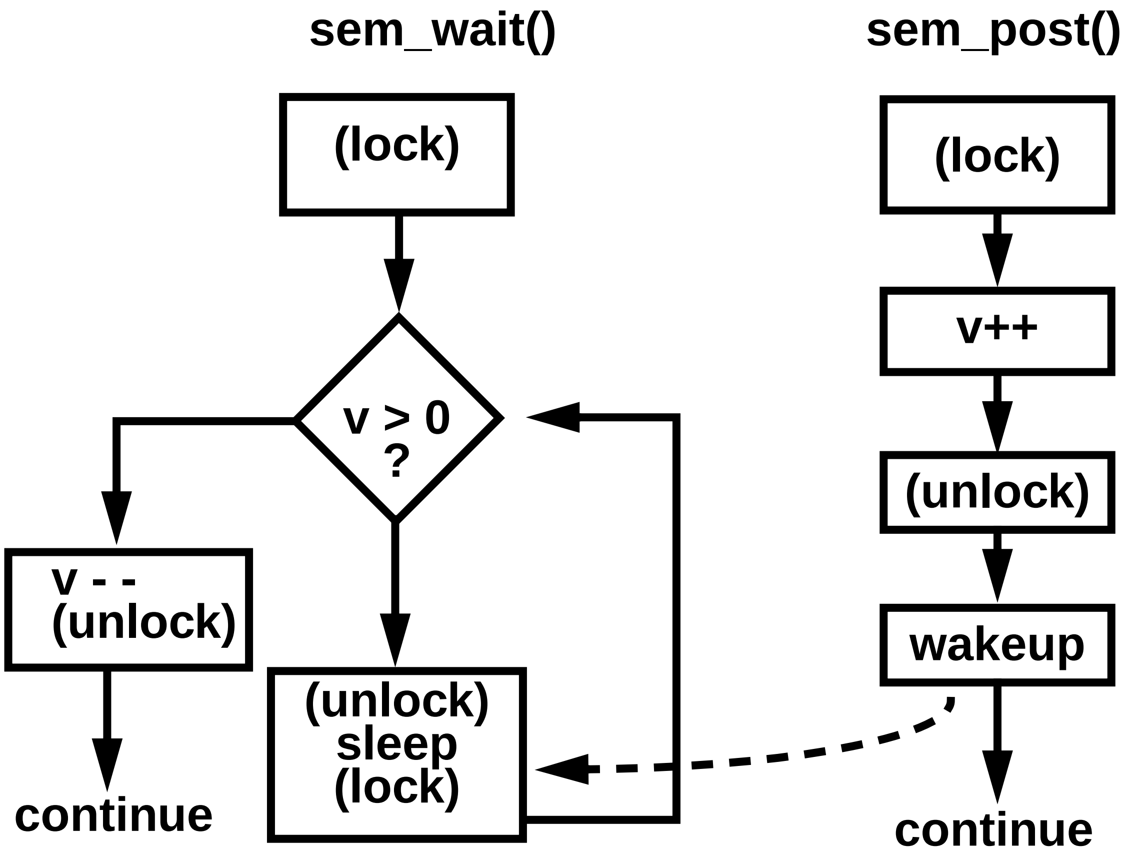
Therefore correct usage of semaphores (and optionally UI condition variables) requires that they be executed in a loop:

```
while (sem_wait(&s) == -1) {<probably do nothing>}
do_thing();  /* NOW the semaphore is decremented! */
```

OR you can simply guarantee that the program will not receive any signals (and it will not call `fork()`, UI). And no one else who has to maintain the program will ever modify the code to do either :-)

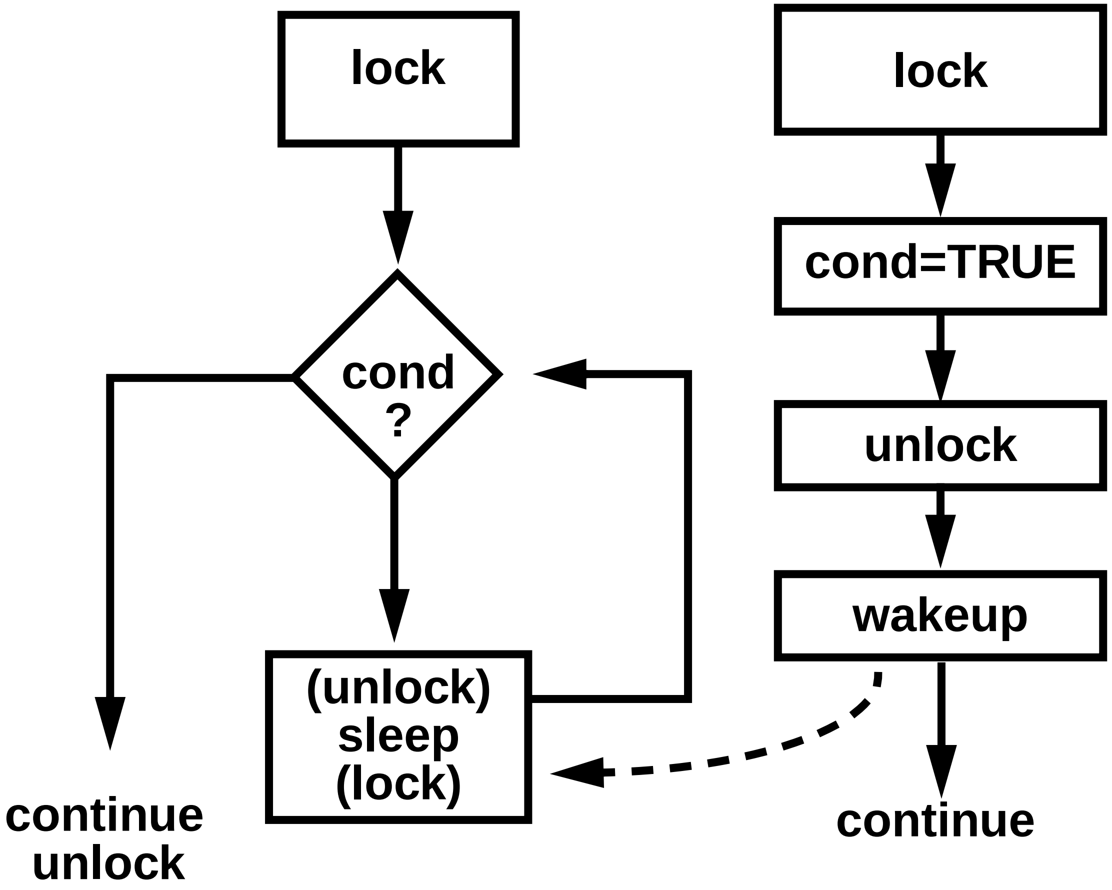
* The Solaris man pages are a bit confusing here, but it is -1.

# A Different View of Semaphores



**(This is the actual implementation on SPARC v7, v8. MIPS, Alpha, and SPARC v9 have a tricky implementation using other instructions.)**

# Generalized Condition Variable



**Win32 has “EventObjects”, which are similar.**

# Condition Variable Code

```
           Thread 1                                  Thread 2

pthread_mutex_lock(&m);
while (!my_condition)
  pthread_cond_wait(&c, &m);

                                pthread_mutex_lock(&m);
                                my_condition = TRUE;
                                pthread_mutex_unlock(&m);
                                pthread_cond_signal(&c);
                                /*  pthread_cond_broadcast(&c); */
do_thing();
pthread_mutex_unlock(&m);
```

# Java wait()/notify() Code

```
            Thread 1                                   Thread 2

synchronized (object)
  {while (!object.my_condition)
     object.wait();

                                 synchronized (object);
                                   {object.my_condition = true;
                                    object.notify();
                                 // object.notifyAll();
                                   }
   do_thing();
  }
```

**(Explicit use of synchronization)**

# Java `wait()/notify()` Code

```
              Thread 1                            Thread 2

public synchronized void foo()
{while (!my_condition)
  wait();
                                    public synchronized void bar()
                                    {
                                      my_condition = true;
                                      notify();
                                    /*  notifyAll();*/
                                    }
do_thing();
}
```

**(Implicit use of synchronization)**

**You can actually combine the explicit and implicit uses of synchronization in the same code.**

# Nested Locks Are Not Released

**If you already own a lock when you enter a critical section which you’ll be calling `pthread_cond_wait()` or Java `wait()` from, you are responsible for dealing with those locks:**

```
public synchronized void foo()
{
  synchronized(that)
     {while (!that.test()) that.wait();}
}
```

```
pthread_mutex_lock(&lock1);
pthread_mutex_lock(&lock2);
while (!test()) pthread_cond_wait(&cv, &lock2);
```

# Java Exceptions for `wait()`, etc.

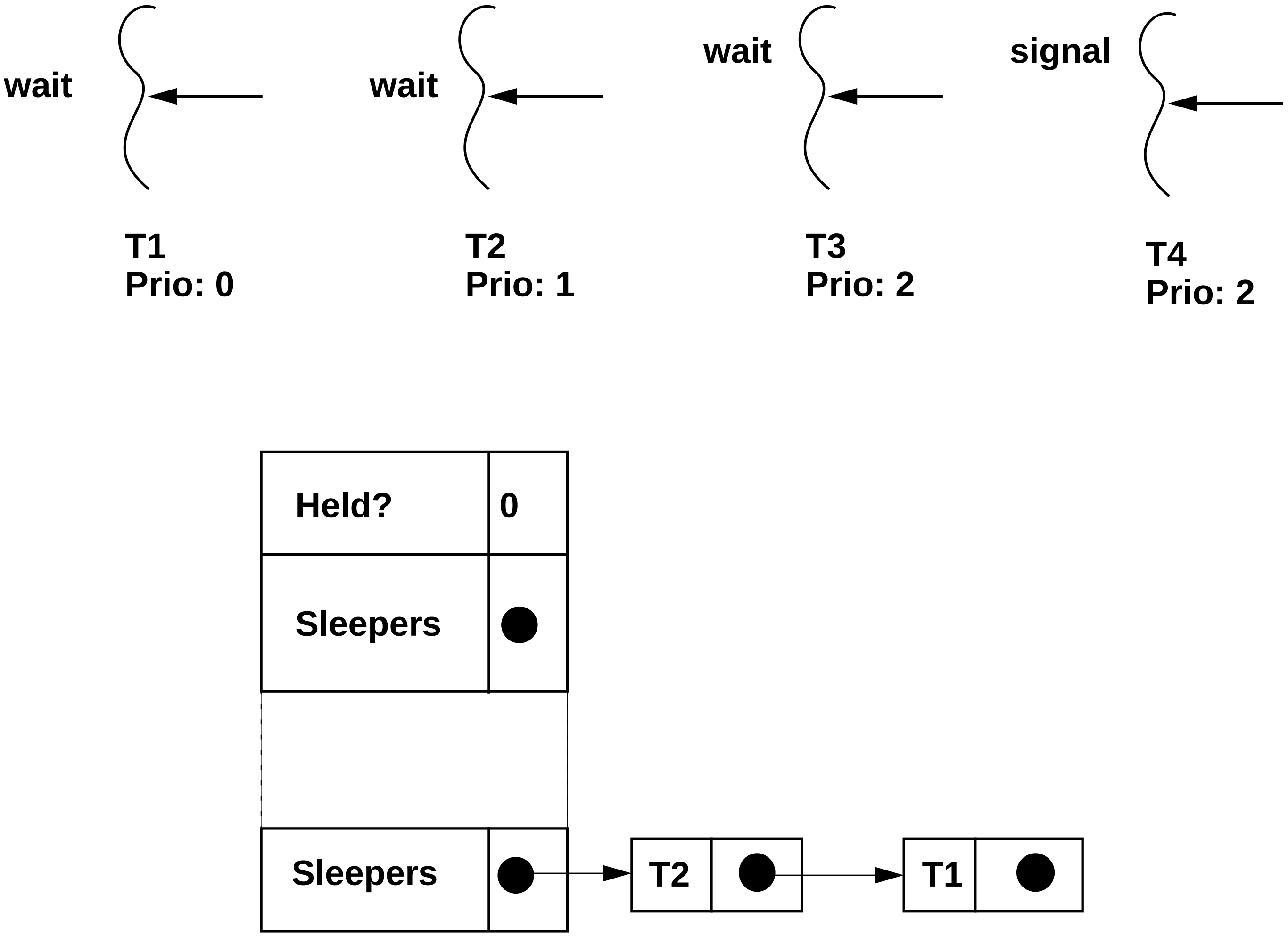
**A variety of Java threads methods throw `InterruptedException`. In particular: `join()`, `wait()`, `sleep()`. The intention is all blocking functions will throw `InterruptedIOException`.**

**This exception is thrown by our thread when another thread calls `interrupt()` on it. The idea is that these methods should be able to be forced to return should something external affect their usefulness.**

**As of Java 1.1, `thread.interrupt()` was not implemented. Anyway, all of our code must catch that exception:**

```
try {
 while (true) {
   item = server.get();
   synchronized (workpile) {
     workpile.add(item);
      workpile.wait();
     }
 }}
catch (InterruptedException ie) {}
```

# Condition Variable



wait

T1
Prio: 0

wait

T2
Prio: 1

wait

T3
Prio: 2

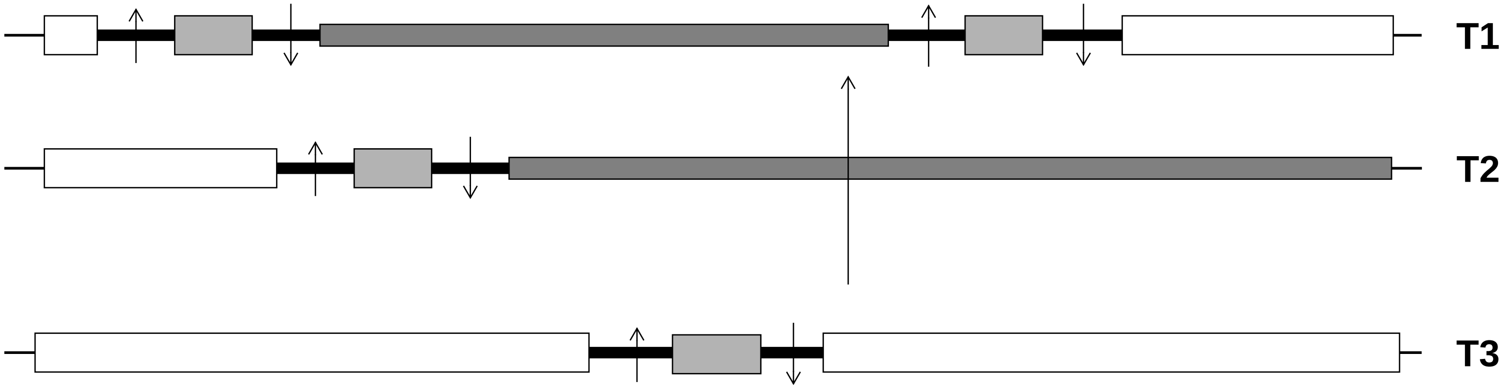
signal

T4
Prio: 2

| | |
|---|---|
| Held? | 0 |
| Sleepers | ● |

Sleepers ● → T2 ● → T1 ●

# CV Execution Graph



**Desired behavior of signal**

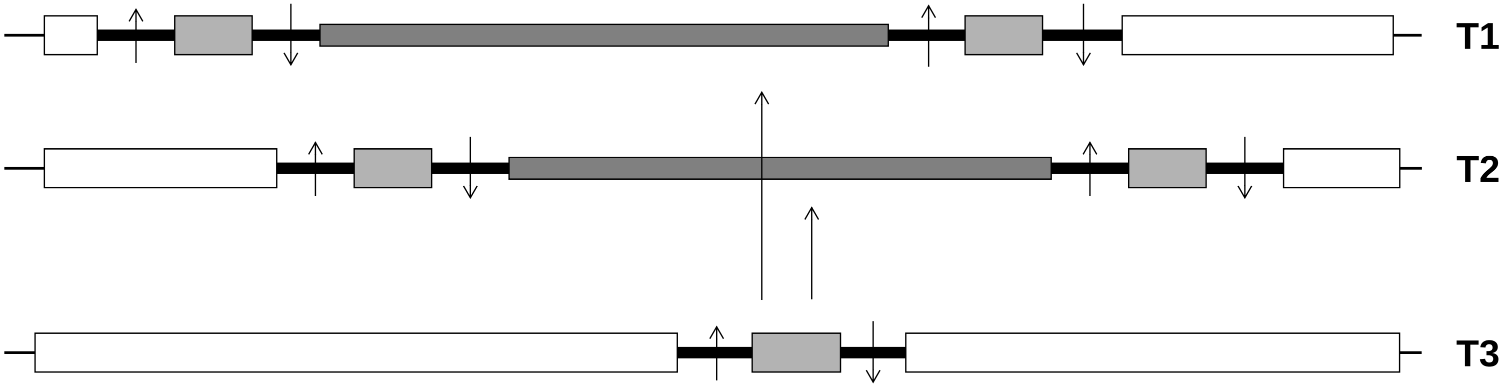
```
pthread_mutex_lock(&m);
my_condition = TRUE;
pthread_mutex_unlock(&m);
pthread_cond_signal(&c);
```

**Signal called outside of critical section.**

**(Condition signal has no relationship to UNIX signals `SIGINT`, etc.)**

**By definition, Java `notify()` must be inside the critical section.**

# CV Execution Graph



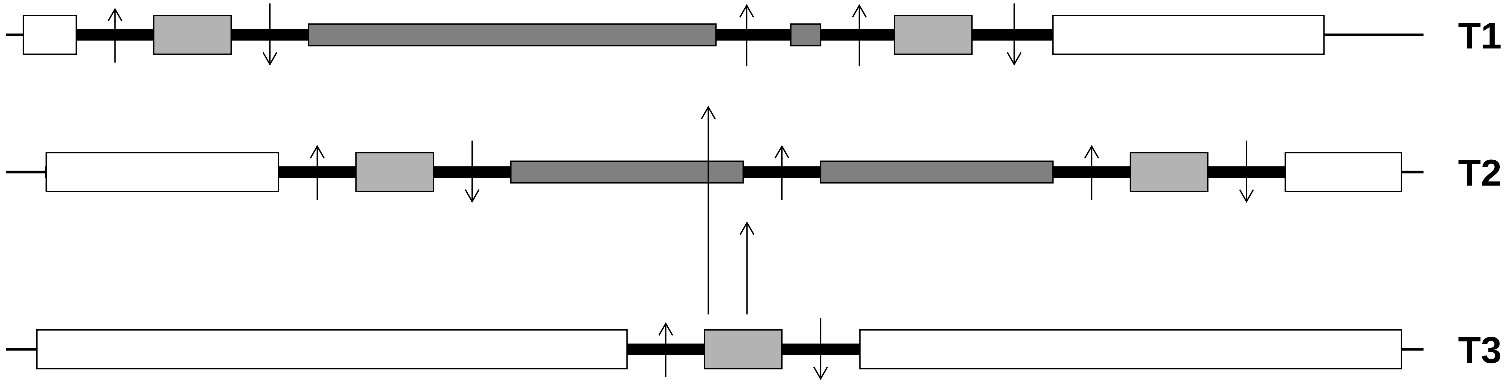
**Desired behavior of broadcast**

```
pthread_mutex_lock(&m);         void synchronized foo()
my_condition = TRUE;            {
pthread_cond_broadcast(&c);       my_condition = true;
pthread_mutex_unlock(&m);         notifyAll();
                                }
```

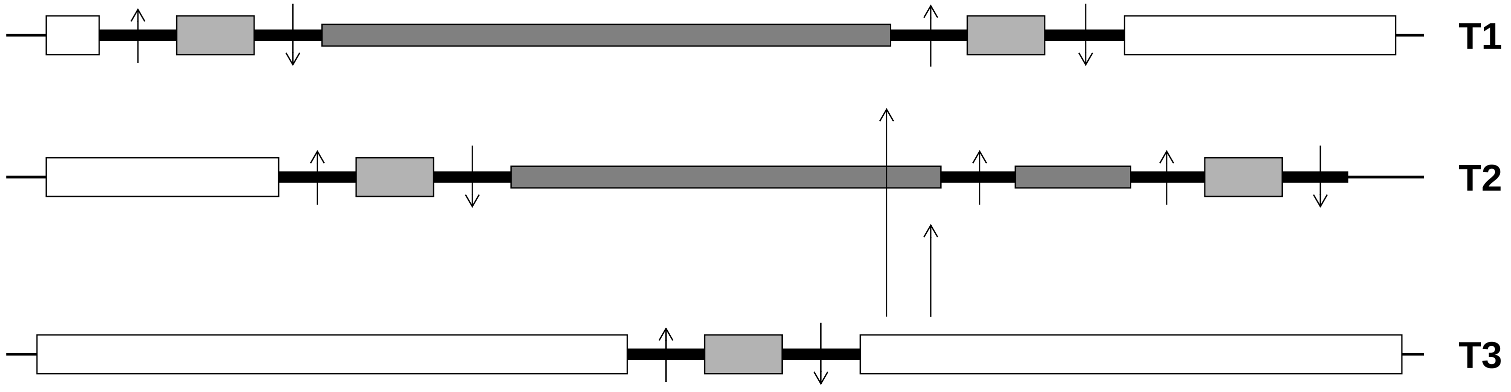
**Broadcast inside of critical section**

**Java notify/notifyAll are defined to be inside of the critical section.**

# CV Execution Graph



Possible behavior of broadcast



Placing the `cond_broadcast()` outside of the critical section

# Broadcast(), notifyAll() is Never *Wrong*

**It’s just not very efficent unless you really need it. (Waking up all those threads takes time!)**

**Many Java programs use `notifyAll()` alot when they really only have work for one thread, because Java isn’t able to distinguish *which* threads are the ones to wake up (e.g., consumers vs. producers).**

**We’ll deal with this soon.**

# Inserting Items Onto a List (Java)

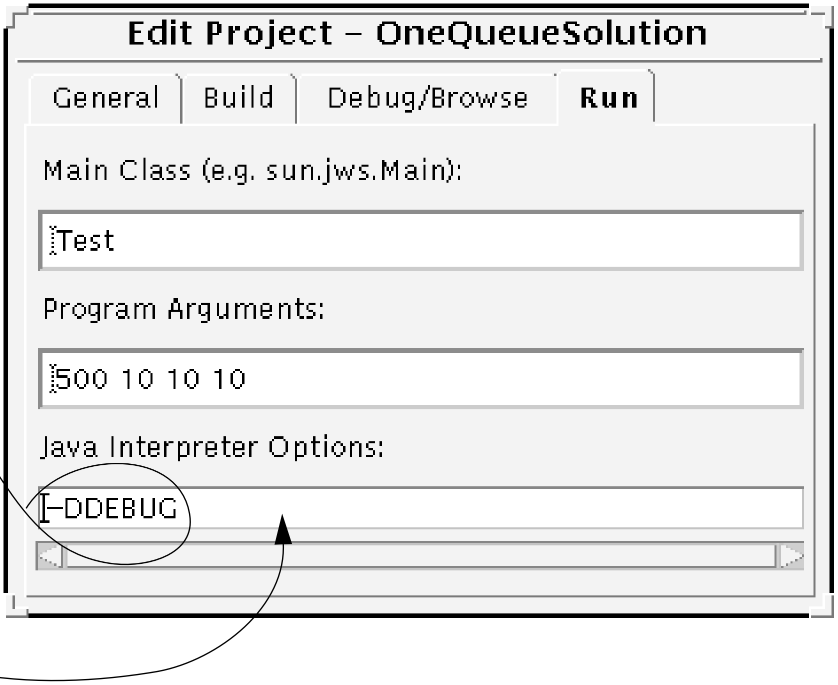
```
public class Consumer implements Runnable {
public void run() {
  try {
    while (true) {
      synchronized (workpile) {
           while (workpile.empty()) workpile.wait();
           request = workpile.remove();
      }
      server.process(request);
    } }
  catch (InterruptedException e) {}// Ignore for now
}


public class Producer implements Runnable {
public void run() {
  while (true) {
    request = server.get();
    synchronized (workpile) {
      workpile.add(request);
      workpile.notify();
    } }
}
```

# Implementing Semaphores in Java

```
public class Semaphore
{int count = 0;

  public Semaphore(int i)
  {count = i;}

  public Semaphore()
  {count = 0;}

  public synchronized void init(int i)
  {count = i;}

  public synchronized void semWait()
  {while (count == 0)
    {try
      wait();
     catch (InterruptedException e) {}
    }
   count--;
  }

  public synchronized void semPost()
  {
   count++;
   notify();
  }

}
```

# Java Tools (Sun's JWS 2.0)

**Project->Edit**



Edit Project – OneQueueSolution

Main Class (e.g. sun.jws.Main): Test

Program Arguments: 500 10 10 10

Java Interpreter Options: -DDEBUG

**Turn on Debug Printing**

**`-DKILL`: Kill Program after 2 minutes.**

# More Wrappers

```
int SEM_WAIT(sem_t *arg)/* Ignore signal
interruptions */
{while (sem_wait(arg) != 0) {}}

void thread_single_barrier(sem_t *barrier, int count)
{
  while (count > 0)
    {SEM_WAIT(barrier);
     count--;
    }
}

#define DEBUG(arg) {if (getenv("DEBUG")) arg;}
```

Thus, to turn on the debug statements:

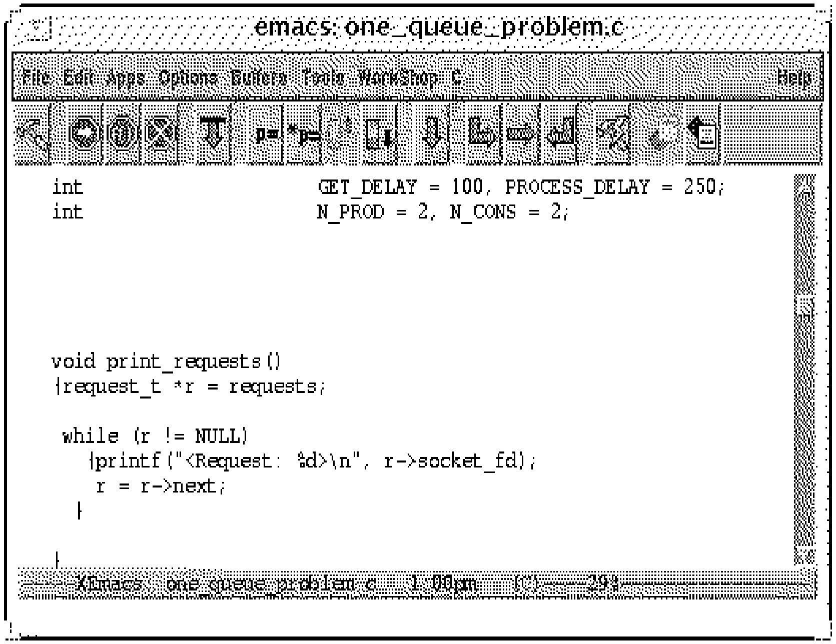
```
% setenv DEBUG
```

And to turn them off:

```
% unsetenv DEBUG
```

Or, from the debugger: *Debug->Edit Arguments... [Environment Variables] -> Name:* DEBUG [Add] [OK]

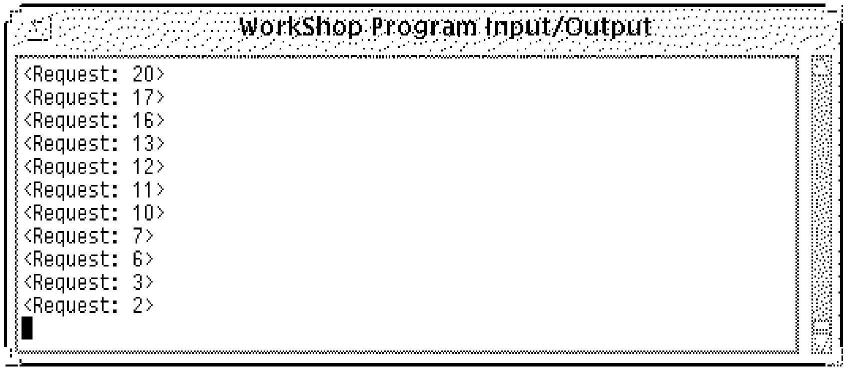
# Workshop Debugger



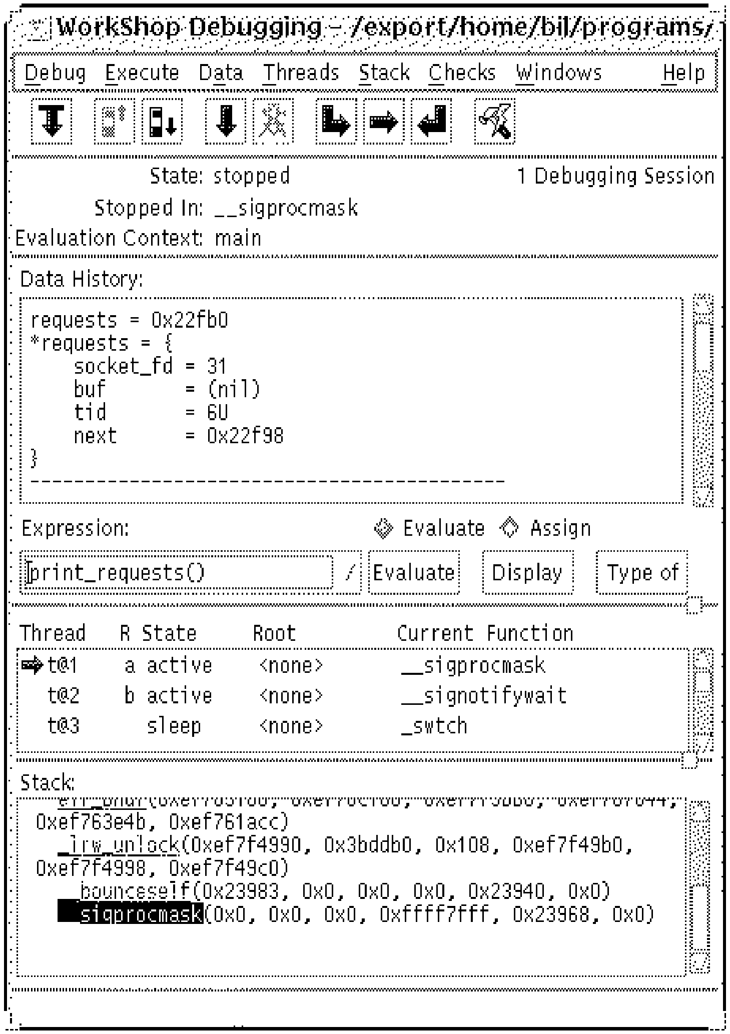
emacs: one_queue_problem.c
File Edit Apps Options Buffers Tools WorkShop C Help

```
int                        GET_DELAY = 100, PROCESS_DELAY = 250;
int                        N_PROD = 2, N_CONS = 2;





void print_requests()
{request_t *r = requests;

 while (r != NULL)
   {printf("<Request: %d>\n", r->socket_fd);
    r = r->next;
   }

}
```

----XEmacs: one_queue_problem.c   1:00pm   (C)----29%----



```
<Request: 20>
<Request: 17>
<Request: 16>
<Request: 13>
<Request: 12>
<Request: 11>
<Request: 10>
<Request: 7>
<Request: 6>
<Request: 3>
<Request: 2>
```



WorkShop Debugging - /export/home/bil/programs/
Debug Execute Data Threads Stack Checks Windows Help

State: stopped

1 Debugging Session

Stopped In: __sigprocmask

Evaluation Context: main

Data History:

```
requests = 0x22fb0
*requests = {
    socket_fd = 31
    buf       = (nil)
    tid       = 6U
    next      = 0x22f98
}
------------------------------------------
```

Expression: ◇ Evaluate ◇ Assign

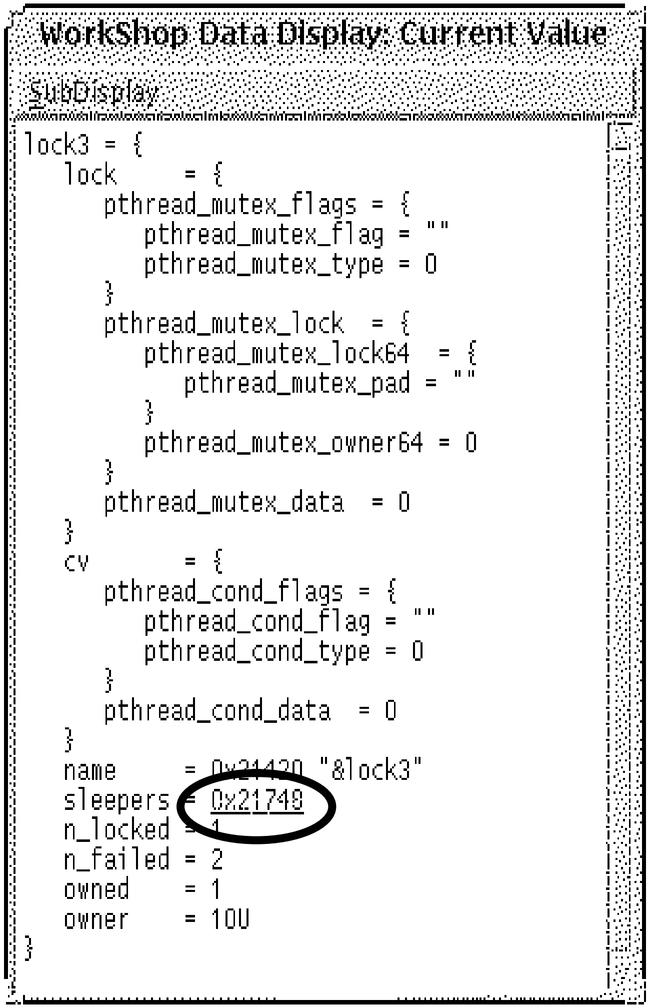
print_requests()

```
Thread  R State     Root        Current Function
➡t@1     a active    <none>      __sigprocmask
 t@2     b active    <none>      __signotifywait
 t@3       sleep     <none>      _swtch
```

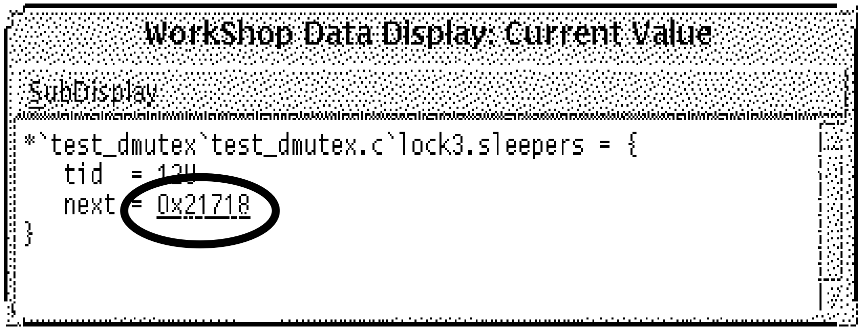
Stack:

```
0xef763e4b, 0xef761acc)
  _lrw_unlock(0xef7f4990, 0x3bddb0, 0x108, 0xef7f49b0,
0xef7f4998, 0xef7f49c0)
    _bounceself(0x23983, 0x0, 0x0, 0x0, 0x23940, 0x0)
    __sigprocmask(0x0, 0x0, 0x0, 0xffff7fff, 0x23968, 0x0)
```

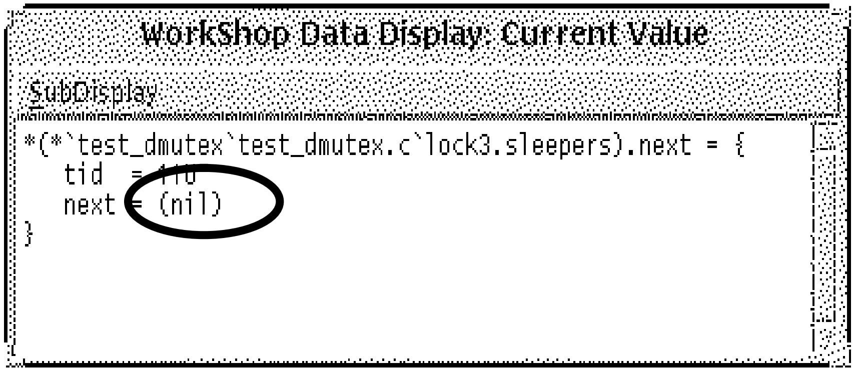
# Data Inspector



```
lock3 = {
   lock     = {
      pthread_mutex_flags = {
         pthread_mutex_flag = ""
         pthread_mutex_type = 0
      }
      pthread_mutex_lock  = {
         pthread_mutex_lock64  = {
            pthread_mutex_pad = ""
         }
         pthread_mutex_owner64 = 0
      }
      pthread_mutex_data  = 0
   }
   cv       = {
      pthread_cond_flags = {
         pthread_cond_flag = ""
         pthread_cond_type = 0
      }
      pthread_cond_data  = 0
   }
   name     = 0x21420 "&lock3"
   sleepers = 0x21748
   n_locked = 1
   n_failed = 2
   owned    = 1
   owner    = 10U
}
```



```
*`test_dmutex`test_dmutex.c`lock3.sleepers = {
   tid  = 12U
   next = 0x21718
}
```



```
*(*`test_dmutex`test_dmutex.c`lock3.sleepers).next = {
   tid  = 11U
   next = (nil)
}
```

# Debugger Details

Evaluate `variable` prints into the Data History window.

Evaluate `function()` runs the function (output to I/O window)

Display `variable` pops up a new Data Inspector window.

Clicking on a pointer (blue) in a Data Inspector window pops up another DI window.

Checks->Enable Memuse Checking turns on the leak detector

Checks->Enable Access Checking turns on out-of-bounds checker.

NB: Debug->Edit Arguments... Arguments: is buggy. Sometimes it works, sometimes it doesn’t (I don’t know why).

# Synchronization Variable Prototypes

```
pthread_mutex_t
error pthread_mutex_init(&mutex, &attribute);
error pthread_mutex_destroy(&mutex);
error pthread_mutex_lock(&mutex);
error pthread_mutex_unlock(&mutex);
error pthread_mutex_trylock(&mutex);

pthread_cond_t
error pthread_cond_init(&cv, &attribute);
error pthread_cond_destroy(&cv);
error pthread_cond_wait(&cv, &mutex);
error pthread_cond_timedwait(&cv, &mutex, &timestruct);
error pthread_cond_signal(&cv);
error pthread_cond_broadcast(&cv);

sem_t
error sem_init(&semaphore, sharedp, initial_value);
error sem_destroy(&semaphore);
error sem_wait(&semaphore);
error sem_post(&semaphore);
error sem_trywait(&semaphore);

thread_rwlock_t
error thread_rwlock_init(&rwlock, &attribute);
error thread_rwlock_destroy(&rwlock);
error thread_rw_rdlock(&rwlock);
error thread_rw_tryrdlock(&rwlock);
error thread_rw_wrlock(&rwlock);
error thread_rw_trywrlock(&rwlock);
error thread_rw_unlock(&rwlock);
```

# Java Synchronization Prototypes

```
synchronized void foo() {...}
synchronized (object) {...}
synchronized void foo() {... wait(); ...}
synchronized void foo() {... notify[All](); ...}
synchronized (object) {... object.wait(); ...}
synchronized (object) {... object.notify[All](); ...}
```

**From Bil’s Extensions package**

```
public void mutex.lock();
public void mutex.unlock();
public void mutex.trylock();

public void cond_wait(mutex);
public void cond_timedwait(mutex, long milliseconds);
public void cond_signal();
public void cond_broadcast();

public void sem_init(initial_value);
public void sem_wait();
public void sem_post();
public void sem_trywait();


public void rw_rdlock();
public void rw_tryrdlock();
public void rw_wrlock();
public void rw_trywrlock();
public void rw_unlock();
```

# *Advanced Synchronization*

# `pthread_cond_timedwait()`
# `object.wait(timeout)`

**You can have a timeout associated with a condition variable wait. *You would only use this in realtime situations.* Be aware that:**

- **If it times out, there still might be an appreciable delay before it is able to reacquire the associated mutex, or before it gets scheduled on a CPU.**
    - **It is possible that the condition will become true during this period! You may choose to reevaluate the condition, you may choose to consider only the timeout.**
- **If it is woken up, there still might be an appreciable delay before it is able to reacquire the associated mutex, or before it gets scheduled on a CPU. In particular, the timeout period might pass during this period. It won't make a difference, as the timeout will have been turned off at wakeup.**

# pthread_cond_timedwait()
# object.wait(timeout)

**In Pthreads, `pthread_cond_timedwait()` returns `ETIMEDOUT` upon timeout, which is good because you know it timed out.**

**In Java, `thread.wait(timeout)` returns nothing, which is bad, because you don't know that it timed out. So you have to do extra work to verify the actual time (remember that a spurious wake up is always a possibility!)**

**(cf: Run the program `TestTimeout`)**

**Avoid timeouts**

# The Lost Wakeup Problem

The lost wakeup problem will cause random failures because a condition signal will be lost every now and then. This is true.

A while back there was quite a bit of confused discussion on this problem. The wrong question was (unfortunately) repeated in much of the SunSoft documentation (2.2 - 2.5).

This particular problem is not possible to write in Java using only wait/notify. However you can do it if you use the additional synchronization objects we provide.

# The Lost Wakeup Problem in POSIX
# ACTUAL

```
     Thread 1 (The Consumer)              Thread 3 (The Signaler)

pthread_mutex_lock(M);
while (!condition)
                                condition = TRUE;
                                pthread_cond_signal(CV);
  pthread_cond_wait(CV, M);

/* Won’t get here! */
do_thing();
pthread_mutex_unlock(M)
```

**If you just blow off using mutexes, lots of things won’t work, including this version of the lost wakeup**

# Semaphore Details

**POSIX defines two types of semaphores, named semaphores which can be accessed by unrelated process, and unnamed semaphores which must reside in the process' address space (possibly shared address space between multiple processes).**

```
sem_open(name, flag);
sem_close(sem);
sem_unlink(name);

sem_getvalue(sem);
```

**There is also a function `sem_getvalue()` which will tell you what the value of the semaphore *was*. It is not very useful because you normally only care about what the value *is*.**

# sem_trywait()

**Just like mutexes, you may simply proceed and do something else. The code is slightly more complex due to the use of `errno` and `EINTR`.**

```
err = sem_trywait(&sem);
if ((err == -1) && (errno == EAGAIN)) ...
```

# Spurious CV Wakeups

**In addition to the EINTR situation, condition variables are allowed to be woken up with *no* provocation what-so-ever!**

**By allowing this, a number of hardware design issues become simpler, and everything can run faster.**

**The consequences of this are that you must *always* recheck the condition.**

```
Good Programmer                   Bad Programmer

pthread_mutex_lock(&m);           pthread_mutex_lock(&m);
while (!condition)                if (!condition)
  pthread_cond_wait(&c, &m);        pthread_cond_wait(&c, &m);

do_thing();                       do_thing();
pthread_mutex_unlock(&m);         pthread_mutex_unlock(&m);
```

**Java may also experience spurious wakeups, so you still must recheck the condition upon waking up.**

**(Yes, I know you can come up with special circumstances where you can know it won’t be changed. *Don’t do that!*)**

# Don’t Wait for Threads to Exit

**`pthread_join()` and relatives are really just different kinds of synchronization variables. They synchronize on the exit and release of the thread resources (stack, structure, TSD, etc.).**

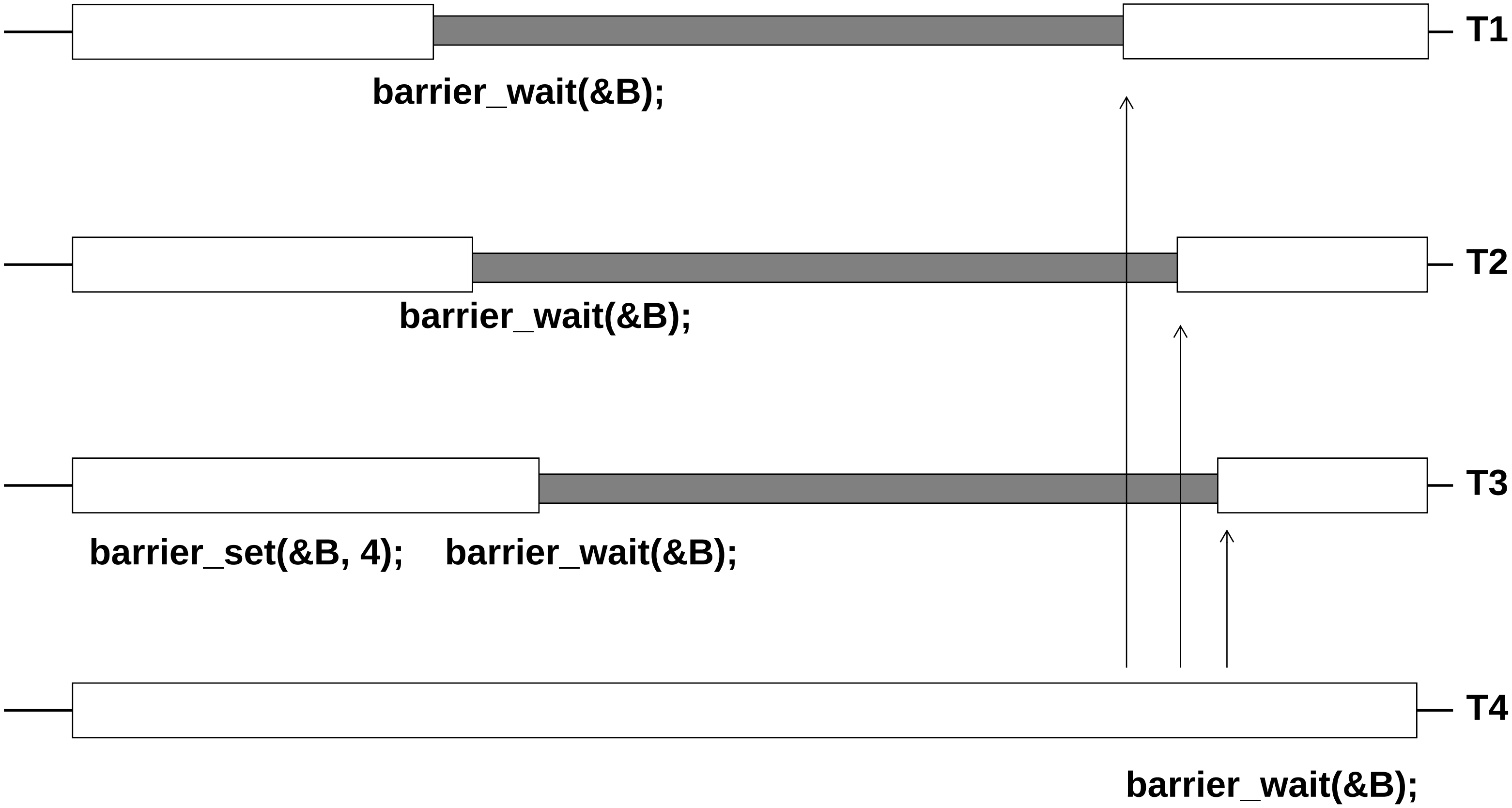
**It is common for people to write programs where they assume that when a thread exits, that the work it was supposed to do is complete. While this is generally true and will produce the correct results, it leads to muddled thinking.**

**You *should* be thinking about the work that was supposed to have been done, and do your synchronizing on that!**

**The only true value of waiting for a thread to exit is when the resource in question *is* the thread. For example, if you have allocated the stack yourself and need to free it, or if you cancel the thread. Cancelled threads return `PTHREAD_CANCELED` (sic).**

**(In UI, you had the option of waiting for *any* thread. Don’t ever do that!)**

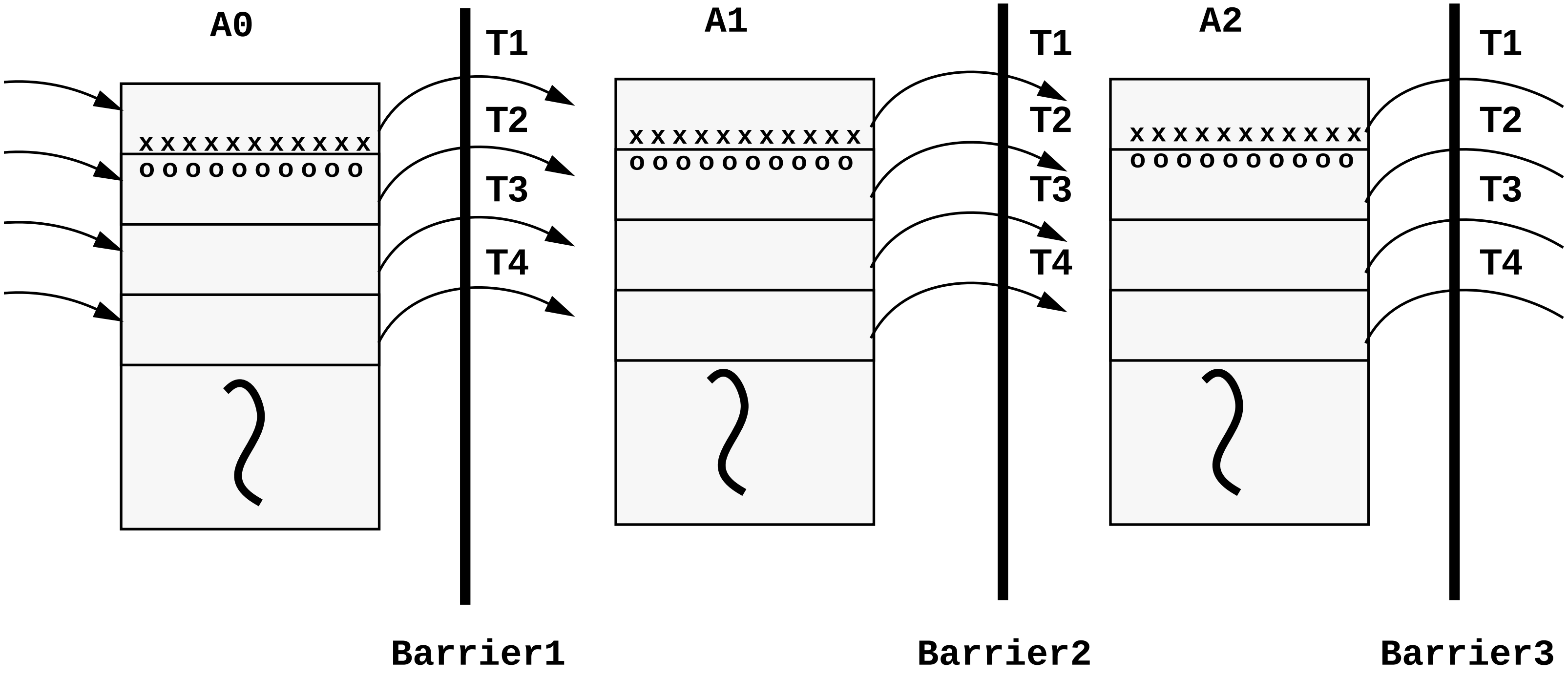
# Barriers



**“Wait until everyone is ready.”**

**(Not part of any library, but easily buildable)**

# Barriers in Simulations (simple)



| Transform 1 | Transform 2 | Transform 3 | Transform 4 |
|---|---|---|---|
| **A0 -> A1**<br>No Wait | **A1 -> A2**<br>`wait(Barrier2)` | **A2 -> A0**<br>`wait(Barrier3)` | **A0 -> A1**<br>`wait(Barrier1)` |

**Each thread does the same subarray and waits at each step.**

# Monitors

- A monitor is an encapsulation of both data and locking.
  - Some languages provide (or require!) monitors (Java).
  - Easy to create monitor class in C++, or to build the effect by hand in C.
  - Some limitations wrt deadlocking or high efficiency.

**Without Monitors:**

```
pthread_mutex_t M;
int count;


counter(int i)
{ pthread_mutex_lock(&M);
  count += i;
  pthread_unmutex_lock(&M);
}

foo()
{ pthread_mutex_lock(&M);
  printf(“Count: %d”, count);
  pthread_unmutex_lock(&M);

}
```

**With Monitors (so-to-speak):**

```
counter(int i)
{ static int count;
  static pthread_mutex_t M;

  pthread_mutex_lock(&M);
  count += i; i = count;
  pthread_unmutex_lock(&M);
  return(i);
}


foo()
{
  printf(“Count: %d”,
counter(0));
}
```

# Monitors in C++

```
class Monitor
{pthread_mutex_t *mutex;
public:
Monitor(pthread_mutex_t *m);
virtual ~Monitor();
};

// Monitor constructor
Monitor::Monitor(pthread_mutex_t *m)
{ mutex = m;
  pthread_mutex_lock(mutex);
}

// Monitor destructor
Monitor::~Monitor()
{ pthread_mutex_unlock(mutex);}

void bar(void)
{Monitor m(&mutex);
 int temp;

 temp = global;
 delay(10, 0);
 printf("T@%d %d == %d\n", pthread_self(), temp, global);
 global = temp+1;

 /* Now the destructor gets called to unlock the mutex */
}
```

# Win32 Mutexes

**Win32 mutexes are kernel objects and require system calls. They are also *recursive*.**

```
lock(&m);
...
  lock(&m);
  ...
  unlock(&m);
...
unlock(&m);
```

**Recursive mutexes are simple to implement in POSIX, but it is unclear if they are actually valuable, or if they simply encourage poor programming!**

**My educated guess is the latter.**

# Win32 “Critical Sections”

These are basically just mutexes by a different name. They are not kernel objects and cannot be cross-process. They are much more efficient than Win32 mutexes (about 100x faster!)

# Win32 Interlocked Functions

```
InterlockedIncrement(&value);
```

```
InterlockedDecrement(&value);
```

```
InterlockedExchange(&value, newValue);
```

These functions use the interlocked instructions on x86 machines (and presumably `LockedLoad` on Alpha) to allow you to manipulate integers safely without a mutex. Unfortunately, you can’t do anything with these values other than to set them. They might change between the time you read them and you make a decision based on their value.

With one exception, these functions provide you with very fast write-only variables.

The one thing you can do is implement reference counting.

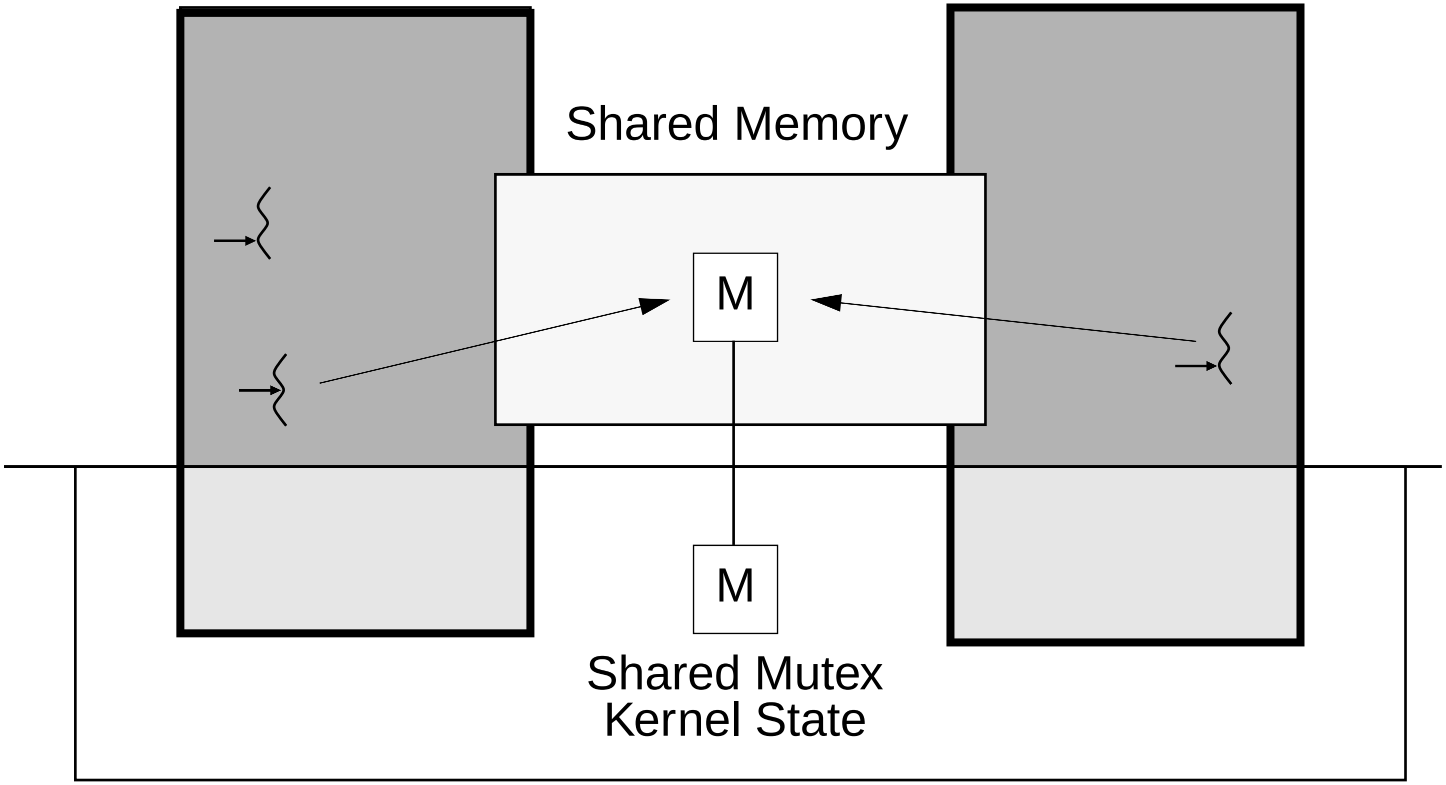
# Avoid These Functions!

# Win32 Multiple Wait Semaphores

- You create a list of mutex semaphores OR a list of event semaphores.
- You then can wait for either:
    - One of the semaphores in list to allow resumption
    - All of the semaphores in the list to all resumption
- Upon waking up, the calling thread will own the semaphore(s) in question

**(I think this is of dubious value. Easily replicated with condition variables.)**

# Cross Process Synchronization Variables



**(Fast cross-process synchronization)**

# Dynamic SV Allocation

**Synchronization variables are just normal data structures and can be malloc'd like any other. They can be stored in a file as part of a binary image, they can be mapped in to different processes, used by different threads, etc.**

**Doing this is sort of tricky and requires well-disciplined programming, of course. E.g., Should you let the process which owns a mutex die, tough luck!**

**When freeing a malloc'd SV, it's up to you to ensure that no other thread will ever try to use it again. ('Course this is nothing new for malloc'd structures.) Obviously, no sleepers allowed.**

**When malloc'ing an SV, you must call the "init" function before using it. Before freeing, you must call the appropriate "destroy" function (see `list_local_lock.c`).**

## Be careful!

# Debug Mutexes

Some vendors supply “debug mutexes” which can be used while debugging your program. They supply more information about their usage (owner, number of times locked, etc.)

Digital UNIX, HP-UX 11.0, Solaris 2.7 have them. (IRIX? AIX? LINUX?)

Here’s the output from Bil’s super-duper debug mutex:

```
bil@cloudbase[117]: test_dmutex

T@12 in critical section 3

&lock3 T@12  Times locked:   1,  failed:   2.
            Sleepers: ( T@10 T@11 )
&lock2 ---- Times locked:   0,  failed:   0.
            Sleepers: ( )
&lock1 T@9  Times locked:   1,  failed:   5.
            Sleepers: ( T@4 T@5 T@6 T@7 T@8 )
```

# Word Tearing

**Synchronization of data assumes that data to be uniquely addressable -- the data being changed will not affect any other data.**

**This is not always the case. To wit: many modern machines cannot address (load and store) anything smaller than a word (perhaps 32-bits, perhaps 64). If you tried to use two different parts of the same word as two different variables, you'd be in trouble. *EVEN WITH (distinct) MUTEXES.* Don't do that.**

```
char array[8];                     /* Bad! */
Thread 1: array[0]++;
Thread 2: array[1]++;
```

**Normally this will not be a problem, because the compiler aligns different variables on word boundaries. Hence:**

```
char a1, a2;                       /* OK */
```

**will not be a problem.**

# Volatile

**This keyword in C is used to indicate to the compiler that the variable in question changes independently of the local code. Hence, the compiler is not allowed to optimize away loads or stores. Indeed, loads must come from main memory, not be filled from cache. Stores should be expedited around the store buffer.**

**The idea here is that memory-mapped I/O uses memory addresses as I/O registers and every read or write is meaningful.**

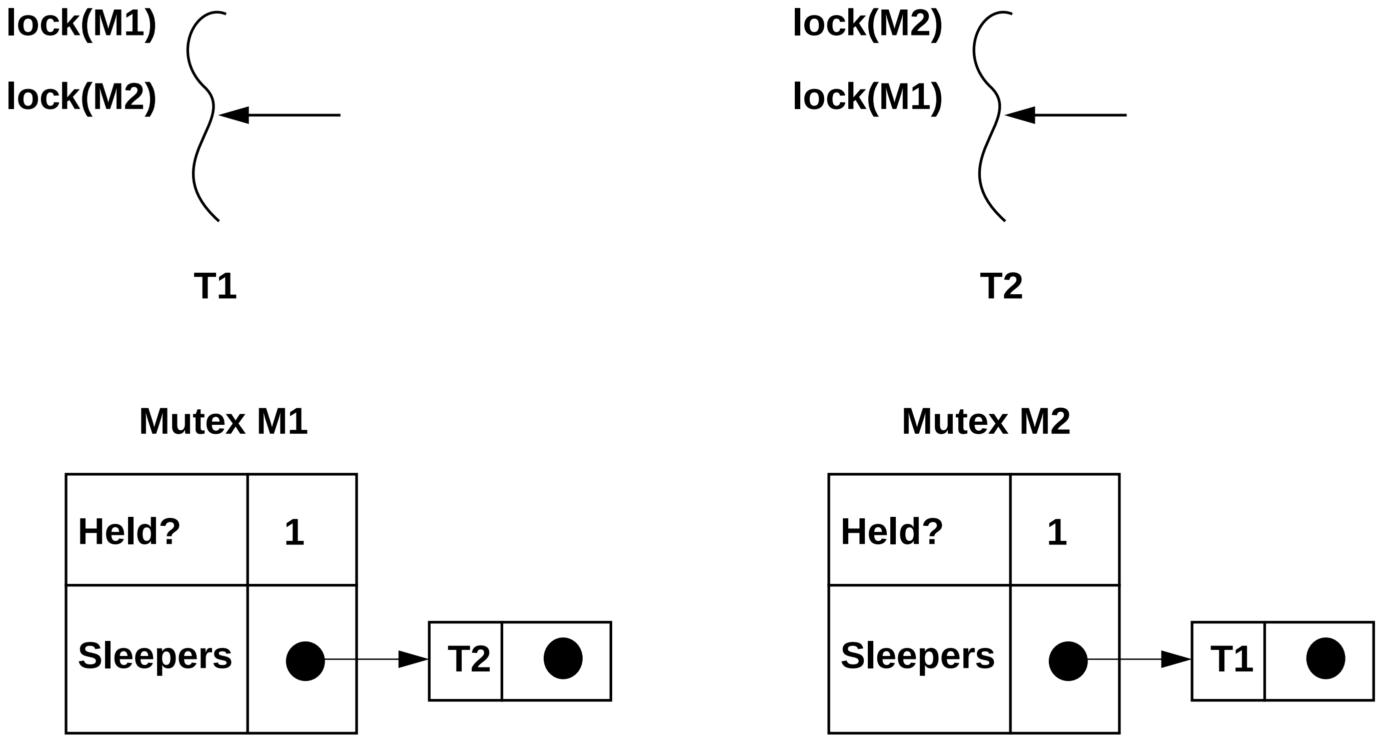
**This is completely orthogonal to threads. Do not use `volatile` with threads, thinking that it will solve any threading problem. You won't like the results.**

**The Java spec says `volatile` can be used instead of locking. It's right, but misleading. Use locking.**

***Don't do that!***

# *Synchronization Problems*

# Deadlocks



lock(M1)
lock(M2)
T1

lock(M2)
lock(M1)
T2

Mutex M1

| | |
|---|---|
| Held? | 1 |
| Sleepers | ● |

T2 ●

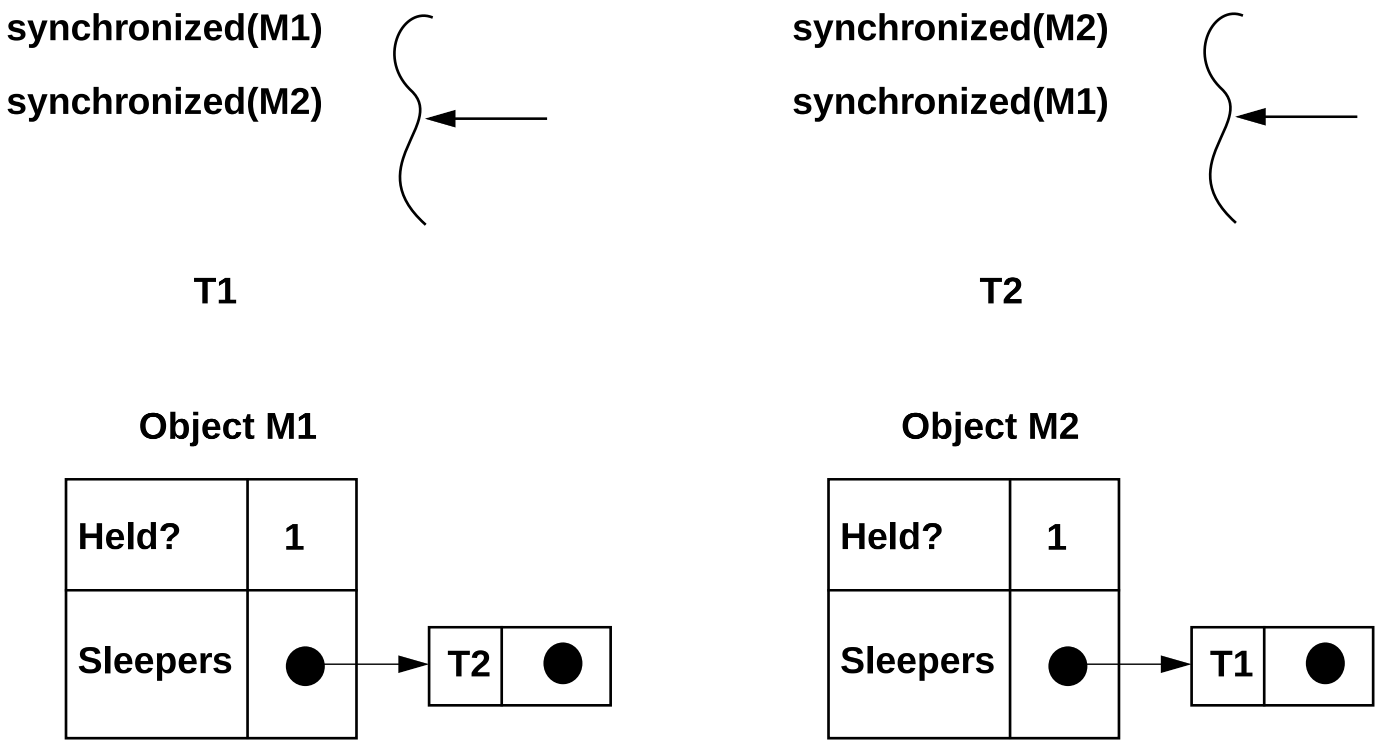
Mutex M2

| | |
|---|---|
| Held? | 1 |
| Sleepers | ● |

T1 ●

**Unlock order of mutexes cannot cause a deadlock**

# Java Deadlocks



synchronized(M1)

synchronized(M2)

T1

synchronized(M2)

synchronized(M1)

T2

Object M1

| Held? | 1 |
|---|---|
| Sleepers | ● |

T2 ●

Object M2

| Held? | 1 |
|---|---|
| Sleepers | ● |

T1 ●

# Avoiding Deadlocks

- Establish a Hierarchy: Always lock Mutex_1 before Mutex_2...
- Use the trylock primitives if you want to violate the hierarchy for extreme efficiency concerns.

```
pthread_mutex_lock(&m2);
...
     if (EBUSY == pthread_mutex_trylock(&m1))
         {pthread_mutex_unlock(&m2);
          pthread_mutex_lock(&m1);
          pthread_mutex_lock(&m2);
         }
do_real_work();               /* Got ‘em both! */
}
```

- Use a static analysis program to scan for hierarchy violations.
- Java has no “trylock”, but extreme efficiency is not a Java issue anyway.
- Unlock order of mutexes cannot cause a deadlock in a proper hierarchy.

# Recursive Locking

First of all, “recursive” isn’t really an accurate description, it’s really iterative. But everybody calls it recursive, so we will too.

What if one function locks a mutex and then calls another function which also locks that mutex? In POSIX, using normal mutexes, this will deadlock. In Win32 and Java synchronized sections, it will not.

```
void foo()                    public synchronized void foo()
{pthread_mutex_lock(&m);      {
 bar();                           bar();
 pthread_mutex_unlock(&m);        ...
}                             }

void bar()                    public synchronized void bar()
{pthread_mutex_lock(&m);      {
 ...                              wait();    // works fine!
 pthread_mutex_unlock(&m);    }
}
```

Building recursive mutexes in POSIX is a simple exercise. The first question is “Do you need to?” The answer is “No.” The second question is “Do you want to?” Probably not.

The reason for wanting recursive mutexes is because the code is badly structured. The right thing to do is to restructure the code.

# Race Conditions

A race condition is where the results of a program are different depending upon the timing of the events within the program.

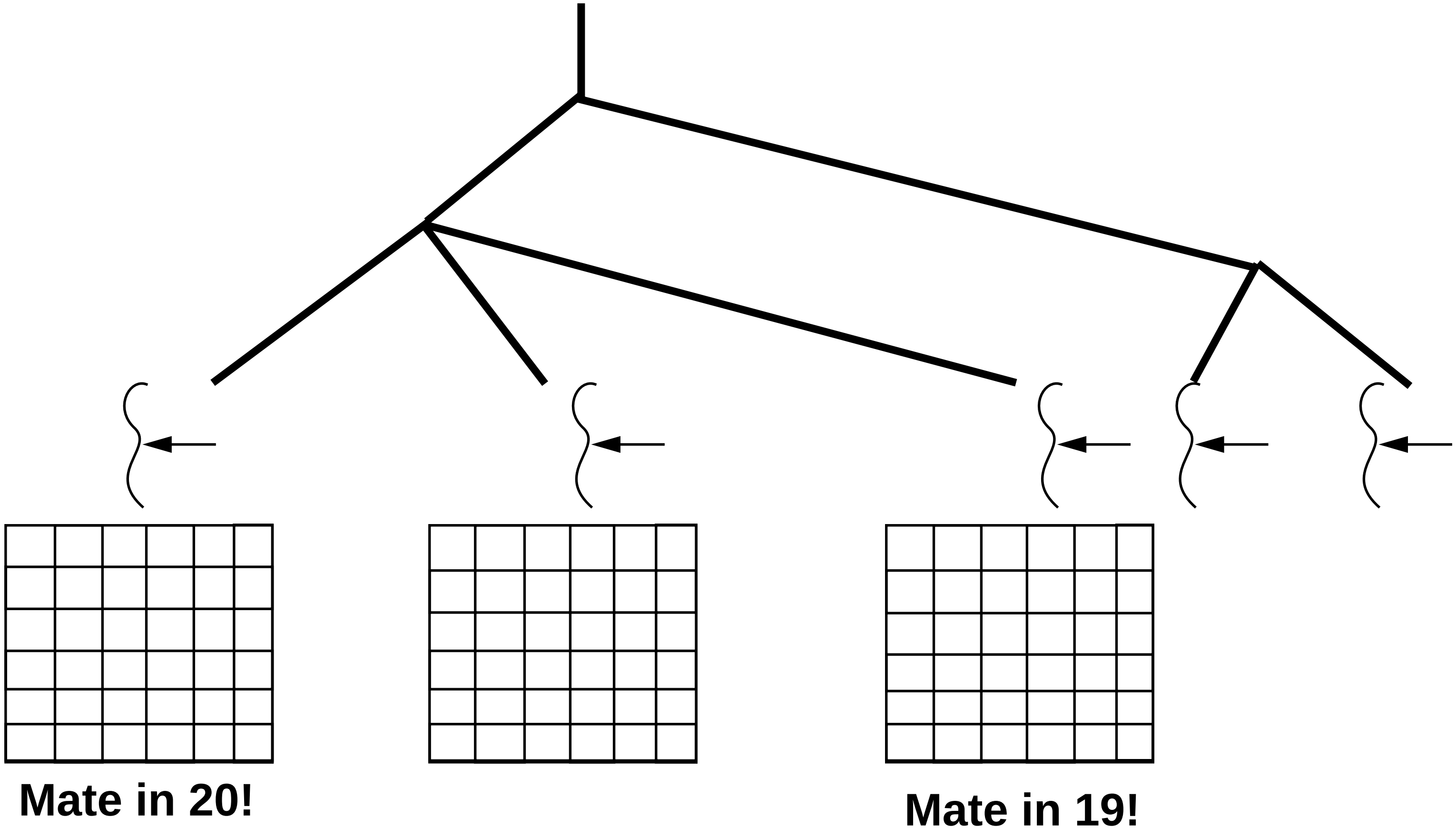
Some race conditions result in different answers and are clearly bugs.

```
Thread 1                            Thread 2

mutex_lock(&m)                      mutex_lock(&m)
v = v - 1;                          v = v * 2;
mutex_unlock(&m)                    mutex_unlock(&m)
```

Most of the time, when people refer to race conditions, they mean failure to hold a mutex while accessing shared data. That kind of problem is easily fixed. This kind is harder.

# Race Conditions

Some race conditions result in answers with different details, but where either detail is correct.

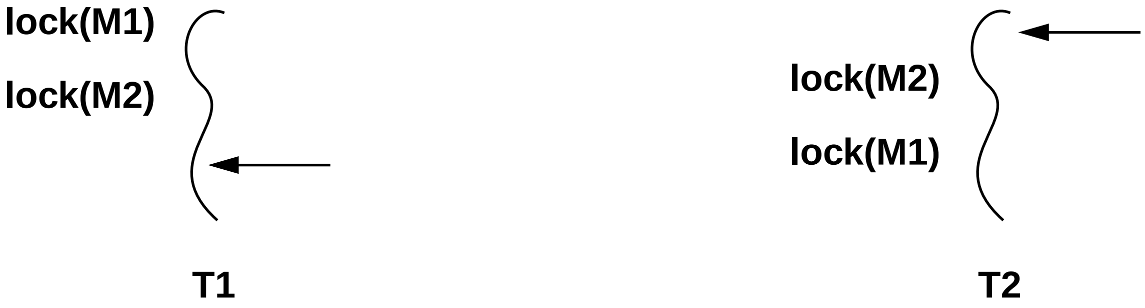


Mate in 20!

Mate in 19!

# Race Conditions

The most common race conditions result in deadlocks!



lock(M1)

lock(M2)

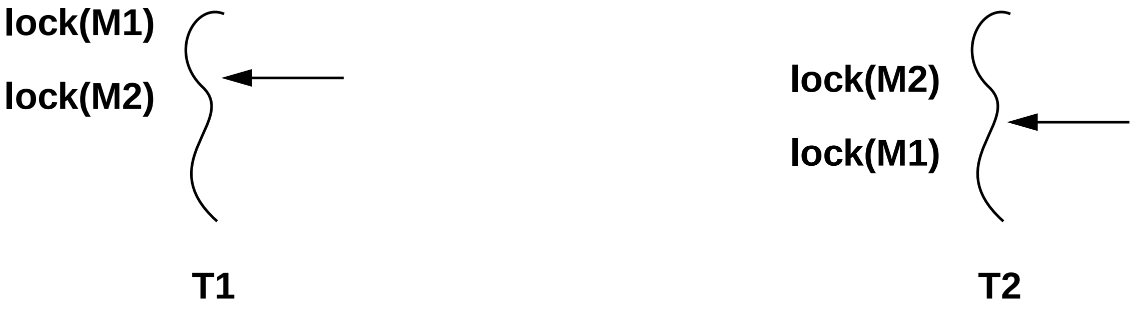
T1

lock(M2)

lock(M1)

T2

The code will work the first 500,000,000 times, but then...



lock(M1)

lock(M2)

T1

lock(M2)

lock(M1)

T2

# Recovering From Deadlocks

1. Don't! Fix the program!!
2. Be very, very, very careful!
3. Never try to recover in a single-process MT program
   - If the program is icontool, just kill it
   - If the program is a 747 autopilot...
   - You must not "just release the mutex"
4. For multi-process programs:
   a. Look at system V semaphores or UI *robust mutexes*
   b. Design the program for two-phase commit

**Lots of people like to talk about recovering from deadlocks, but it is a VERY difficult thing to do well, and programming for it can be VERY time consuming, with a slow-running result.**

# Robust Mutexes

**They are unique to “Solaris Threads” in Solaris 2.7 If a shared memory mutex is owned by a process which crashes, the next thread which tries to lock it will receive an error return. Now all the programmer has to do is to figure out what shared data might have been partially changed, and fix it.**

```
mutex_init(&m, USYNC_PROCESS_ROBUST, 0);

e = mutex_lock(&m);
switch (e)
  case 0:             /* Normal return value. */
     do_normal_thing();

  case EOWNERDEAD:    /* Lock acquired. Owner died.*/
     recover_data();
     mutex_init(&m, USYNC_PROCESS_ROBUST, 0);

  case ENOTRECOVERABLE: /* Lock not acquired. */
     exit(1);
```

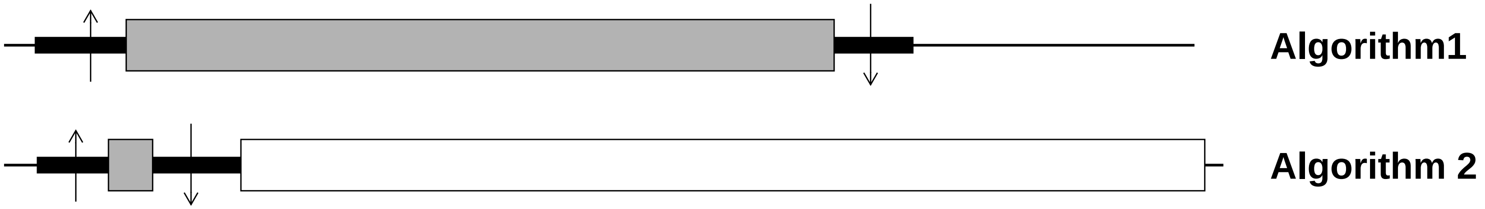
# *Locking Performance*

# Performance Problems

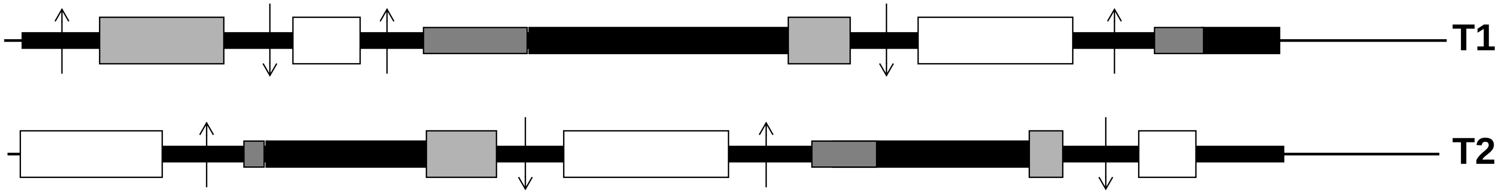
1. **Verify it's important**
    - **(VAX Idle loop)**
2. **Produce a repeatable test case**
3. **Consider other possible algorithms**
4. **Analyze the problem area in detail**
5. ***Consult with a peer <------------ ! ! ! ! ! ! ! ! ! ! !***
6. **Consider specialized optimizations**
    - **Loop unrolling**
    - **Data ordering**
    - **Data redefinition**
    - **Assembly coding**
    - **Complex locking schemes**

# High Contention Locks

- Can you write a less contentious algorithm?



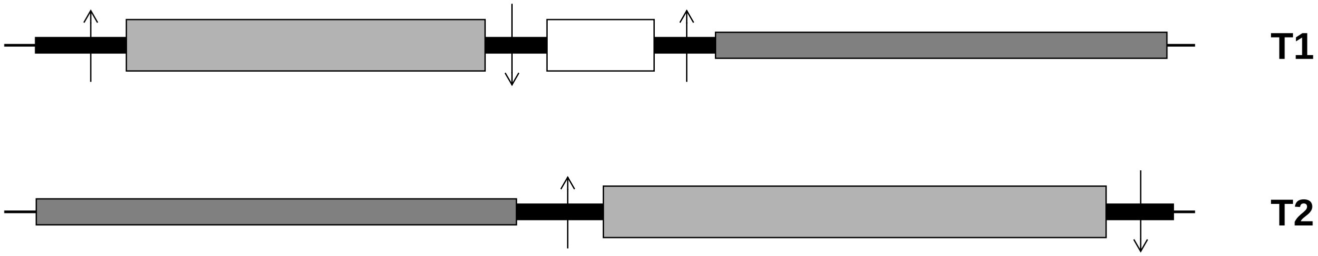
- Is context switching dominating your run time? Spin Locks



- Is a deserving thread being starved for mutex ownership?
  - Is it because another thread is too quick? FIFO Locks
  - Is it a matter of scheduling priorities? Priority Inheritance

# High Contention Locks

- If your best algorithm looks like this:



**You Lose!**

**(better stated: “You’ve gone as far as you can go. You’re a success!”)**

# There is Always a Limit

Every program has an optimal number of CPUs which it can run on. Exceeding that number will yield inferior results. The more you exceed it, the worse it will get. This is usually related to memory bus throughput.

The degradation should be gradual. 1 extra CPU is a little bit worse than the correct number, etc.

Large numerical programs can run productively with 64 CPUs on a Cray 6400.

NFS got its best results with 8 CPUs on a SC2000 (1993). Then a 12-way SC2000 (1995). Now a 24-way UE6000.

The best DBMS results came from a 16-way SC2000. Now a 16-way UE6000.

# Synchronization Performance

| | POSIX | Java 1.1 | Java 1.2 | Java 1.2 |
|---|---|---|---|---|
| Function (μs) | | (Green) | (Green) | (Native) |
| Synchronized Method | | 4 | 4 | 4 |
| Mutex Lock/Unlock | 1.8 | 16 | 30 | 20 |
| Mutex Trylock | 1.3 | | | |
| Reader Lock/Unlock | 4.5 | 40 | 80 | 60 |
| Writer Lock/Unlock | 4.5 | | | |
| Semaphore Post/Wait | 4.3 | 14 | 20 | 12 |
| Notify (CV Signal) Zero Threads | 0.2 | 6 | 6 | 6 |
| Local Context Switch (unbound threads) | 90 | 340 | 75 | 130 |
| Local Context Switch (bound threads) | 40 | | | |
| Process Context Switch | 54 | | | |
| Change Signal Mask | 18 | | | |
| Cancellation Disable/Enable | 0.6 | 16 | 45 | 50 |
| Test for Deferred Cancellation | 0.25 | 3 | 20 | 25 |
| Create an Unbound Thread | 330 | 1200 | 1600 | 2600 |
| Create a Bound Thread | 720 | | | |
| Create a Process | 45000 | | | |
| Reference a Global Variable | 0.02 | 0.08 | 0.08 | 0.08 |
| Reference Thread-Specific Data | 0.6 | 2 | 4 | 4 |

**110MHz SPARCstation 4 (This machine has a SPECint of about 20% of the fastest workstations today.) running Solaris 2.6.**

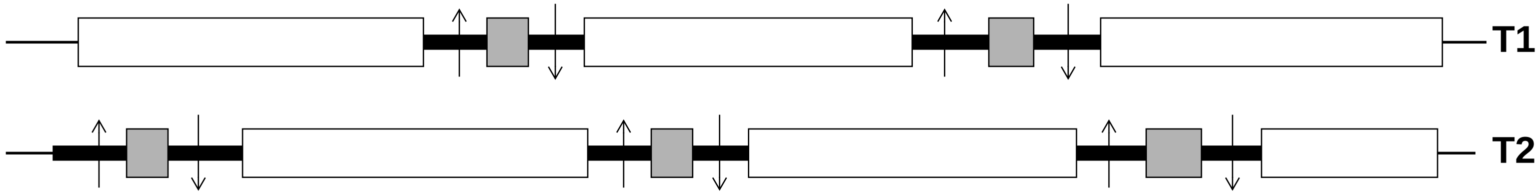
# Spin Locks

```
/* Example code. Do NOT use this code! */

spin_lock(pthread_mutex_t *m)
{int i;
  for (i=0; i < SPIN_COUNT; i++)
    {if (pthread_mutex_trylock(m) != EBUSY)
      return; }              /* got the lock! */
  pthread_mutex_lock(m);     /* give up and block. */
  return; }                  /* got the lock after blocking! */
```

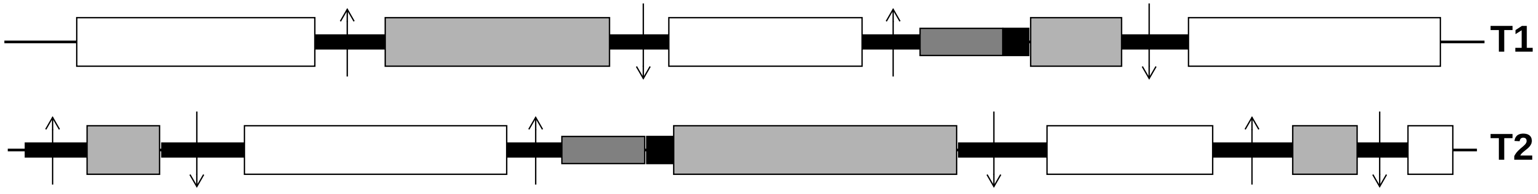
**(We’ll look at spin locks again and see the actual design.)**

**This is not an issue for Java because extreme efficiency is not an issue there...**

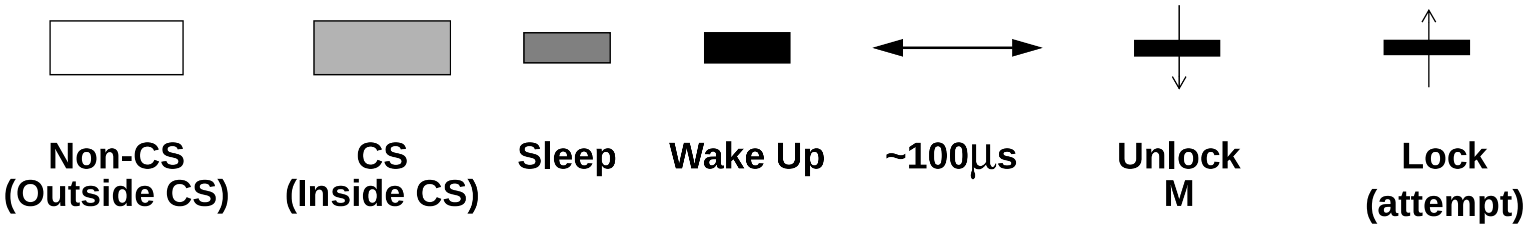
# Spin Lock Execution Graph



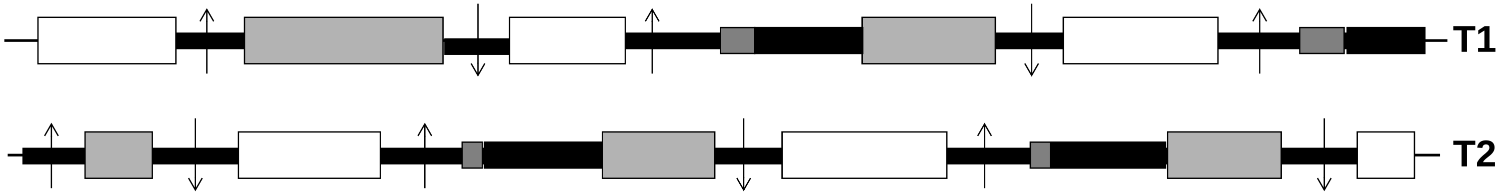
1: The common case: Non-CS >> M, Non-CS>> CS. Very little contention.
Spin locks won’t help.



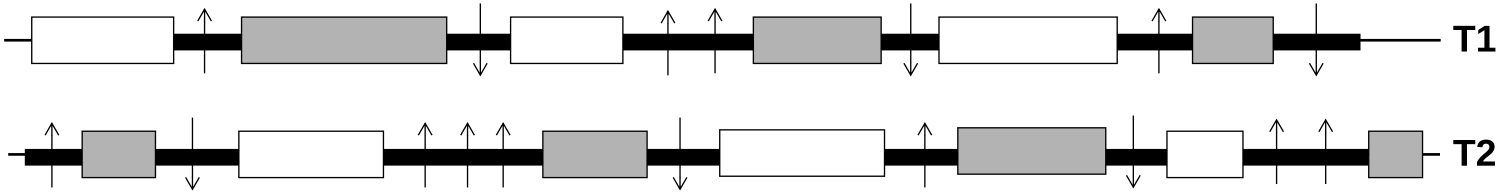
2: The uncommon case: Non-CS >> M, Non-CS ~ CS. Significant contention.
Spin locks won’t help.



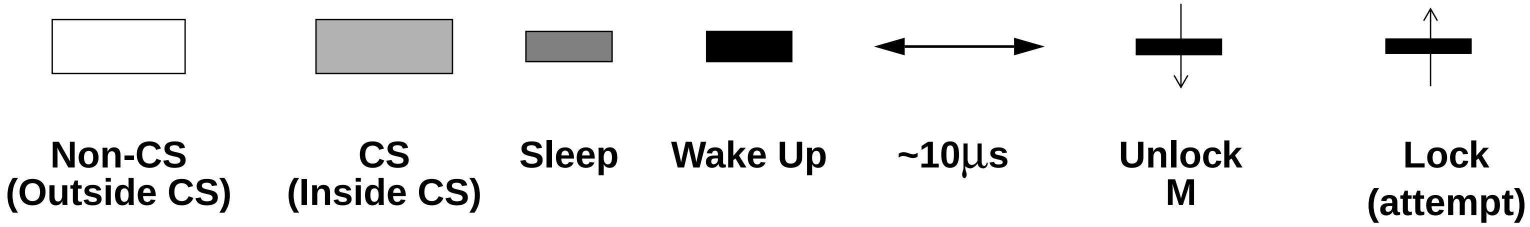
# Spin Lock Execution Graph



**3: The very uncommon case: Non-CS ~ M, Non-CS ~ CS. Significant contention. Spin locks will help.**



**4: The very uncommon case: Non-CS ~ M, Non-CS ~ CS. Significant contention. Spin locks do help.**



# Spin Locks are Only Useful When:

- There is high contention
- The critical section is short (CS << 100μs)
- The owner is currently running on an MP machine

When deciding to use spin locks, you must do the statistics yourself. You wrote the program, you know the length of the critical sections, you know the machine you’re running on.

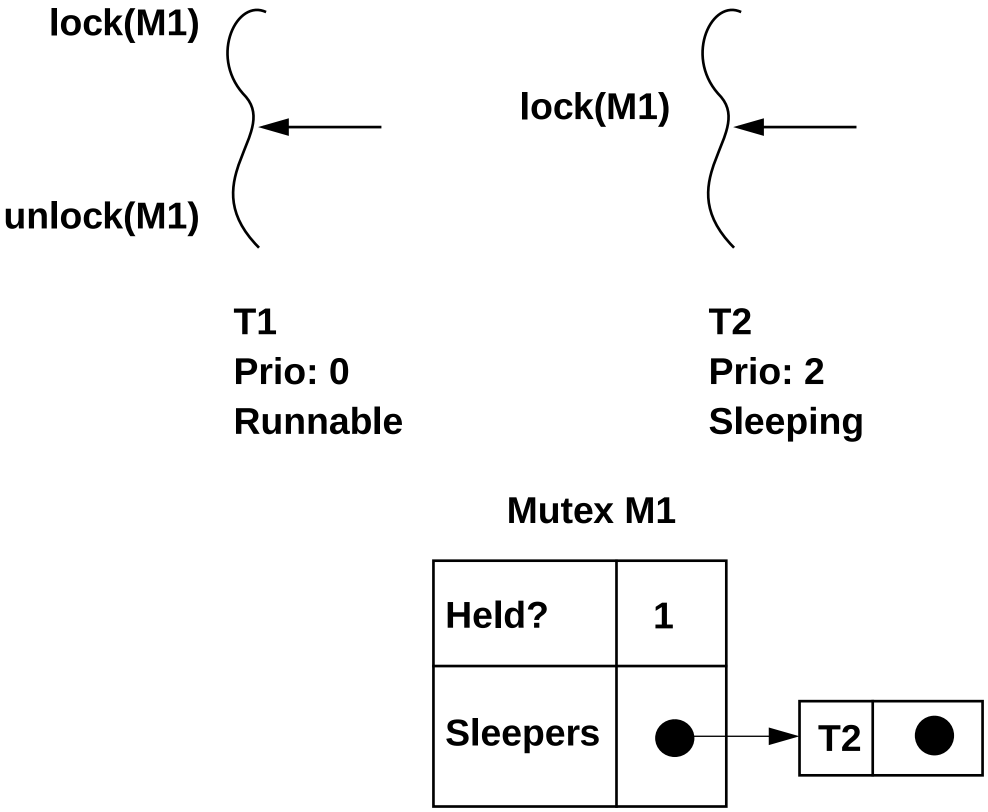
Spin Locks are not part of the POSIX API, but are included in some Pthreads extensions. Ask your vendor.

# Adaptive Spin Locks

- **Spin Locks are only useful if the holder of the lock is currently running on a CPU.**
- **An *Adaptive Spin Lock* looks to see if the owner is running before it begins to spin.**
    - **This requires the owner to be known**
    - **and the kernel run state to be known.**

**Adaptive spin locks are not generally available to the user-level program. There are some tricks the OS writers can use to get around this. Ask your vendor. (All mutexes in Solaris 2.6 are adaptive spin locks.)**

# Priority Inversions



lock(M1)

unlock(M1)

T1
Prio: 0
Runnable

lock(M1)

T2
Prio: 2
Sleeping

T3
Prio: 1
Active

Mutex M1

|  |  |
|---|---|
| Held? | 1 |
| Sleepers | ● |

| T2 | ● |
|---|---|

**Java Synchronized sections are also subject to priority inversion.**

# Priority Inheritance Mutexes

**are part of the optional portion of the POSIX spec. They are defined for realtime threads only. Use by a non-realtime thread is undefined. POSIX defines two types:**

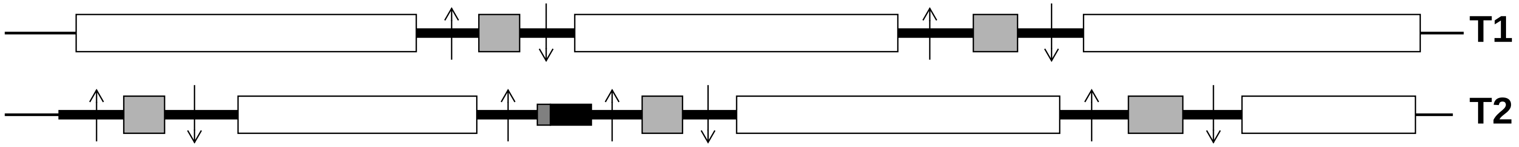
**Priority inheritance mutexes have the owner thread inherit the priority of the highest sleeper thread: `PTHREAD_PRIO_INHERIT`.**

**Priority ceiling mutexes have the owner thread take a constant priority value from the mutex itself: `PTHREAD_PRIO_PROTECT`.**

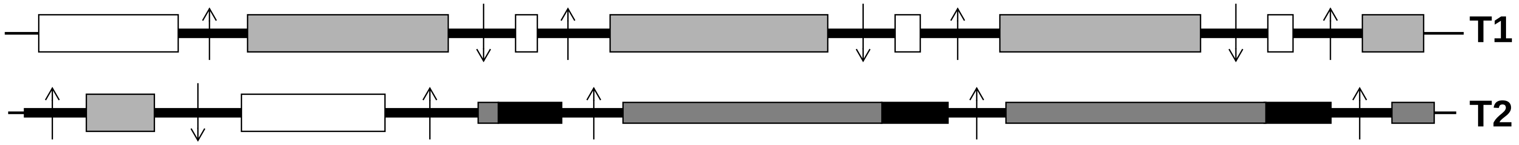
**You can also build PI mutexes yourself (in POSIX or Java).**

**Unless you are a heavy realtime dude, don’t expect ever to use these. (I am not!)**

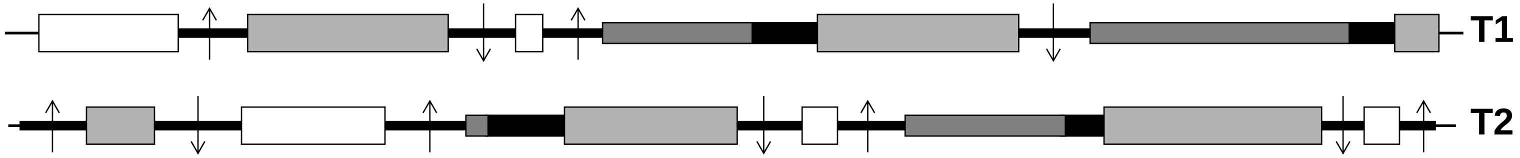
# FIFO Mutexes



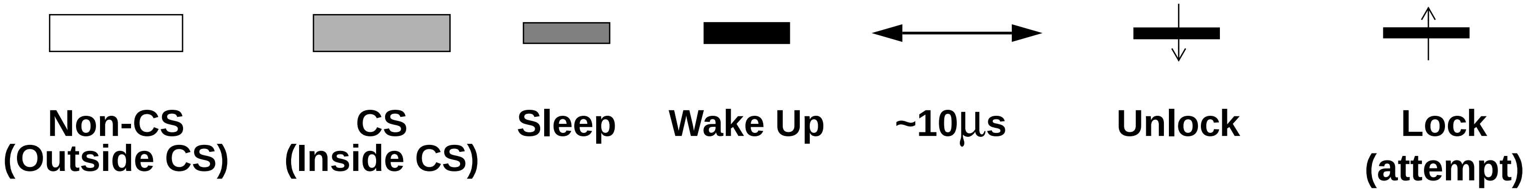
**1: The common case: Very little contention, normal mutexes work well.**



**2: The uncommon case: T1 keeps reacquiring the mutex.**



**3: The uncommon case: Using a FIFO mutex.
Does not improve performance, only ”fairness”.**



# FIFO Mutexes

FIFO mutexes are also easy to build. The one caveat is that the code cannot be 100% correct without extensive hardware support (which does not exist).

Happily, 99.999% correct is quite sufficient for most situations.

See extensions.c, Extensions.java

# Complex Locking

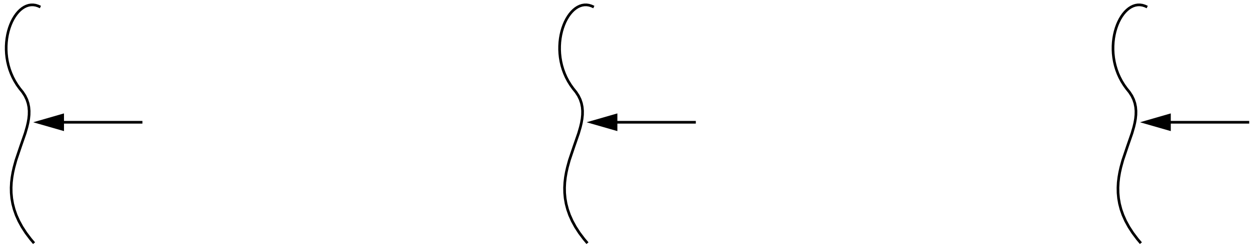
- Always at least 2x slower than simple mutexes.
- Spin Locks are only useful if 1 - 4 CPUs are spinning at one time.
    - More than just 1 spinner would be *highly* unusual!
    - If you have lots of spinners, create fewer threads.
- Complex locking primitives are very seldomly useful.

**Be Careful!**

# *Thread Specific Data*

# Thread Specific Data

**Sometimes it is useful to have a global variable which is local to a thread (`errno` for example).**



```
err = read(...)
if (err) printf(“%d”, errno);
```

```
err = write(...)
if (err) printf(“%d”, errno);
```

```
err = ioctl(...)
if (err) printf(“%d”, errno);
```

# TSD

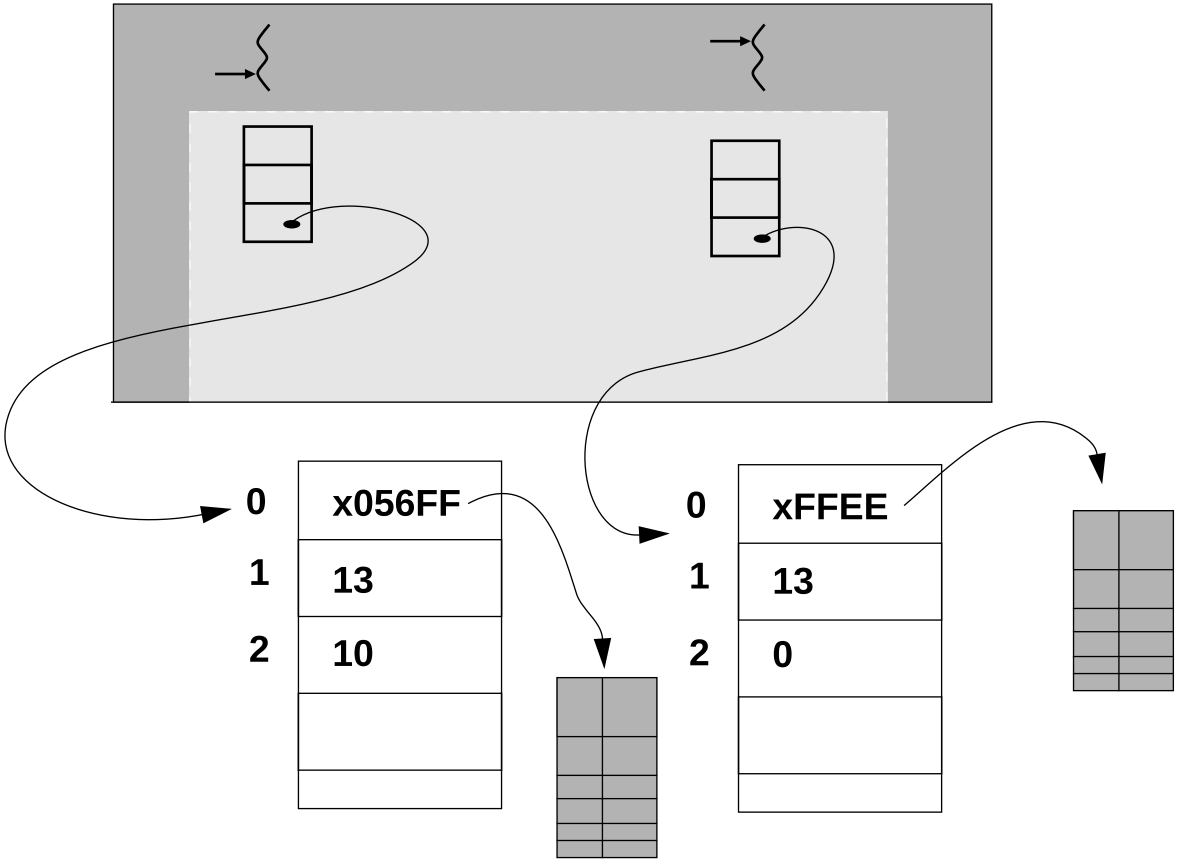
## (Solaris Implementation)

**TSD Keys**

| | |
|---|---|
| car_key | 0 |
| cdr_key | 1 |
| house_key | 2 |

**Destructors**

| | |
|---|---|
| 0 | `fun0()` |
| 1 | `NULL` |
| 2 | `fun2()` |
| | |
| | |



| | |
|---|---|
| 0 | x056FF |
| 1 | 13 |
| 2 | 10 |
| | |
| | |

| | |
|---|---|
| 0 | xFFEE |
| 1 | 13 |
| 2 | 0 |
| | |
| | |

**Officially, TSD keys are opaque. Often they are integers.**

# TSD Usage

```
extern pthread_key_t house_key;

foo(float x)
{
  pthread_setspecific(house_key, (void *) x);
  bar();
}

bar()
{float n;
  n = (float) pthread_getspecific(house_key);
}

main()
{...
  pthread_key_create(&house_key, destroyer);
  pthread_create(... foo, 3.14159265358979);
  pthread_create(... foo, 1.41421356);
...}
```

# Win32 TSD

- **Win32’s TSD is quite similar to POSIX:**

```
key = TlsAlloc();
TlsSetValue(key, data);
data = TlsGetValue(key);
```

# Thread Local Storage

This is a different technique of obtaining the same results.

Instead of making a function call, if you could just declare a variable to be thread-local, you could get the same effect as TSD, only it would be faster. This is what TLS refers[1] to.

- Only Win32 provides TLS.
- TLS requires compiler changes — an unacceptable requirement for POSIX.

[1] The term TLS is not used consistently. Win32 uses “Dynamic TLS” to mean what POSIX calls TSD, and “Static TLS” to mean what POSIX calls TLS.

# Deleting TSD Keys

**It is possible to delete TSD keys and to recycle the array locations for use by other keys:**

```
extern pthread_key_t house_key;
extern pthread_key_t car_key;

pthread_key_create(&house_key, destroyer);
... use the key ...
pthread_key_delete(&house_key);

pthread_key_create(&car_key, destroyer);
... use the key ... (the old location might be reused by the new key)
```

**If you do delete a key, the destructors will not run (you’re responsible for that), and any future use of the old key is an error (which might not be caught by the system).**

**Don’t do That!**

# TSD Destructors

**When a thread exits, it first sets the value of each TSD element to `NULL`, then calls the destructor on what the value was. No order is defined for the various keys.**

**A typical destructor will do something simple, such as freeing memory:**

```
void destructor(void *arg)
{free(arg);}
```

**It is legal (if silly) to do more complex things, such as:**

```
void destructor(void *arg)
 {free(arg);
 pthread_setspecific(key2, malloc(1000));}
```

**The destructor for `key2` will be run a second time if necessary. And third, forth... up to at least `_PTHREAD_DESTRUCTOR_ITERATIONS` times. (On Solaris, this will run forever.)**

***Don’t do that.***

# pthread_once()

To allow dynamic initialization of data, you need to lock a mutex, initialize the data, set a flag (indicating that the data is initialized), and unlock the mutex. Then, on every use of that data, you must lock the mutex and check the flag.

You could write it yourself:

```
                                               int date;

foo()                                          foo_init()
{  static pthread_mutex_t m = ...INITIALIZER;  {
   static int date;                               set_date(&date);
   static int initialized = FALSE;             }

   pthread_mutex_lock(&m);
   if (!initialized)                           foo()
     {set_date(&date);                         {static pthread_once_t once = \
      initialized = TRUE;                           PTHREAD_ONCE_INIT;
     }
   pthread_mutex_unlock(&m);                   pthread_once(&once, foo_init)
   ... use date...                             ... use date...
   return;                                        return;
}                                              }
```

Load-time or start-time initialization is easier!

# Double Checked Locking

One issue with dynamic initialization is that every time you use the function (or object) in question, you must lock a mutex in order to see if it’s been initialized. This extra microsecond can be significant (although we tend to exaggerate). The “Double Checked Locking” pattern allows us to optimize away most of this time.

```
value_t *get_value() {
  static int initialized = FALSE;
  static value_t *value = NULL;
  static pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

  if (initialized) return value;
  pthread_mutex_lock(&m);
  if (value == NULL) value = initialize_value();
  pthread_mutex_unlock(&m);
  initialized = TRUE;
  return value;
}
```

# Double Checked Locking

**Here is the same code in Java. Typically this method will be in a singleton object, although it doesn’t have to be.**

```
public Value get_value() {

  if (initialized) return value;
  synchronized(this) {
     if (value == NULL) value = initialize_value();
  }
  initialized = TRUE;
  return value;
}
```

**Note the pattern of usage for `initialized` and `value`. This pattern is required due to considerations of hardware design and possible “out of order” execution and store buffer reordering of writes.**

# TSD for Java

**This is one of the very few times where you will extend `Thread`. The logic here is that TSD is an integral part of the thread, so that’s where the code belongs. Note that you will still be running a `Runnable` object. If you manipulate that TSD from outside the thread, you’ll have to protect it.**

```
public MyThread extends Thread
{public Car car = new Car();
 public Cdr cdr = new Cdr();
 public House house = new House();
}

public MyRunnable implements Runnable
{...

  public run()
  {testDrive((MyThread)Thread.currentThread().car);}

}

Thread t = new MyThread(new Myrunnable());
t.start();
```

# TSD for Java

**In JDK1.2, there will be a solution for this:**

```
static ThreadLocal car = new ThreadLocal();

void testDrive()
{
  if (car.get() == null)
     car.set(new Car(“Bil’s Volvo PV”));
  ...
  Car bil = (Car) car.get();
  ...
}
```

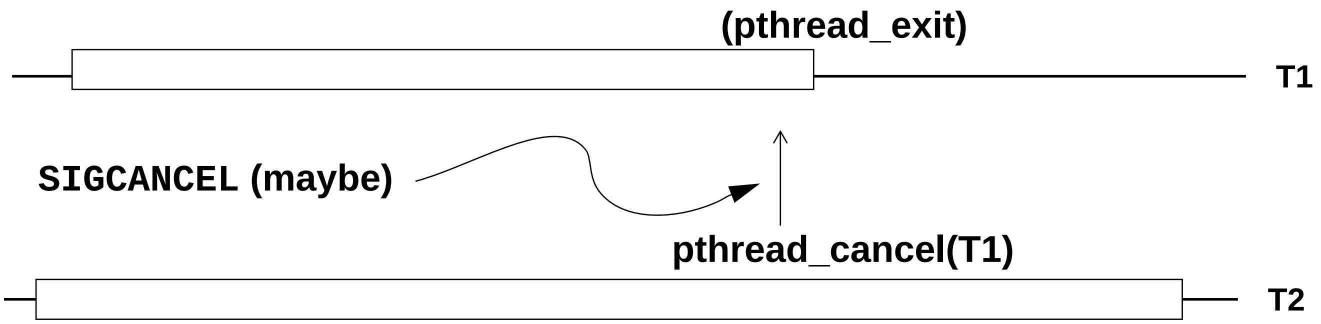
**This is really just a global hashtable maintained by Java:**

| | |
|---|---|
| **Thread-1** | **“Bil’s Volvo PV”** |
| **Thread-2** | **“Epp’s SAAB 9000”** |
| **Thread-3** | **“Bo’s Model T”** |

# *Cancellation*

# Cancellation

**Cancellation is the means by which a thread can tell another thread that it should exit.**



| **POSIX** | **Win32** | **Java** |
|---|---|---|
| `pthread_cancel(T1);` | `TerminateThread(T1);` | `thread.stop();` |

**There is no special relation between the creator of a thread and the waiter. The thread may die before or after join is called.**

**(Java specifies `thread.destroy()`, but never implemented it.)**

# Cancellation is a Dirty Word

**Unfortunately, cancellation will not clean up after itself. If you have changed part of a data structure when you get cancelled...**

```
yourBankAccount.debit(amount);
  Cancelled here? Too bad...
MasterCardsBankAccount.credit(amount);
```

**Any POSIX/Win32 mutexes which are held at the time of cancellation will remain locked. And there will be no (legal) way of ever unlocking them! Java synchronized sections will be released, but that doesn’t help with the cleanup problem.**

**Unrestricted, asynchronous cancellation is a bad thing.**

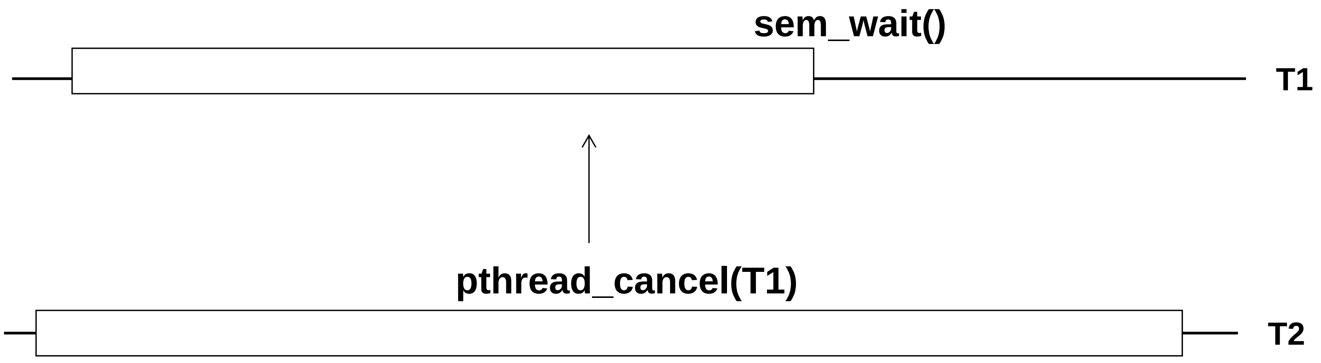
# Cancellation State and Type

- State
  - **`PTHREAD_CANCEL_DISABLE`**
    **(Cannot be cancelled)**
  - **`PTHREAD_CANCEL_ENABLE`**
    **(Can be cancelled, must consider type)**
- Type
  - **`PTHREAD_CANCEL_ASYNCHRONOUS`**
    **(any time what-so-ever -- not generally used)**
  - **`PTHREAD_CANCEL_DEFERRED`**
    **(Only at cancellation points)**

**Only POSIX has state and type (defaults: `PTHREAD_CANCEL_ENABLE`, `PTHREAD_CANCEL_DEFERRED`)**

**Java and Win32 are effectively always “enabled asynchronous”**

# PTHREAD_CANCEL_DEFERRED



```
T2                          T1                   Cancellation Point

{...                        {...                 sem_wait(...)
die[T1] = TRUE;             sem_wait(...)        { _wait()
pthread_kill(T1, SIGXXX);   ...                    if (die[self])
}                           }                        pthread_exit();
                                                   ...
```

**Above is kinda-sorta the idea. Deferred cancellation requires *cancellation points* (such as `sem_wait` or `pthread_testcancel`). *A sleeping thread will be woken up from a cancellation point.* A thread will run until it hits a cancellation point. All major blocking functions are cancellation points, though not mutexes.**

# Java Uses `thread.interrupt()`

Any thread can call `interrupt()` on any other thread. This is the intended analogue of Pthread's deferred cancellation. It sets a flag for the target thread, and... You can test for interruption directly:

```
boolean Thread.interrupted()    Also clears flag.
boolean thread.isInterrupted()  Doesn't clear flag.
```

and many methods (`wait(), sleep(),` etc.) will throw `InterruptedException`. I/O functions (`read()`, etc.) will throw `InterruptedIOException`.

Catching `InterruptedException` will clear the flag, catching `InterruptedIOException` will not.

Unimplemented in JDK 1.1. In JDK 1.2 major all significant blocking points (well...) will be interrupt points, though not synchronized sections.

cf: Run the programs `TestTimeout` and `ServerInterruptible.`

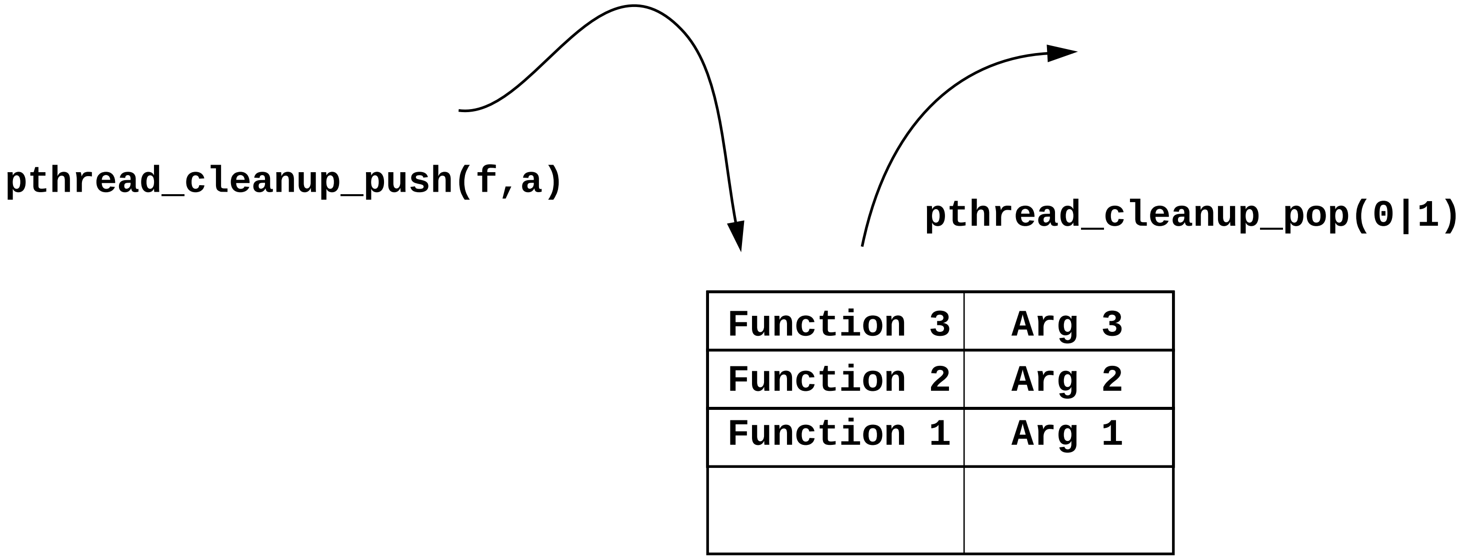
# Mutexes Are Not Interruptible

**In POSIX, a thread blocked, waiting to get a mutex is not at a cancellation point and will continue to sit there until it gets the mutex. Asynchronous cancellation MAY kill the thread (but that would be a bad idea).**

**In Java, a thread blocked, waiting to get a mutex is not at a interruptible point and will continue to sit there until it gets the mutex. Calling `thread.stop()` MAY kill the thread (but that would be a bad idea).**

**It’s just too !@#$!@ hard to allow mutexes to be interruptible (imagine inserting cancellation/interruption handlers at every mutex!), so they aren’t. You’re the programmer. You’re in full control. Release the mutex!**

# Cleanup Handlers



`pthread_cleanup_push(f,a)`

`pthread_cleanup_pop(0|1)`

| | |
|---|---|
| Function 3 | Arg 3 |
| Function 2 | Arg 2 |
| Function 1 | Arg 1 |
| | |

**Cleanup stack**

**Each thread has its own cleanup stack (presumably implemented as part of the normal call stack).**

**The handlers are called by `pthread_exit()` (i.e., with or without cancellation).**

**Nothing similar in Win32 or Java --** ***Big Problem!***

# CVs, `wait()` Must Reclaim Their Locks

When a thread is cancelled (interrupted/stopped) from a POSIX CV (Java wait set), that thread must reclaim the associated mutex (synchronized section) before proceeding.

POSIX cleanup handlers are then run with that lock held.

Any Java finally clause or catch block will be run with the lock held as is appropriate. I.E., this catch block will be run with the lock, but the finally clause won’t:

```
try {
  synchronized (this) {
      try   {while (!this.test()) this.wait();}
      catch (InterruptedException e) {
          System.out.println(e + “ Bummer!”);
      }}
finally
  {cleanup();}
```

# Sleepers

**In all of the deferred cancellation points, sleeping threads will be woken up in order to die. (Similar to the way that `read()` will be interrupted by a signal to return `EINTR`.)**

**`sem_wait()` is a cancellation point. As is `pthread_cond_wait()`, but not `pthread_mutex_lock()`.**

**When `pthread_cond_wait()` gets woken up to die, *it will reclaim the mutex* before it call `pthread_exit()`. This implies that you'd better have a cleanup handler to release that mutex!**

```
pthread_mutex_lock(&mutex);
pthread_cleanup_push(pthread_mutex_unlock, &mutex);

while (!condition)
  pthread_cond_wait(&cv, &mutex);
... do stuff ...

pthread_mutex_unlock(&mutex);
pthread_cleanup_pop(0);
```

# Does it *Have* to be so Complicated?

**Consider `pthread_cond_wait()`... What if POSIX didn’t require it to reacquire the mutex? Would that make things simpler?**

```
pthread_mutex_lock(&mutex);
while (!condition())
  pthread_cond_wait(&cv, &mutex);
... do stuff ...
pthread_mutex_unlock(&mutex);
```

**If there were a cancellation point inside of `condition()`, then clearly you would need to release the mutex. And we really don’t want to restrict `condition()` to having no cancellation points.**

**There is no way for us to know if we were cancelled inside of `condition()` or inside of `pthread_cond_wait()`, so we’d better have the same set of locks held in both cases.**

**If POSIX had decided that `pthread_cond_wait()` should not be a cancellation point, then we could avoid this bit. But it didn’t. So...**

***Yes, it does!***

# Cleanup Handlers

**Push & pop are macros and must be called from the same function (same lexical scope).**

```
foo()
{...
  pointer = (thing_t *) malloc(sizeof(thing_t));
  pthread_cleanup_push(free, pointer);
  ... lots of work ...
  free(pointer)                 // or: nothing here and...
  pthread_cleanup_pop(0);       // pthread_cleanup_pop(1);
...
}
```

**You cannot jump into or out of a push/pop context (although the compiler may not detect such a programming bug).**

# Cancellation and C++ Exceptions

**Some C++ compilers (Sunsoft's C++, who else?) produce code which works with cancellation, i.e., the C++ destructor forms will be called before the thread exits.**

**Each thread has its own definition for `terminate()` and `unexpected()`, which `set_terminate()` and `set_unexpected()` change per-thread. The default `terminate()` function is `abort()` for the main thread, and `pthread_exit()` for all others threads.**

```
T2
foo()
{my_class stack_object;
  ... lots of work ...
  ... lots of work ...                <-- Cancel!   (Destructor runs?)
  ... lots of work ...
}                                     <-- Destructor runs!
```

# Bounded Time Termination

**It is extremely difficult to guarantee a given bound on thread termination after a cancellation request. Normally this is not a problem.**

**Practically speaking, you can be fairly certain (99.99%?) that a thread will disappear within 10ms *CPU time for that thread* from the time the cancellation request completes (the function returns).**

**(Assuming minimal time spent in cleanup handlers.)**

**When will that cancelled thread get scheduled? Who knows!**

**How soon will it be able to obtain locks required for its cleanup handlers? Who knows!**

**For non-realtime programs, this is quite acceptable in practically all cases.**

**Realtime programmers... I don’t know realtime.**

# InterruptedException

Like cancellation, `InterruptedException` is difficult to handle correctly. You may wish to handle it locally:

```
void foo()
{
  try {wait();}
  catch (InterruptedException e) {do_something}
}
```

or, more likely, you may wish to simply propogate it up the call stack to a higher level. (Very likely to the very top!)

```
void foo() throws InterruptedException
{
  try {wait();}
}
```

# InterruptedException

**In any case, it is important not to drop an interruption. Your code may be used by some other package someday...**

**Bad Programmer:**

```
void foo() {
  try {wait();}
  catch (InterruptedException e) {}
}
```

**Good Programmer:**

```
void foo() {
  while (true) {
      try {wait(); return;}
      catch (InterruptedException e) {intd=true}
      }
  if (intd) Thread.currentThread().interrupt();
}
```

# Cancellation is Always Complex!

- **You must always wait for the cancelled threads, lest they continue to change data after you think they're gone.**
- **It is very easy to forget a lock that's being held or a resource that should be freed.**
- **Use this only when you *absolutely* require it.**
- **Almost impossible to use Async cancellation at all. Hence neither Win32 nor Java can (reasonably) cancel threads!**
- **Be extremely meticulous in analyzing the thread states.**
- **The concept of *Async cancel-safe* functions exists. Currently NO functions are listed as being Async cancel-safe. :-(**
- **Document, document, document!**

# Cancellation: Summary

## *Don’t do That!*

**Use polling instead of cancellation if you possibly can. Threads sleeping in I/O or in `wait()` can be awoken via signals or `thread.interrupt()`.**

**This is very difficult to do correctly in the first place. And when that guy gets hired to maintain your code...**

# *Thread-Safe Libraries*

# UNIX Library Goals

- **Make it Fast!**
- **Make it MT Safe!**
- **Retain UNIX Semantics!**

**(Pick any two)**

**(Actually, NT faces most of the same problems trying to juggle #1 and #2 as UNIX does.)**

# An Unsafe Library Function

Fri Sep 13 00:00:00 1986

Fri Sep 13 00:01:Fri19ep

```
foo()
{char *s;
 s=ctime(time);
 printf(s);
}
```

```
libc.so
```

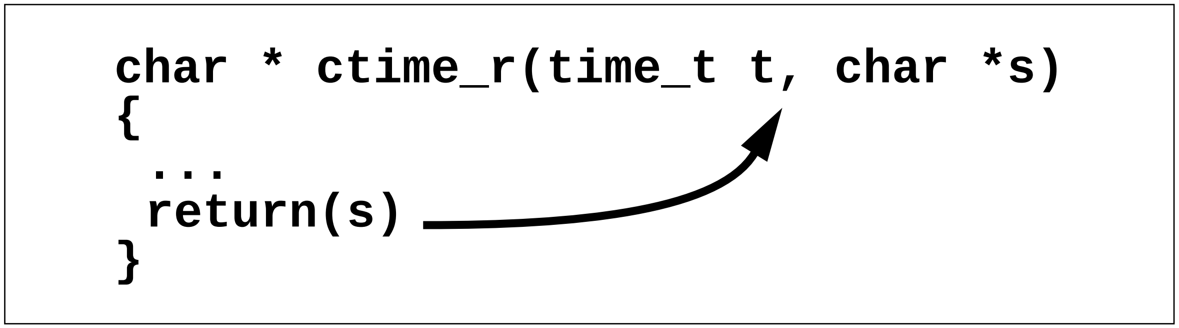
```
char *ctime(time_t time)
{static char s[SIZE];
 ...
 return(s);
}
```

# A Safe Library Function

**Fri Sep 13 00:00:00 1986** **Fri Sep 13 00:00:01 1986**

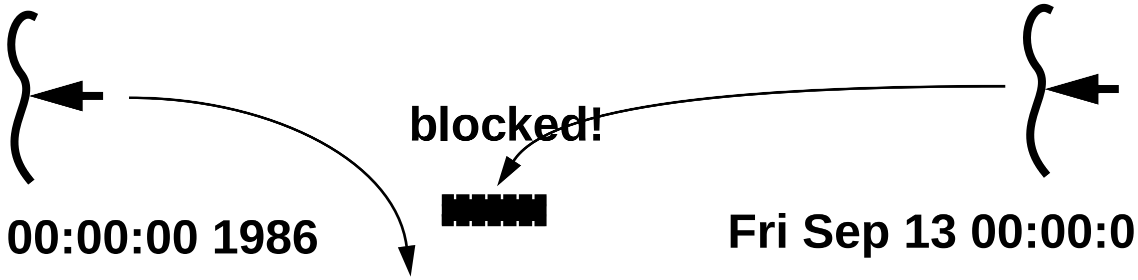
```
foo()
{char s[100];
 ctime_r(time, s);
 printf(s);
}
```

**stdio.so**



```
char * ctime_r(time_t t, char *s)
{
 ...
 return(s)
}
```

# Locking Around Unsafe Library Calls



Fri Sep 13 00:00:00 1986

Fri Sep 13 00:00:01 1986

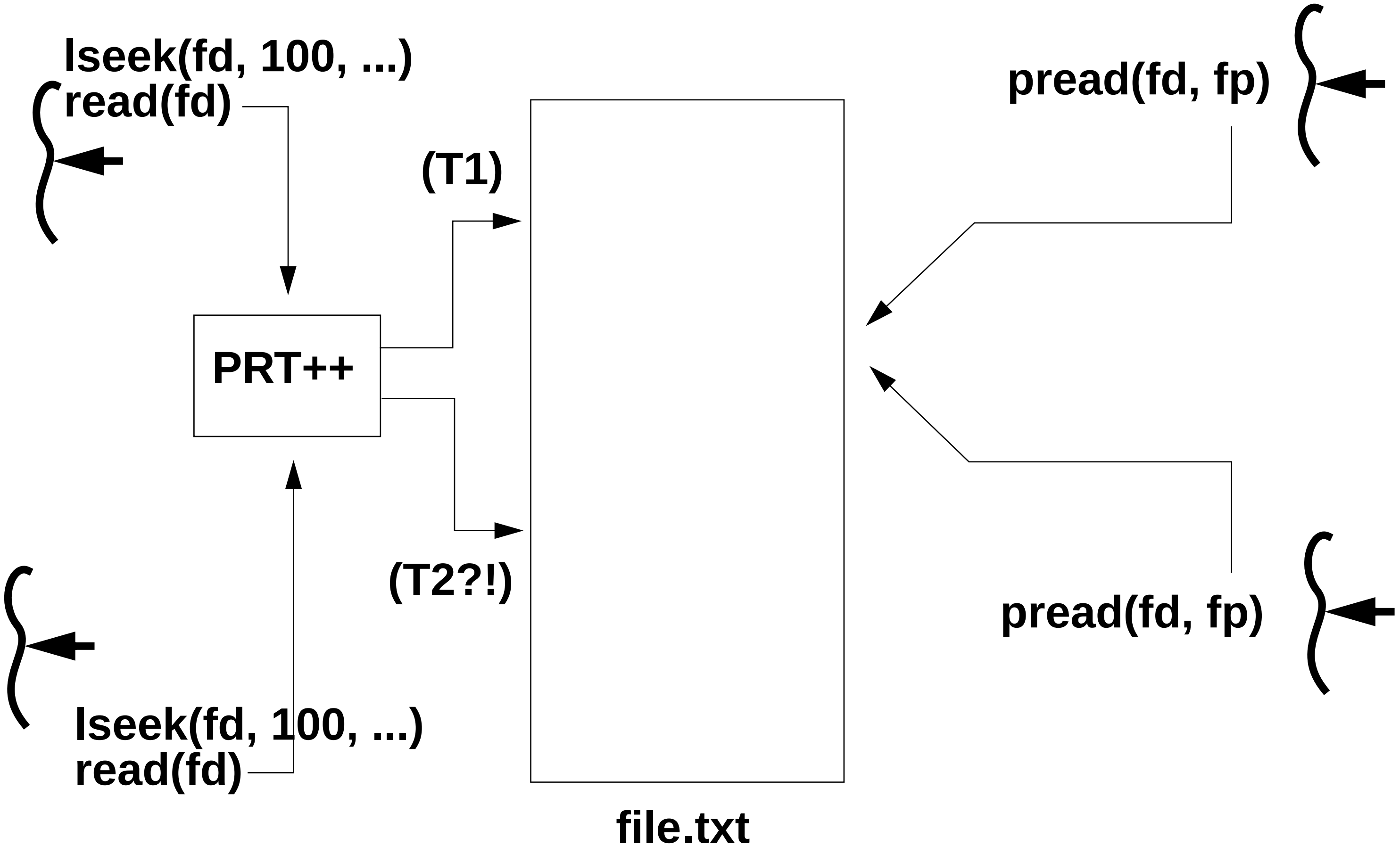
```
foo()
{char *s, s1[100];
 lock(m);
 s = ctime(time);
 strcpy(s1, s);
 unlock(m);
 printf(s1);
 }
```

```
MyFoo extends Foo
{
 synchronized String frob()
 {super.frob();}
...
}
```

stdio.so

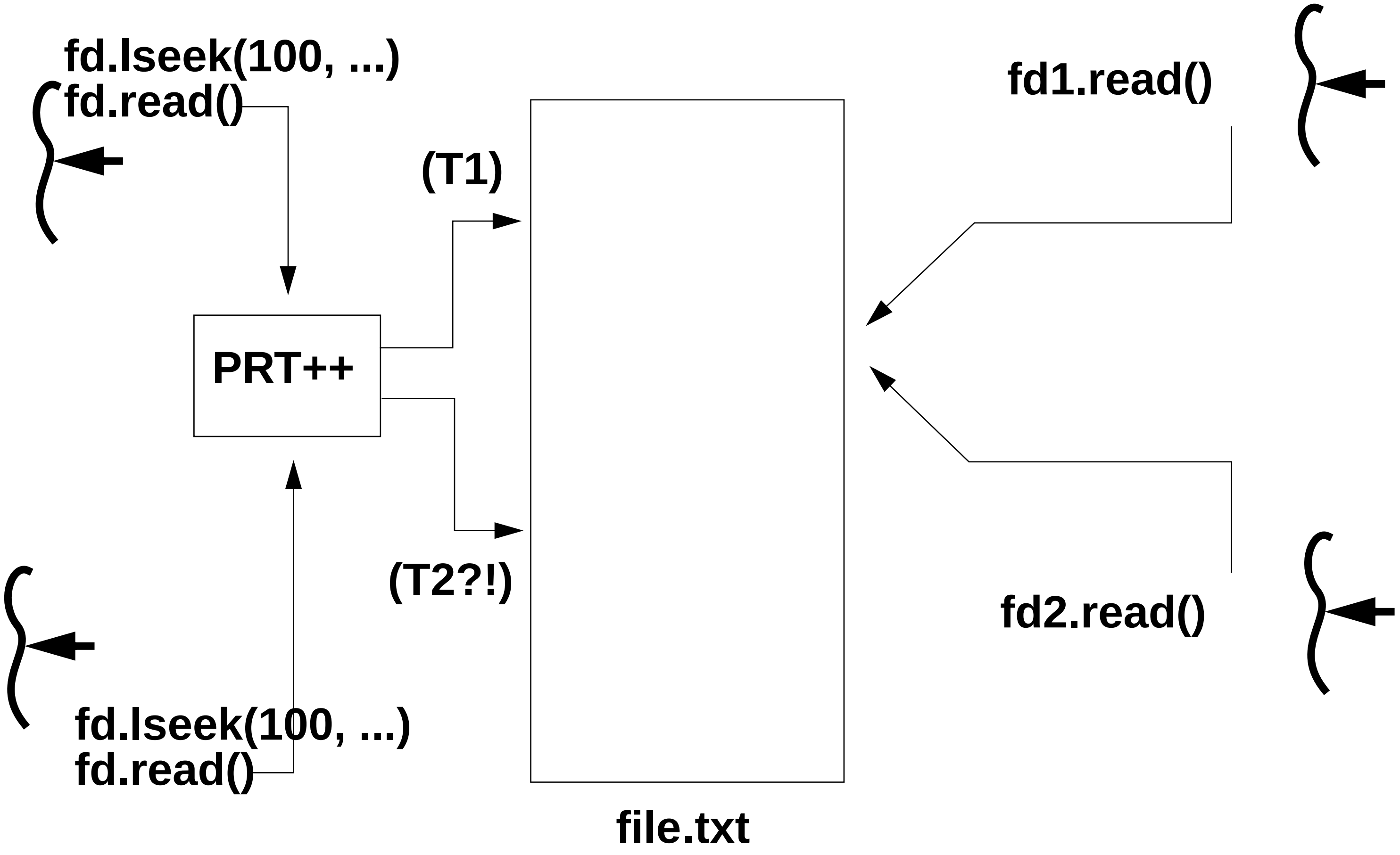
```
char *ctime(time_t time)
{static char s[SIZE];
 ...
 return(s);
}
```

# pread(), pwrite()



These are not POSIX (UNIX98), but exist in Solaris, HP-UX.

# read(), write() in Java



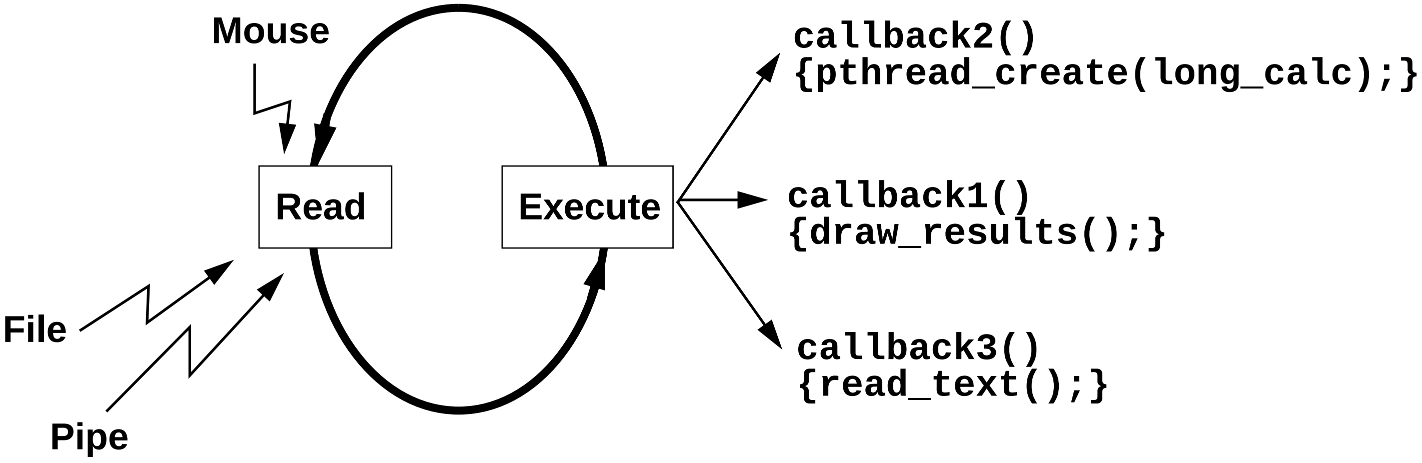
**Java has nothing similar, so you must open more fds.**

# Window Systems

**X11R6 itself is MT-safe, but not many toolkits.**

**XView, Motif and CDE Motif (for example) are not.**
**(CDE Motif 2.1 (Solaris 2.7) is MT-safe.)**

```
long_calc()
{for (i=0; i<Avocado’s_Number; i++)...
 write(Pipe);
}
```



```
callback2()
{pthread_create(long_calc);}

callback1()
{draw_results();}

callback3()
{read_text();}
```

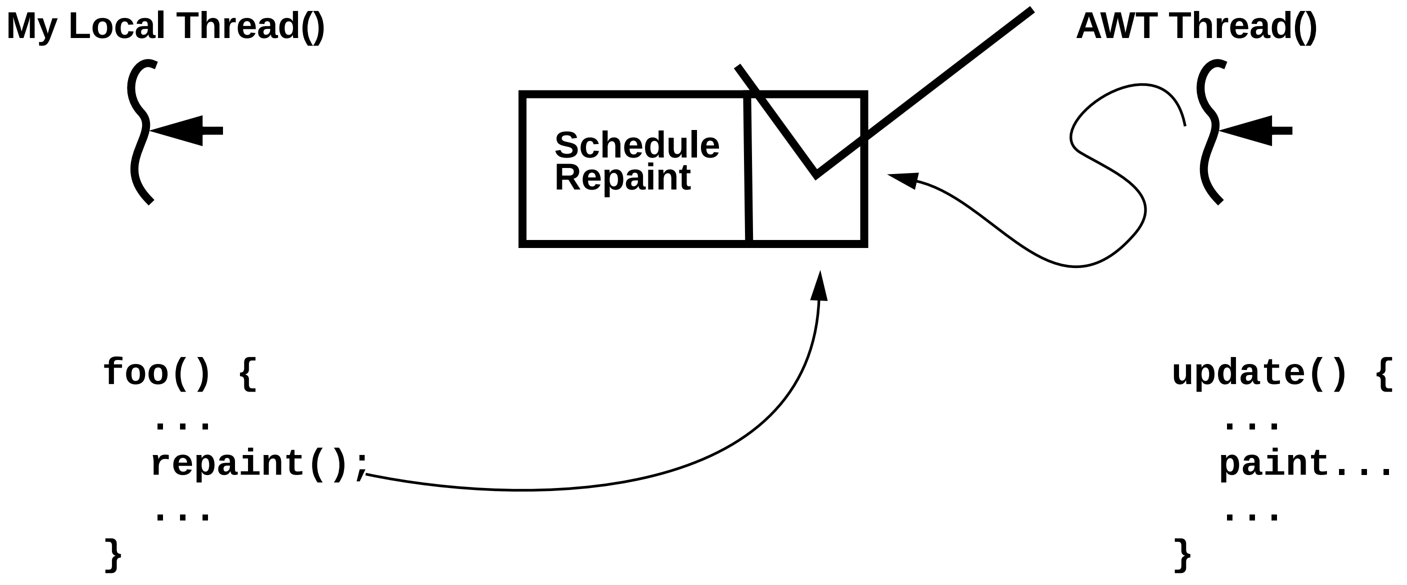
# Java’s AWT

**The AWT calls methods to update and repaint windows in the applet’s main thread. Thus, if you wanted to be doing something while the repaint was going on, you’d have to do that from another thread.**

**For example, if you wanted to sleep for a bit..**

```
private void reset()     // Runs in separate thread
  {
     chosenCell.brighten();
     repaint();

     // Sleep so the user can see the bright color
     goToSleep(sleepTime);

     // Redraw the square in its original color
     chosenCell.darken();
     repaint();
     ....
```

# Java’s AWT



My Local Thread()

AWT Thread()

Schedule Repaint

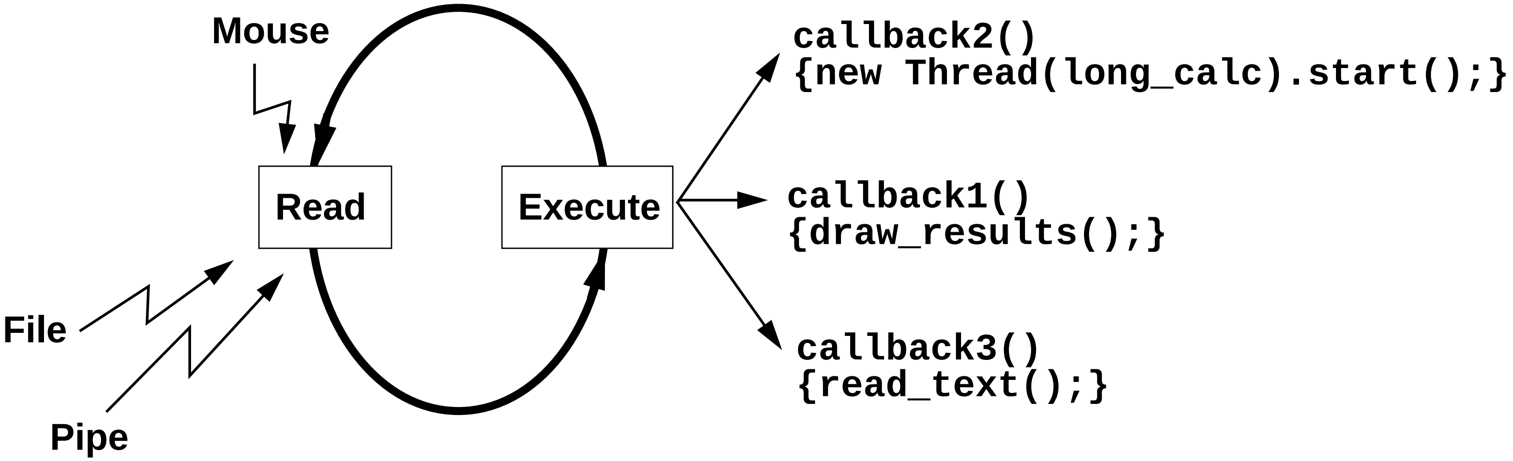
```
foo() {
  ...
  repaint();
  ...
}
```

```
update() {
  ...
  paint...
  ...
}
```

# Java’s Swing

**Swing is specifically not MT-safe. Hence it is faster than AWT. The way to thread a Swing application is similar to what you do for CDE. It provides you with `invokeLater(Runnable)` and `invokeAndWait(Runnable)`.**

```
long_calc()
{for (i=0; i<Avocado’s_Number; i++)...
 invokeLater(callbackRunner);
}
```



```
callback2()
{new Thread(long_calc).start();}

callback1()
{draw_results();}

callback3()
{read_text();}
```

# Java 2 Collection Classes

**List, Set, Map, Hashmap. All methods are not synchronized, making them faster. If you want synchronized methods, you create a new class which the collection class will make for you:**

```
MyList = Collection.synchronizedList(list)
```

| **Thread1** | **Thread2** |
|---|---|
| `myList.get()` | `MyList.put(x)` |

***You probably will never use synchronized collections.***

# errno

In UNIX, the distinguished variable `errno` is used to hold the error code for any system calls that fail.

Clearly, should two threads both be issuing system calls around the same time, it would not be possible to figure out which one set the value for `errno`.

Therefore `errno` is defined in the header file to be a call to thread-specific data.

This is done only when the flag `_REENTRANT` (this flag is valid only for UI threads and early POSIX drafts) `_POSIX_C_SOURCE=199506L` (POSIX) is passed to the compiler. This way older, non-MT programs to continue to run as before, while threaded programs get a TSD value.

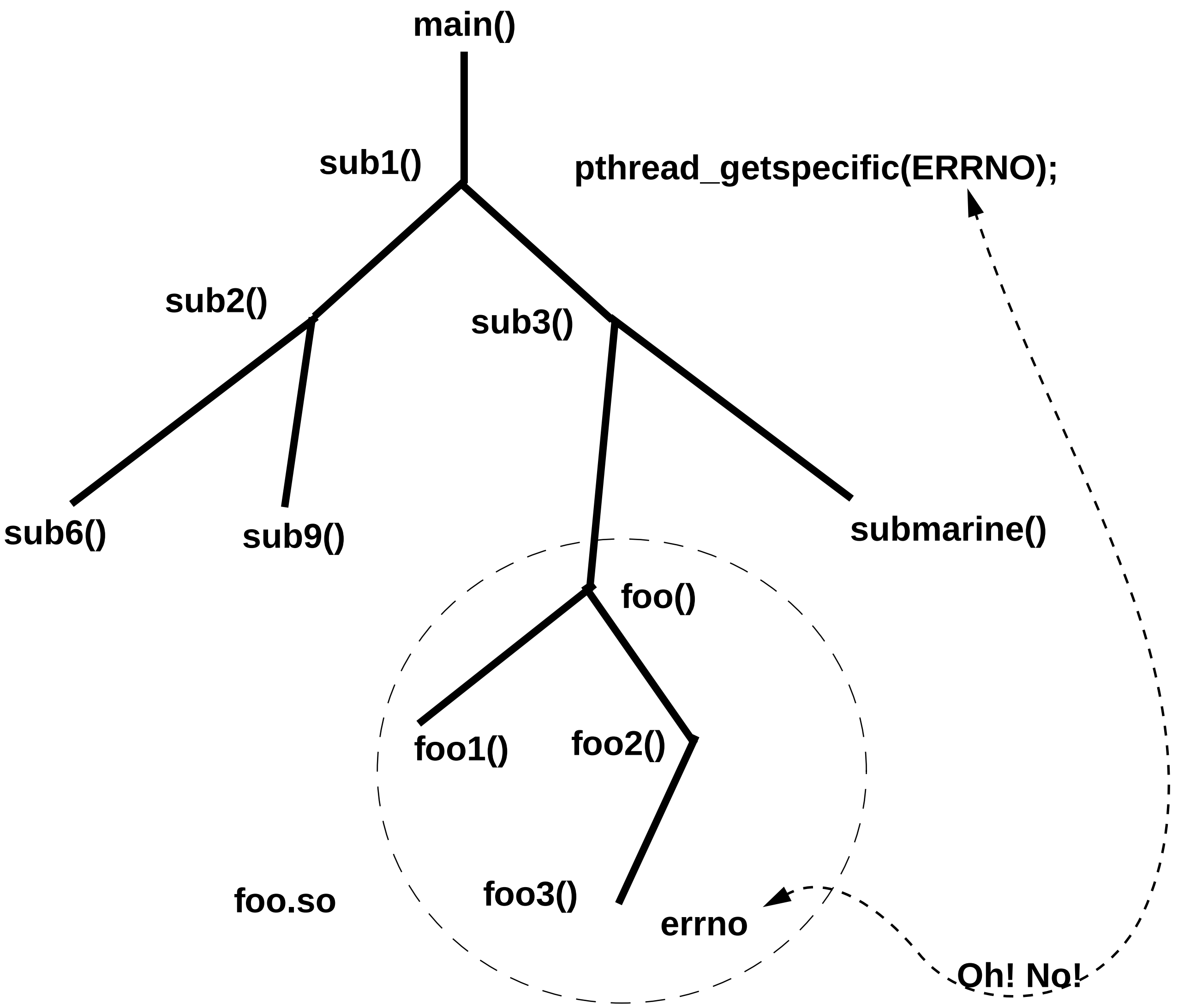
There is the potential for problems should some libraries which you are using not be compiled reentrant.

# errno Now

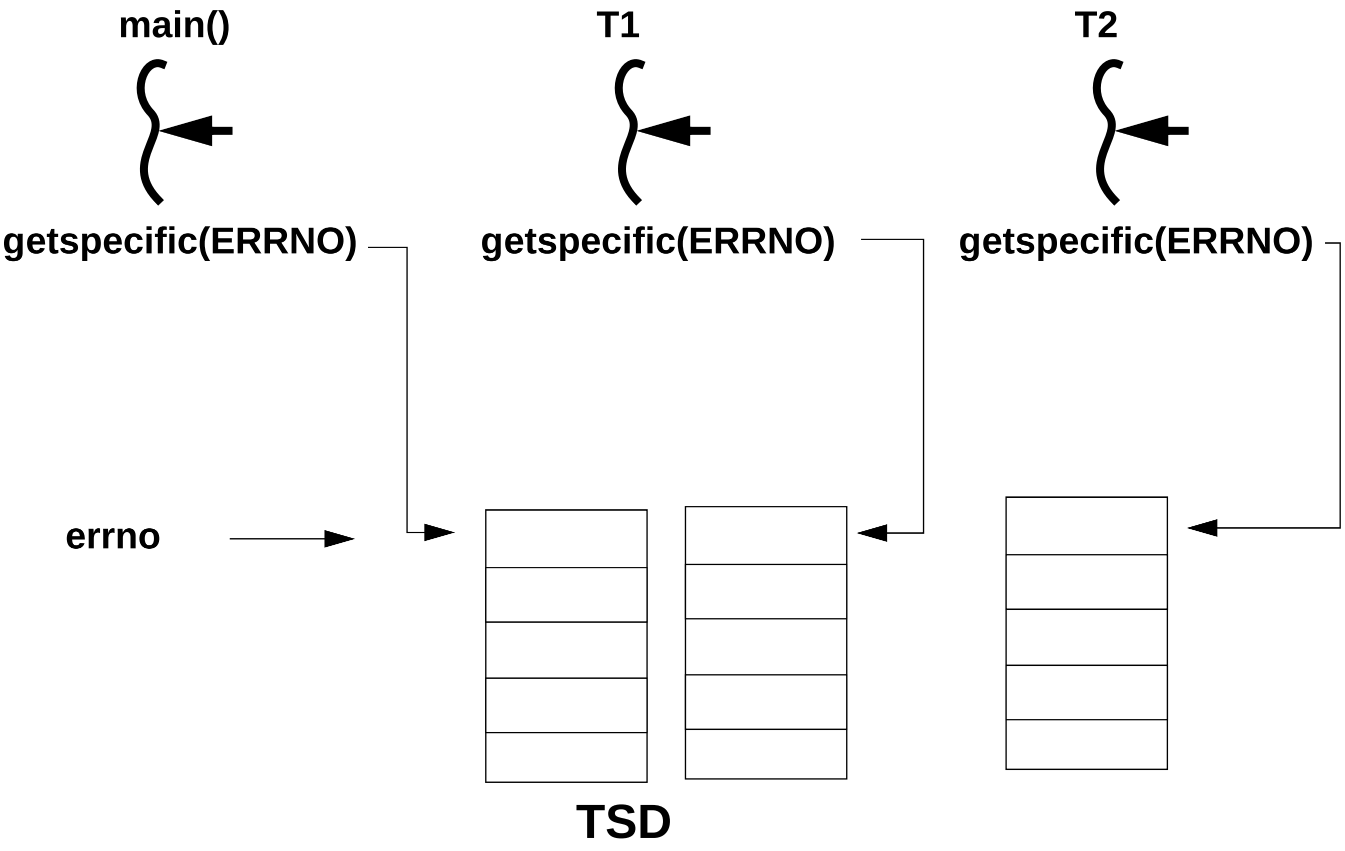
```
#if     (defined(_REENTRANT) || defined(_TS_ERRNO) \
    ||  _POSIX_C_SOURCE - 0 >= 199506L)          \
    &&  !(defined(lint) || defined(__lint))


extern int *___errno();
#define errno (*(___errno()))


#else

extern int errno;


#endif/* defined(_REENTRANT) || defined(_TS_ERRNO) */
```

**(actual code from Solaris' `errno.h`)**

# errno Problems



# errno Solutions



**main_thread: getspecific(ERRNO_KEY) == errno**

**(Solaris only)**

# getc() Back Then

```
#define    getc(p) (--(p)->_cnt < 0 ? \
                   __filbuf(p) : (int)*(p)->_ptr++)
```

**(actual code from stdio.h)**

# getc_unlocked() Now

```
#define    getc_unlocked(p) (--(p)->_cnt < 0 ? \
                   __filbuf(p) : (int)*(p)->_ptr++)
```

**(actual code from stdio.h)**

# getc() Now

```
#if defined(_REENTRANT) || (_POSIX_C_SOURCE - 0 >= 199506L)

  extern int getc(FILE *p)
  {static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

    pthread_mutex_lock(&m);
    i = (--(p)->_cnt < 0 ? \
          __filbuf(p) : (int)*(p)->_ptr++)
    pthread_mutex_unlock(&m);
    return(i);
  }

#else

  #define    getc(p) (--(p)->_cnt < 0 ? \
                       __filbuf(p) : (int)*(p)->_ptr++)

#endif
```

**(not actual code)**

# Are Libraries Safe?

| | |
|---|---|
| MT-Safe, Safe | Safe to call from multiple threads. |
| MT-Hot | This function is safe and fast. |
| MT-Unsafe | This function is not MT-safe, but was compiled reentrant. |
| Alternative Call | This function is not safe, but there is a similar function (e.g., `ctime_r()`). |
| MT-Illegal | This function wasn’t even compiled reentrant and therefore can only be called from the main thread (Solaris). |
| Async-Cancel Safe | OK to cancel the thread while in this fn. |
| Async Safe (aka:) | OK to call this function from a handler. |
| (Fork Safe) | OK to call from the child before `exec()`. |

The Solaris man pages will only list MT-safe (or safe), Async-safe, and MT-unsafe. No Solaris libraries are MT-Illegal. For third party libraries however...

# Reentrant (definitions)

**This term is used differently sometimes. I have seen it used to mean:**

- **MT-Safe (most common, that’s how I use it)**
- **Recursive**
- **Signal-Safe**

**Any combination of the above are possible and different from any other combination! Recursive & Signal-Safe != MT-Safe != Recursive != MT-Safe & Recursive & Signal-Safe etc.**

# Stub Functions in `libc.so`

**Because many library functions need to operate correctly in both threaded and non-threaded programs, many of the threads functions have to be available at run time. But we don’t want them to do anything!**

```
printf(...)
{static pthread_mutex_t M;
  pthread_mutex_lock(&M);
  _printf(...)
  pthread_mutex_unlock(&M);
}
```

**So stub functions are defined in `libc.so` for pthread mutexes and pthread CVs. They just return -1.**

**If `libpthread.so` is linked in, its definitions are found before the `libc.so` stub functions. (If you link `libc.so` explicitly, you must place it after `libpthread.so`)**

# Functions Without Stubs

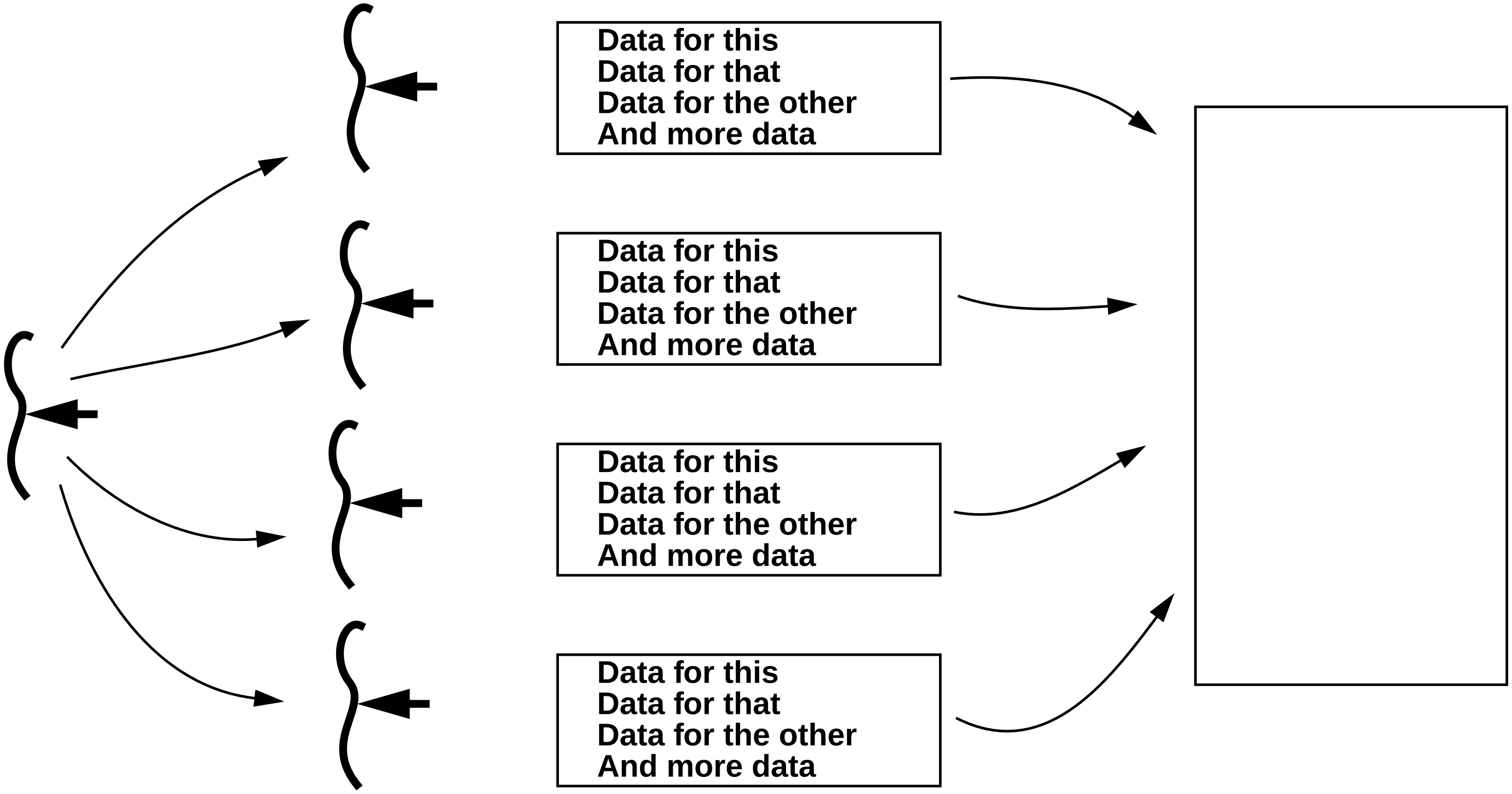
**Use `#pragma weak`, which defines a symbol to be 0 if it does not have any alternative definition at run time:**

```
#pragma weak pthread_create

fft(array, size)
{
if (pthread_create != 0)
  for (i=0; i<size; i++)
    pthread_create(..., do_fft_row, i);
else
  for (i=0; i<size; i++)
    do_fft_row(i);
}
```

**(Solaris only)**

# *Program Design*

# Master/Slave (1)



Data for this
Data for that
Data for the other
And more data

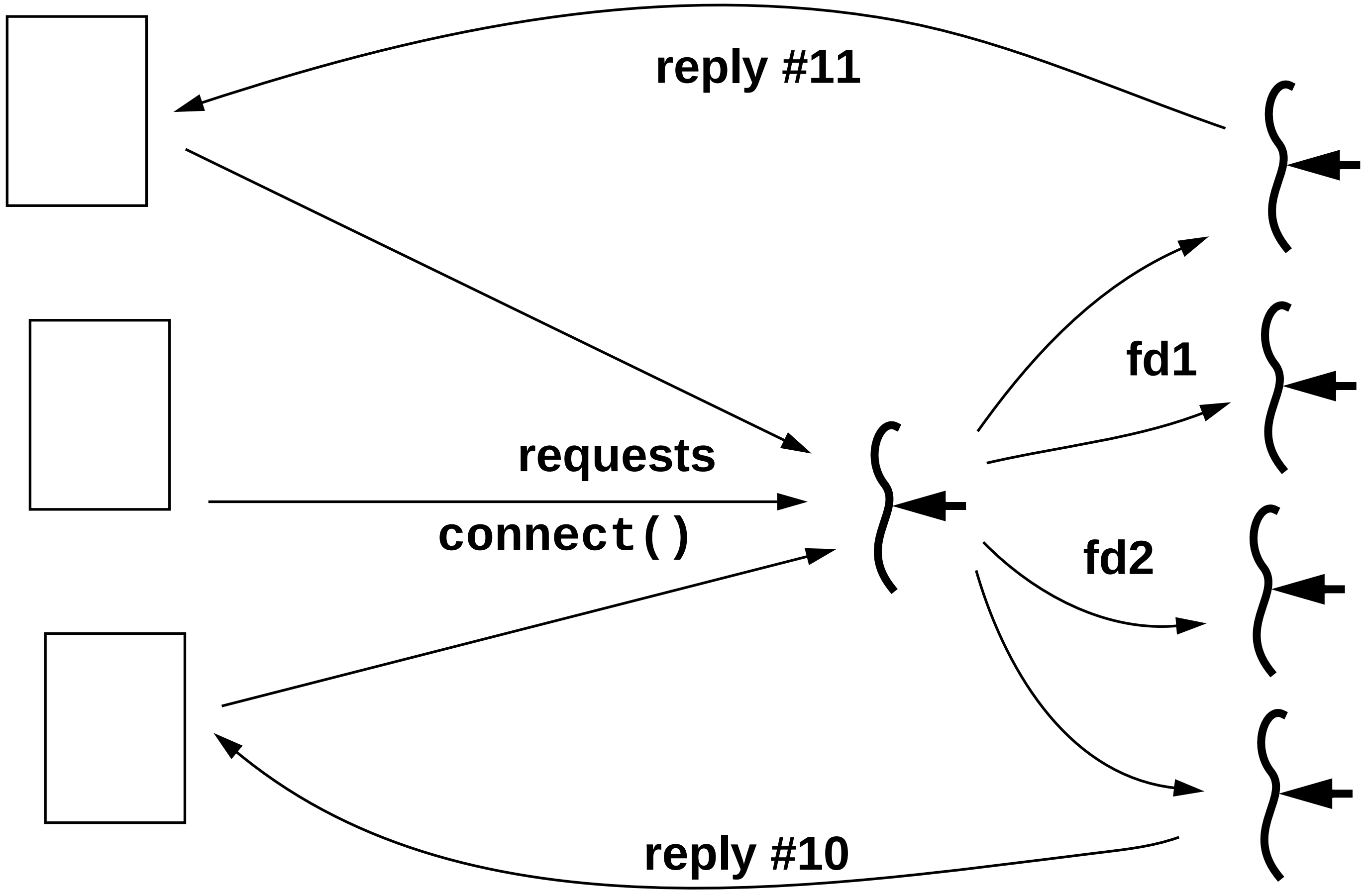
Data for this
Data for that
Data for the other
And more data

Data for this
Data for that
Data for the other
And more data

Data for this
Data for that
Data for the other
And more data

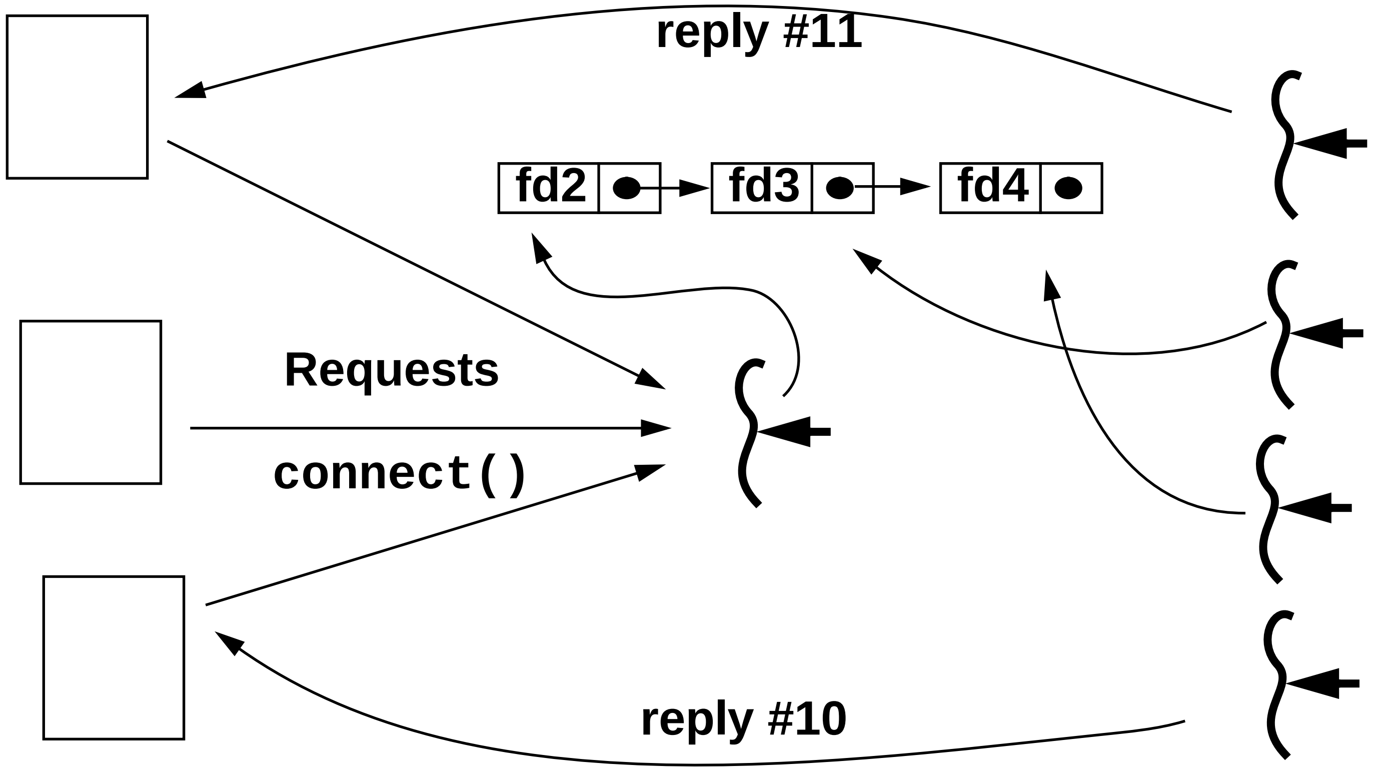
**Master thread simply calls the API, which creates (saves?) threads as it deems useful.**

# Master/Slave (2) “Thread per Request”



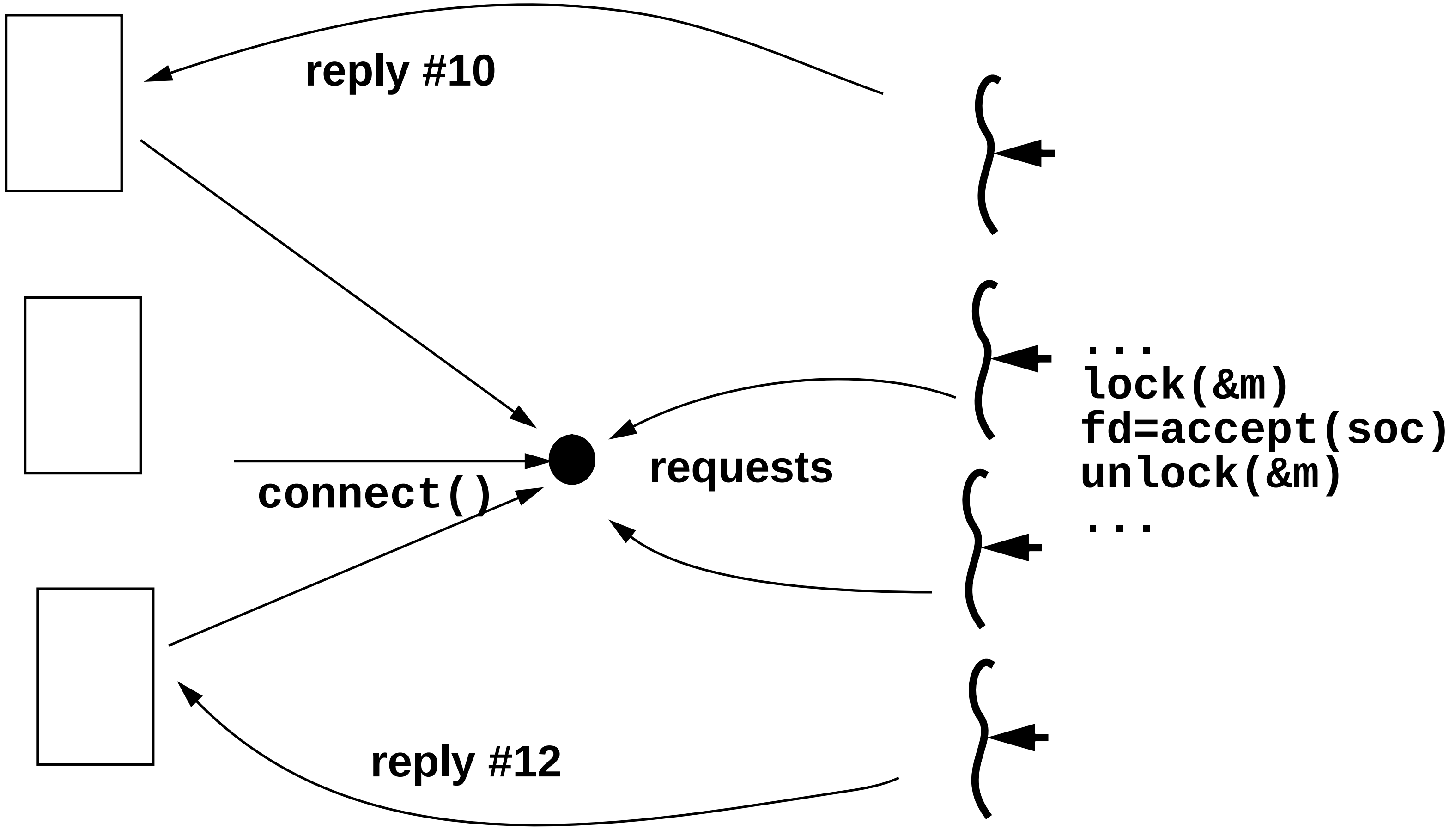
**Worker thread exits after serving the one request (see server.c, Server.java)**

# Workpile
# (Producer/Consumer)



**Worker threads take requests from queue, execute them, then go back for more requests**
**(see server_pc.c, ProducerConsumerServer.java).**

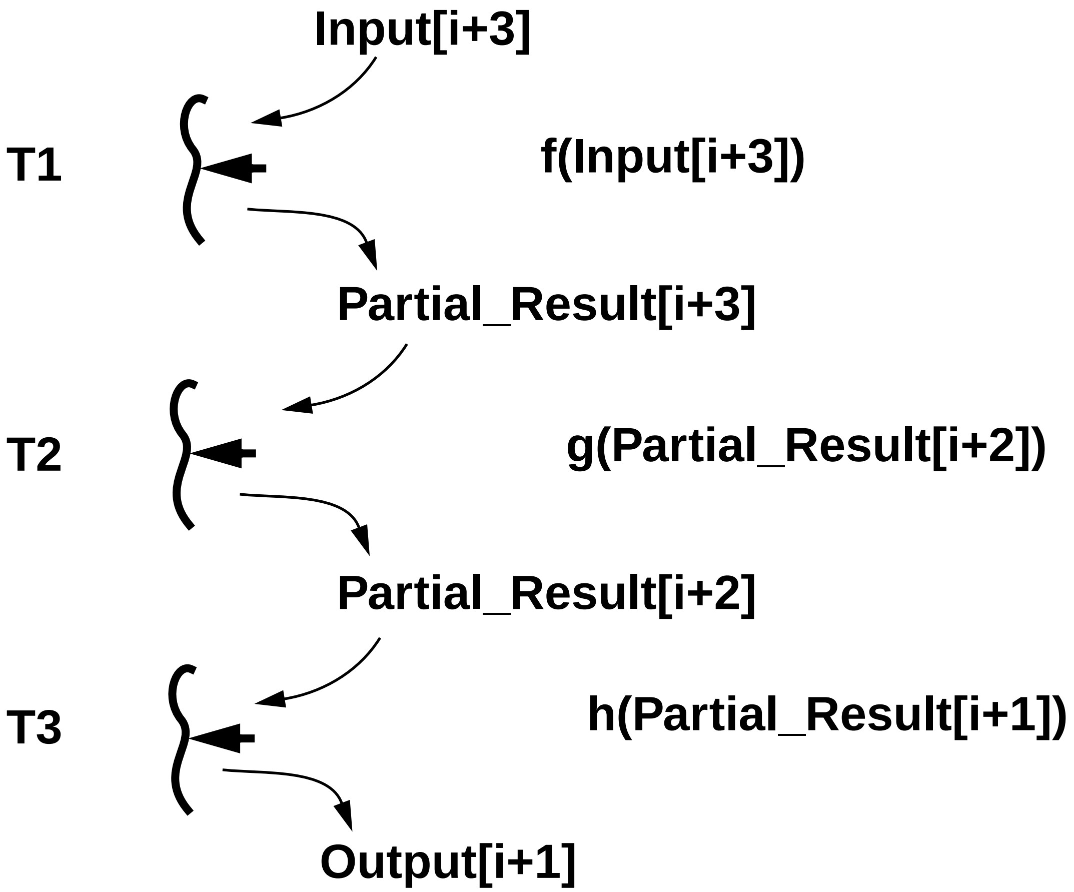
# Dogpile (Pigpile)



```
...
lock(&m)
fd=accept(soc)
unlock(&m)
...
```

**Producer/Consumer where `accept()` does the queuing in the kernel.**

# Pipeline



Input[i+3]

T1

f(Input[i+3])

Partial_Result[i+3]

T2

g(Partial_Result[i+2])

Partial_Result[i+2]

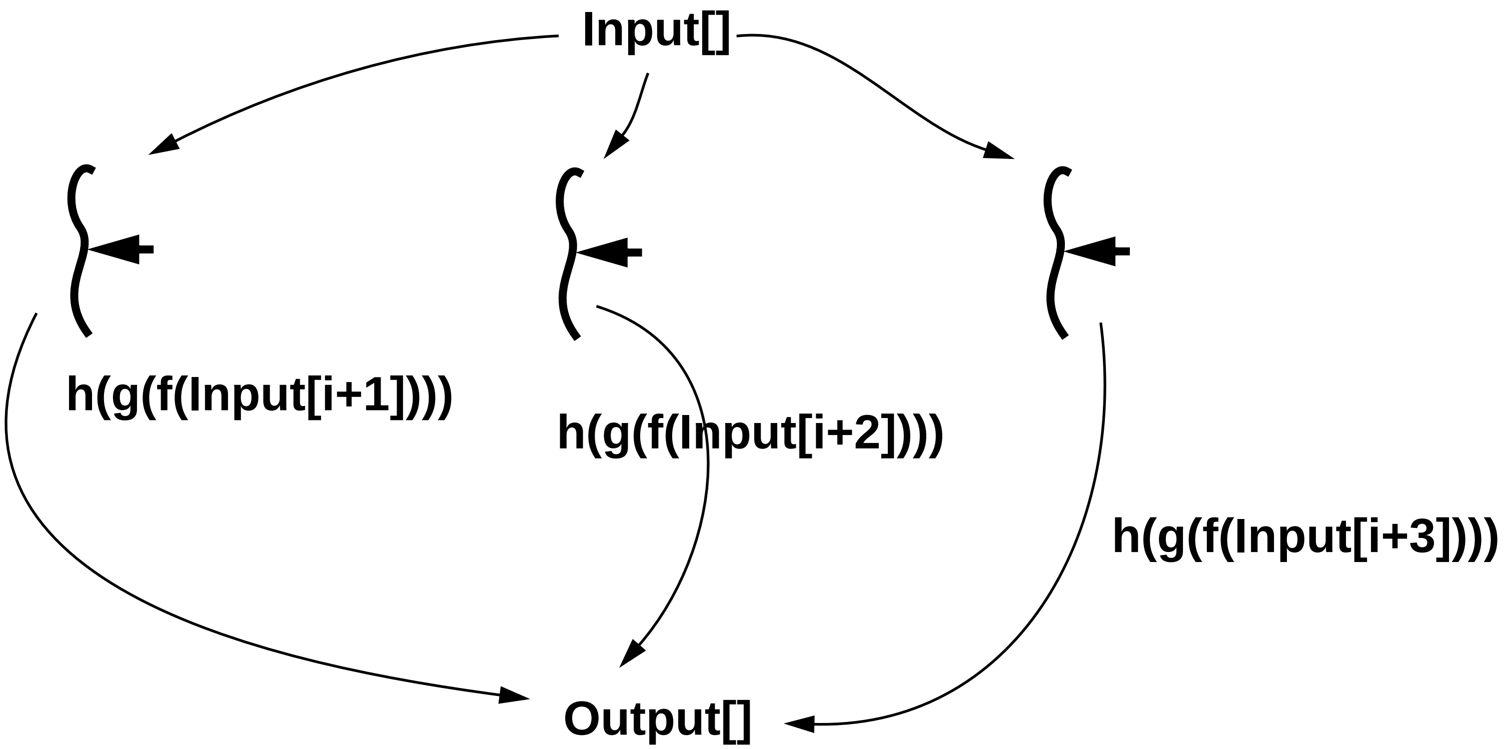
T3

h(Partial_Result[i+1])

Output[i+1]

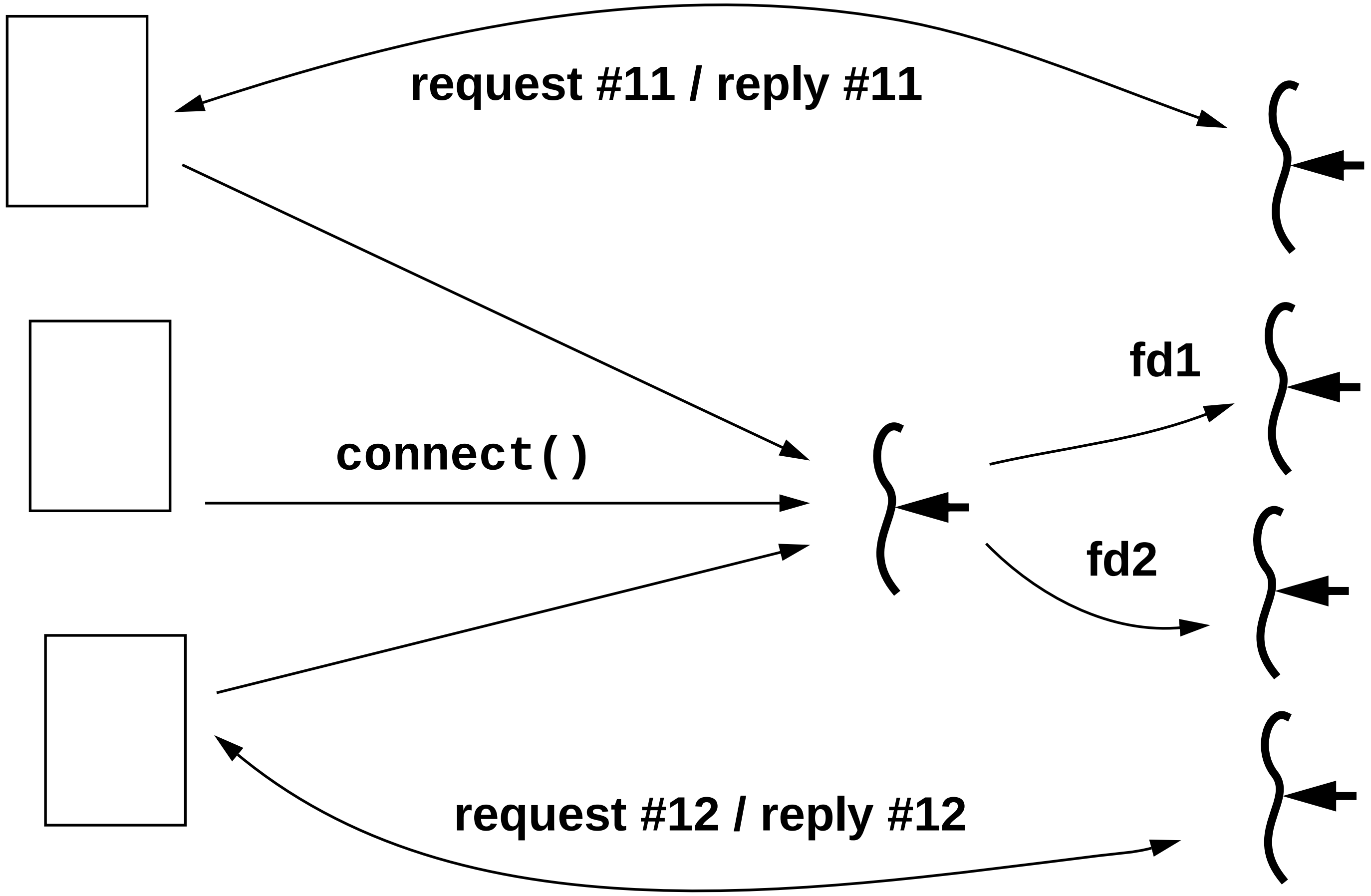
**Each thread executes a function on a *portion* of the input. T1 executes f() on the input, T2 executes g() on the result of that, and T3 executes h() on that.**

# Pipeline Alternative



Executes same job, but with lower overhead and complexity. Can also be designed as a Producer/Consumer.

# “Thread per Client”



**Master/Slave (2), but the worker threads service the client forever.**

**Java does not have a `select()` call! (yet)**

# *Making Libraries Safe & Efficient*

# Making Libraries Safe

- Trivial functions: No globals and only “stupid” statics
- Declared globals
    - Pass a structure to the function (e.g. `ctime_r()`)
    - Return a new structure from function
- Simple global state
    - `rand()` type functions
- Thread specific state
    - `rand()` type functions
    - `strtok()` type functions
- Dealing with excessive contention
    - Monte Carlo
    - `malloc()`

# Possible `rand()` Implementations

```
rand_1()                          /* about 3us */
{ static int seed;
  static mutex_t m;
  int i;

  lock(&m);
  i = _rand(&seed);             /* Updates seed */
  unlock(&m);
  return(i);
}



rand_2()                          /* about 2us */
{int *seed;

  seed = getspecific(SEED_KEY);
  i = _rand(seed);              /* Updates seed */
  return(i);
}
```

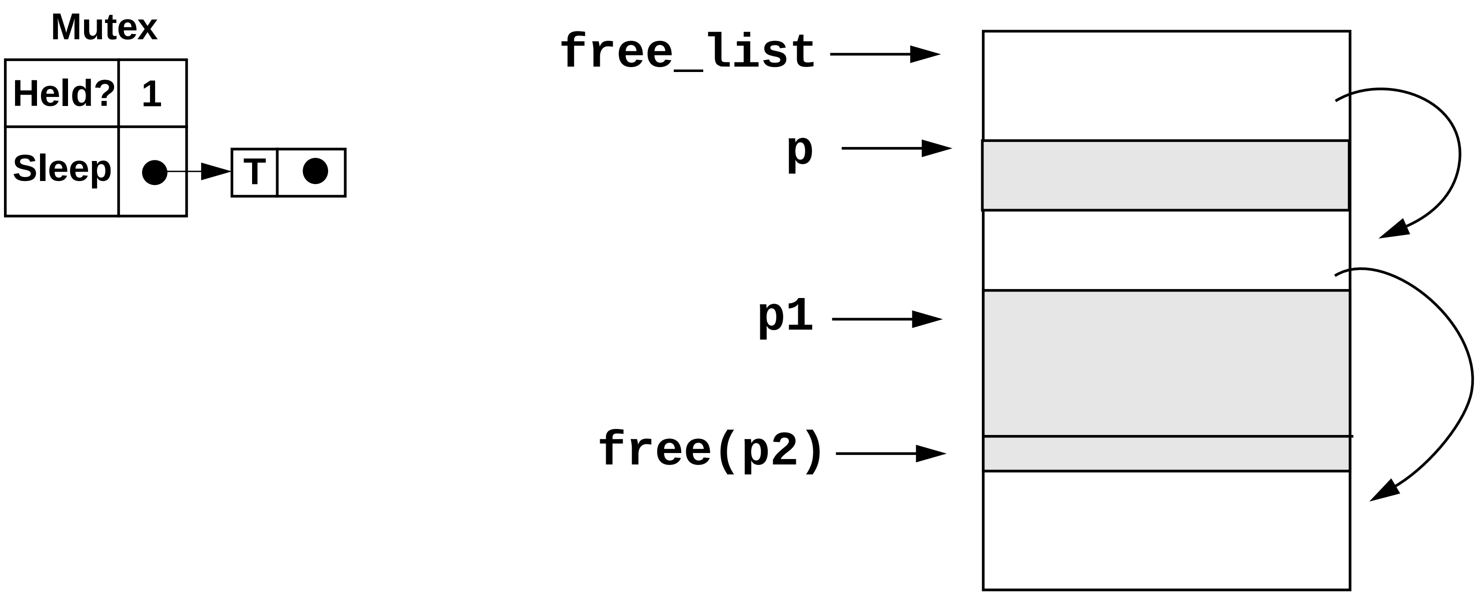
# Actual POSIX `rand()` Definition

```
foo()
{unsigned int seed;           /* You keep track of seed */
 int ran;

  ran = rand_r(&seed);        /* about 1us */
  ... use ran ...
}


rand_r(int *seed)
{int i;
  i = _rand(seed);            /* Updates seed */
  return(i);
}
```

**Meaning that you have to keep track of your own value of seed (yuck!)**

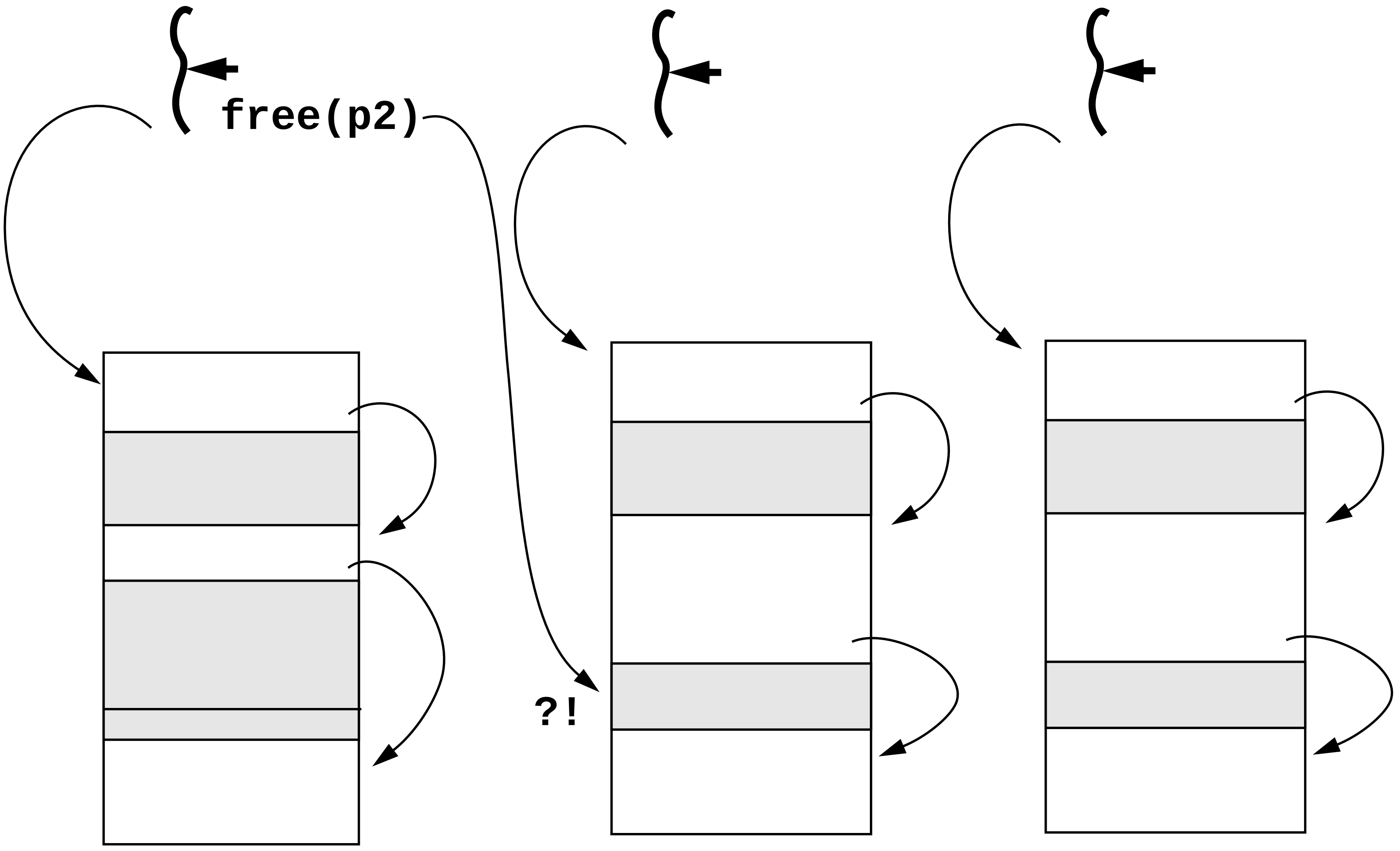
**(You may wish to implement #1 or #2.)**

# Current `malloc()` Implementation



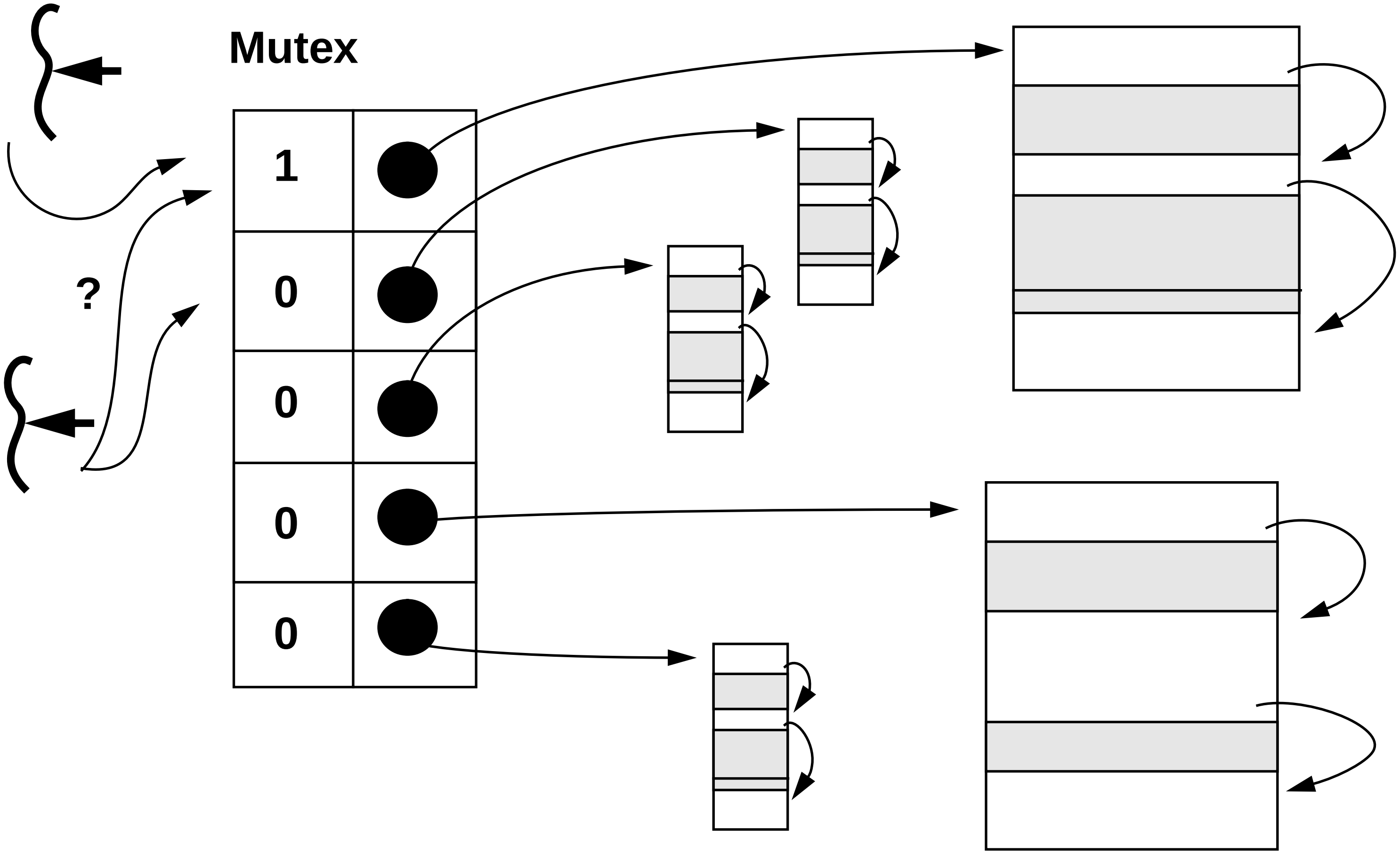
**(Potential for a lot of contention.)**

# `malloc()` using TSD



**No contention, but complex if threads exit often, or share data.**

# `malloc()` with Many Heaps



**Less contention than #1, simpler than #2. Easier to optimize also.**

**How to implement `free()`?**

# *Signals*

*(UNIX only)*

# The Three Uses of Signals

- Signals are used to indicate that an error occurred in the program (SIGFPE, SIGSEGV, SIGBUS, etc.). Aka *Synchronous Signals.*
    - These are sent when the CPU traps on an illegal operation.
- Signals are used to tell the program that something external has occurred and that the program should take some additional action. (SIGHUP, SIGPOLL, SIGURG, SIGWINCH, etc.)
    - These are sent by another program (Asynchronous).
- Signals are used to curtail the current actions of the program (SIGALRM, SIGKILL, SIGSTOP, etc.) (Asynchronous)
    - SIGKILL and SIGSTOP can be sent by another program.
    - SIGALRM is requested by the program itself.

# Handling The Three Uses of Signals

- ***Traps*** **must be handled in a signal handler by the guilty thread.**
    - **(This is the only logical choice, and the one POSIX made.)**
- ***Informational*** **signals may be handled anywhere, by any thread.**
    - **(Also the logical choice, also the one POSIX made.)**
- ***Interruption*** **signals ought to handled by intended thread.**
    - **(But ensuring this is too tough, and POSIX took the easy way out. You have to write this yourself.)**
    - **Because POSIX defines cancellation and suspension is not a option, the only signal of interest here is SIGALRM. (OK, there are always exceptions, but in MOST cases, it’s only SIGALRM.)**

# Signal Delivery

- Single-threaded programs run exactly as they did before
- When an asynchronous signal is sent to a multithreaded program:
    - The signal is delivered to the *process*
    - The *library* decides which thread will handle the signal
    - There is a single, per-process handler routine (or action) for each signal
    - Threads have individual signal masks, which the library looks at to decide the recipient
    - A sleeping thread may be awoken to run a handler, after which it will return to sleep (if on a mutex), or return -1 (semaphore), or just return normally (CV).
- The process-level signal mask is unused and undefined. Don't call `sigprocmask()`.

# Signal Delivery Details

- **Asynchronously generated signals:**
  - **(e.g., `SIGALRM`, `SIGINT`, `SIGSTOP`)**
  - **delivered to *a* receptive thread (You don’t know which one!)**
  - **remain pending when masked**
- **Synchronously generated signals (traps)**
  - **(e.g., `SIGSEGV`, `SIGBUS`, `SIGFPE`)**
  - **delivered to thread that caused the trap**
- **Locally generated signals via `pthread_kill()`**
  - **one thread can send a signal to another thread**
  - **they are asynchronous signals and treated as such (i.e., you don’t know exact time of delivery, you can’t count them, and they will remain pending if masked out)**

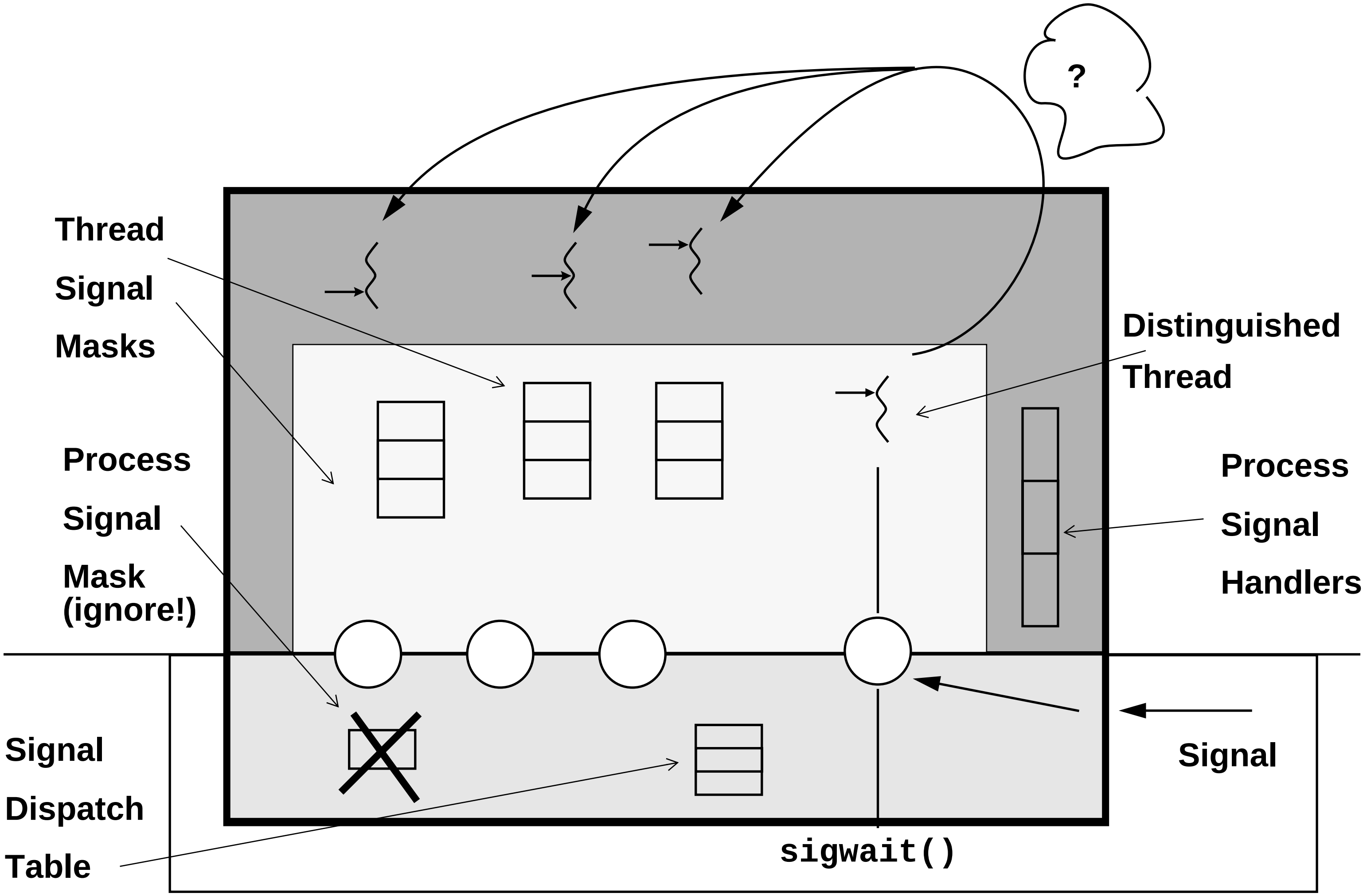
# Signal Delivery Transition Details

**In Solaris 2.4 and earlier (i.e., before the ratification of POSIX) alarms and timers were defined to be per-LWP. In POSIX this is not so, and Solaris 2.5 allows you to choose the semantics you desire. However at some distant point in the future, the old semantics will be dropped.**

**We recommend working strictly with the POSIX semantics.**

**(If you don’t have any UI programs, you can ignore this slide.)**

# Solaris Implementation Details



**This implemenation was used in 2.5. It’s different in 2.6!**

# sigwait()

**This function sits around waiting for a signal to be sent to the process. It then receives the signal synchronously, and lets the programmer decide what to do with it.**

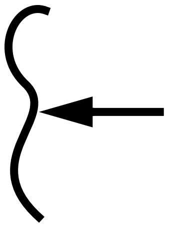
```
pthread_sigmask(BLOCK, signal_set, ...);       /* Always! */
while(1)
 {sigwait(signal_set, sig);
  switch (sig)
    { case SIGALRM :thr_expired_timer();
                       break;
      case SIGUSR2 :thr_timer_setalarm();
                       break;
      ...etc...
      }
   }
```

**In POSIX, the actions `SIG_DFL` and `SIG_IGN` take precedence over `sigwait`, so they must be turned off by creating an empty handler (Ugh! -- see callout.c).**

**You must block the signals from *all* threads.**

# Signal (Async) Safety

**Because a signal can interrupt a program at any point, most of the concurrency issues faced by threaded programs are also faced by interrupt handlers. This problem is compounded by the danger of self-deadlock. (This problem is not unique to threaded programs.)**



```
                              foo()
                              {                               handler()
                              lock(M1);                       {
SIGNAL COMES HERE ->           ...                            lock(M1);          Deadlock!!
                              unlock(M1);                     unlock(M1);
                              }                               }
```

# Very Few Functions are Signal Safe

- Among them is `sem_post()`
    - i.e., You may awake a sleeping thread from a handler
- You can make most functions async safe by blocking all signals while making the call
    - `pthread_sigmask()` is relatively fast (~5μs on an SS10/40).
    - Some functions (e.g., `strtok()` carry state across invocations) and cannot be make both safe and “reasonable”
- If you must write signal handlers, look up functions you wish to use -- the Solaris man pages indicate async-safety level.

# USE SIGWAIT() !

# Signal Safety

**`sem_post()` (`sema_post()` for UI) is unique in being the only synchronization function which can be called from a signal handler without any special protection.**

**This is because the semaphore functions automatically mask out all signals. The effective code is:**

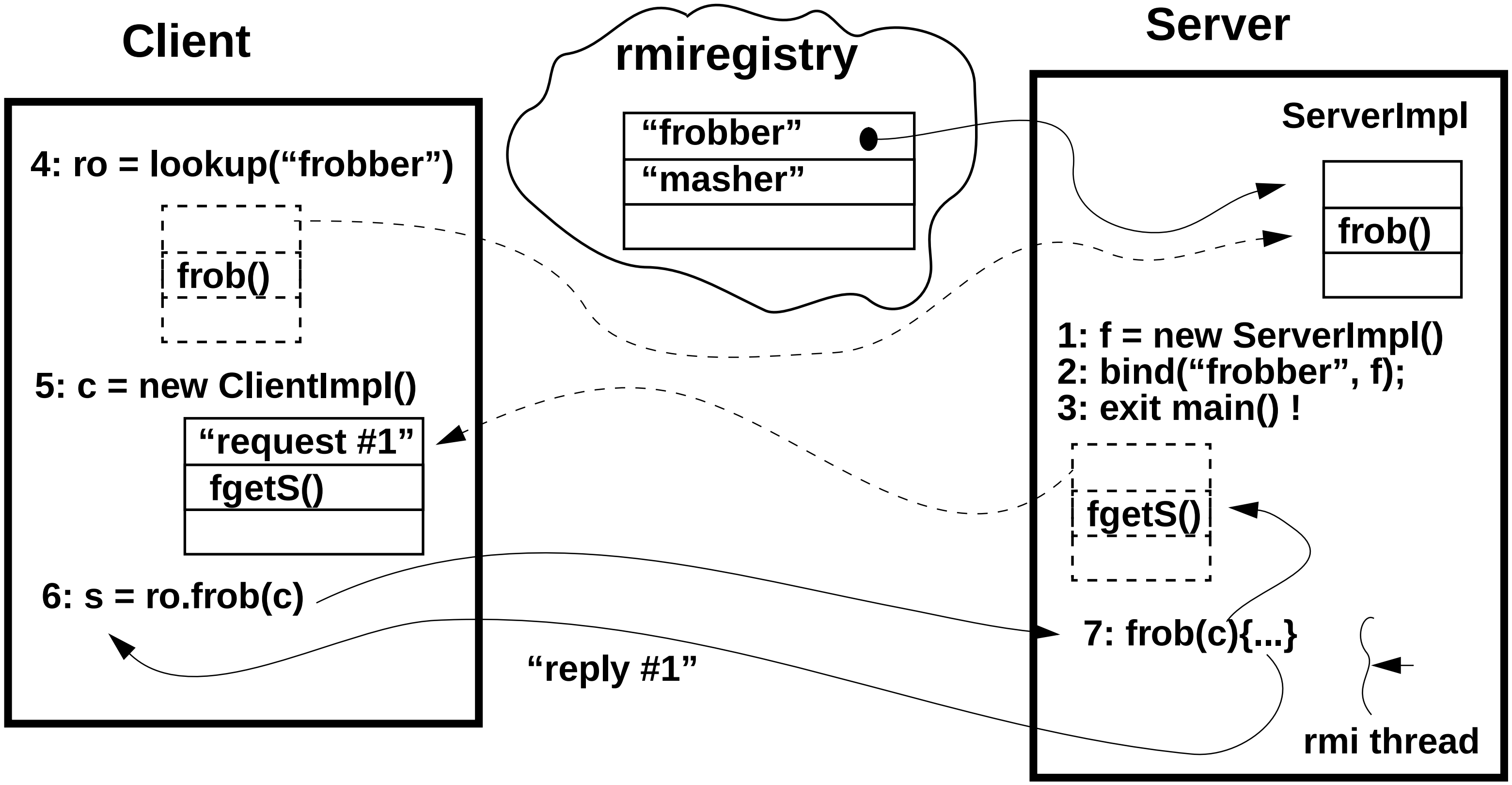
```
sem_post(sem_t *s)
{sigset_t new, old;

  sigfillset(&new);
  pthread_sigmask(SIG_BLOCK, &new, &old);
  _sem_post(s);
  pthread_sigmask(SIG_SETMASK, &old, NULL);
}
```

**This technique can be used generally for any function. It will slow you down by an extra ~5µs.**

# *Other Related Issues*

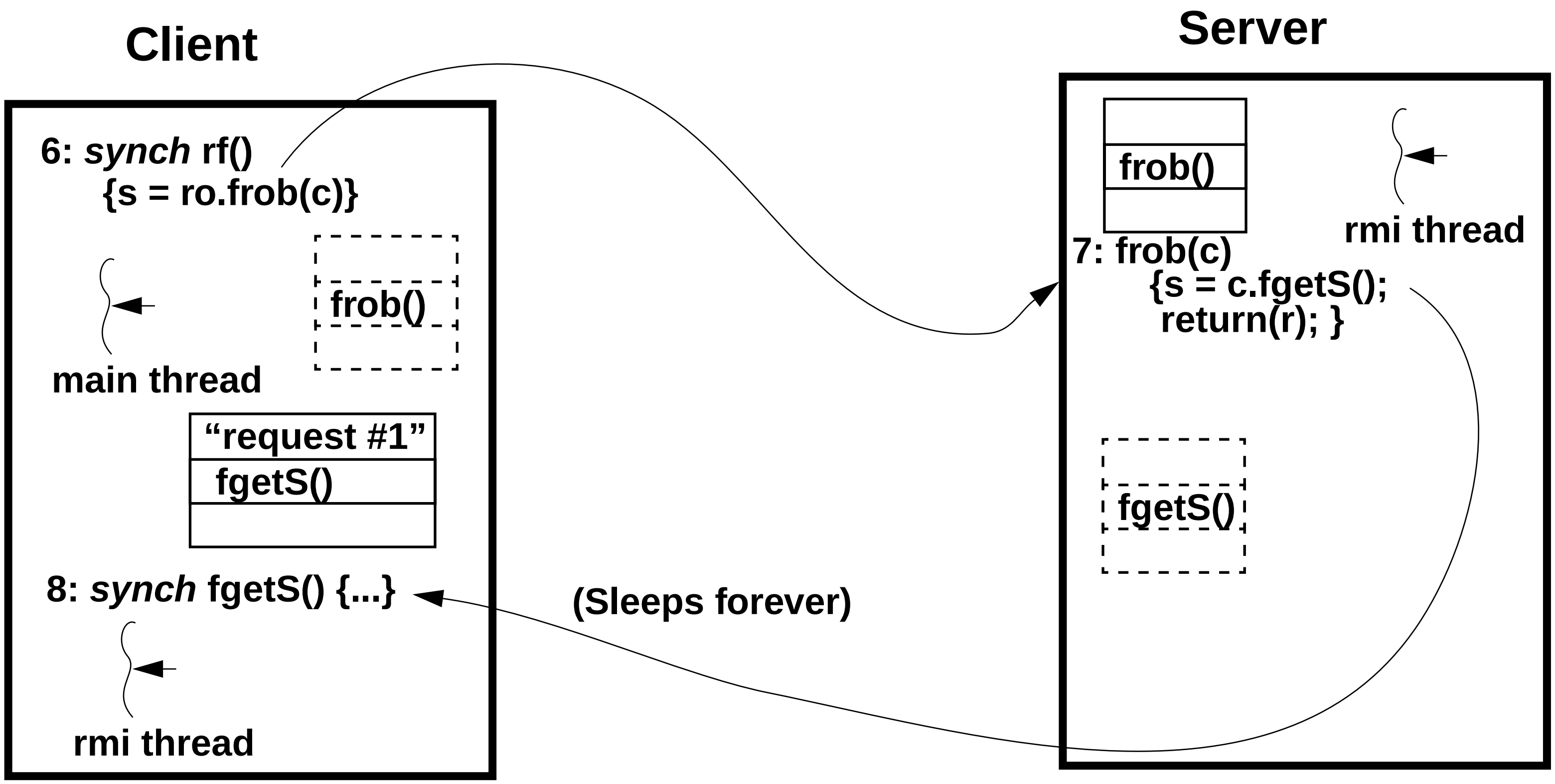
# Java Remote Method Invocation

RMI is not part of the threads interface of course, but (a) many client/server programs will use RMI, and (b) RMI creates its own threads.

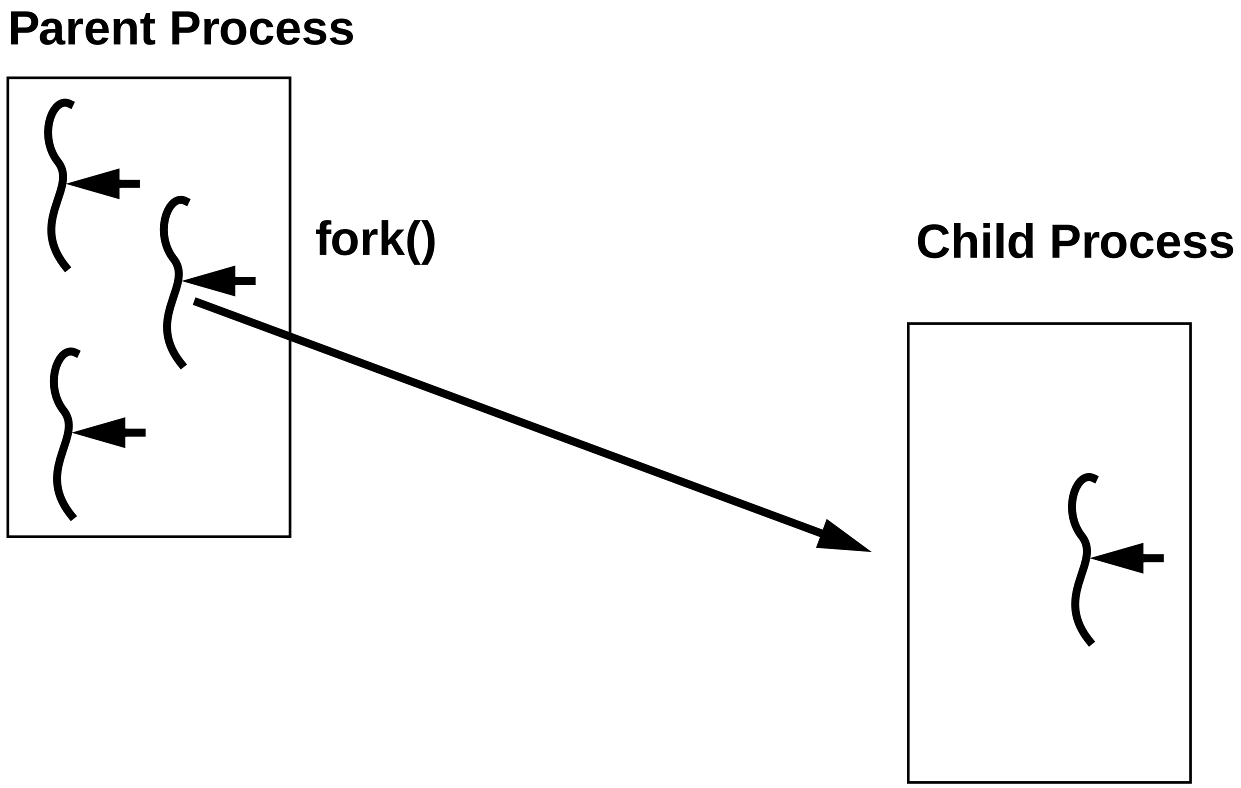


# Synchronized Callbacks in RMI

**A remote call (hence any kind of callback) runs in a different thread than the initial call was made from. If the two methods are synchronized on the same object...**



# New `fork()` Semantics



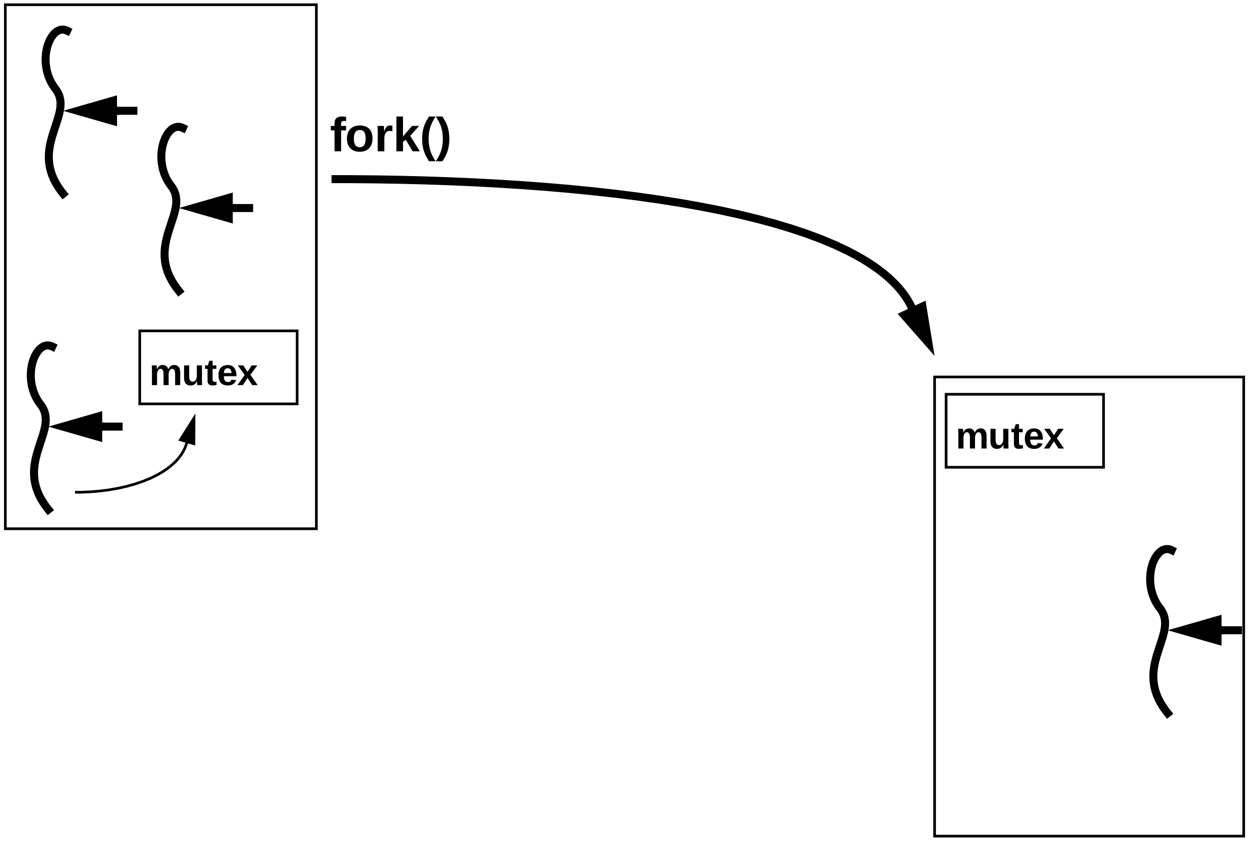
Parent Process

fork()

Child Process

**POSIX only copies the thread that called `fork()`**

# Potential `fork()` Problems



- **No owner in the child process!**

# Avoiding FORK Problems

- **You can ensure that the process is in a known state at the time of the actual fork by using `pthread_atfork()`.**
    - **Unfortunately, functions such as `printf()` and `malloc()` have private mutexes which you can’t grab.**

```
pthread_atfork(prepare, parent, child);

void *prepare()           // Grab all important locks
{ pthread_mutex_lock(&M1);
  pthread_mutex_lock(&M2);
}

void *parent()
{ pthread_mutex_unlock(&M1);
  pthread_mutex_unlock(&M2);
  printf(“Parent process continuing”);
}

void *child()
{ pthread_mutex_unlock(&M1);
  pthread_mutex_unlock(&M2);
  printf(“Child process starting”);
}
```

# Avoiding FORK Problems

**Probably the easiest way of dealing with this is to write a synchronization function yourself which you call when your threads get to “safe” locations:**

```
tell_everyone_to_wait(here);      should_i_wait(here);
fork();                           ... waiting ...
tell_everyone_to_continue(here);  ... continues ...
```

**In general you should avoid the problem by always calling `exec()` right after `fork()`.**

# `fork()` Transition Issues

- In UI threads, `fork()` would copy all threads, while `fork1()` would copy only the creator.
- You can select which semantics you wish to use for `fork()` in Solaris 2.5 and above via compiler flags:
  - `_POSIX_C_SOURCE`
    for strictly POSIX programs
  - `_POSIX_PTHREAD_SEMANTICS`
    for UI thread programs that want POSIX semantics for `fork()`, timers, and alarms.
- We recommend using strict POSIX APIs and semantics everywhere.

**(If you don’t use UI, ignore this slide!)**

# Threads Debugging Interface

- Debuggers
- Data inspectors
- Performance monitors
- Garbage collectors
- Coverage analyzers

**Not a standard interface! HP, IBM, SGI, etc. all will be different.**

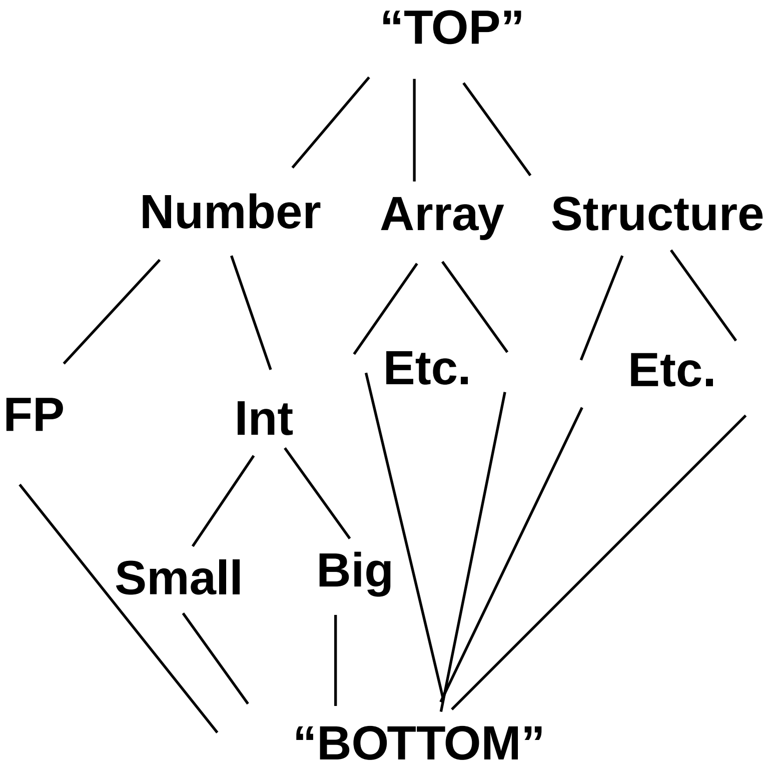
**Sun’s: `libthread_db.so`**

**Never use this anywhere except in special cases such as those above.**

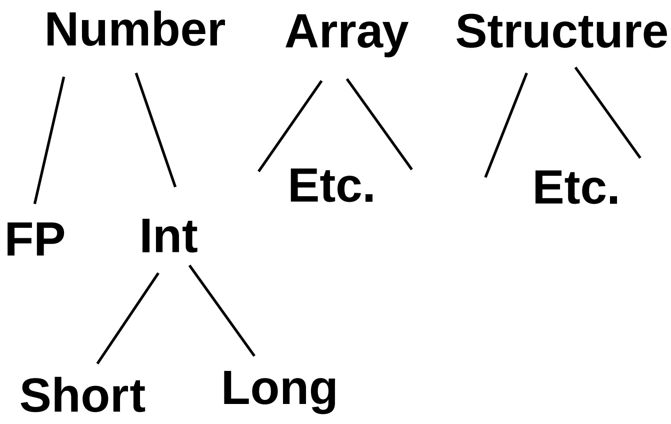
# Casting to `(void *)`

Neither C nor C++ define complete type hierarchies :-(

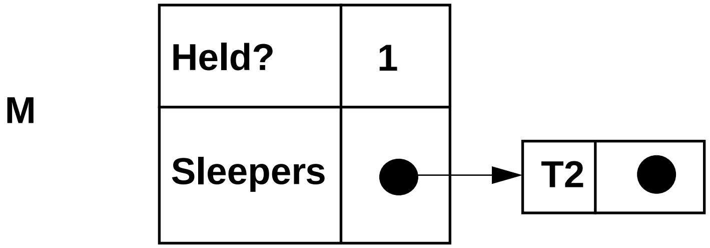
Complete Type Hierarchy (LISP)



Incomplete Hierarchy (C)



# Locks on Locks

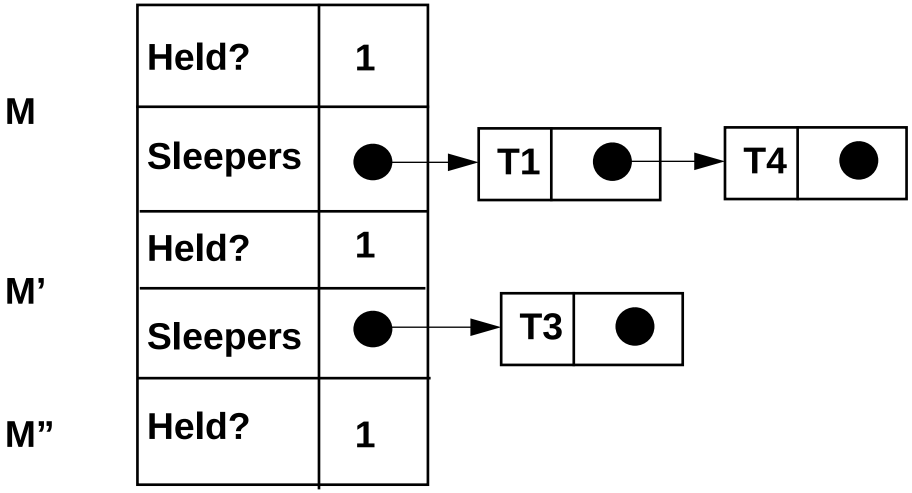


M

Held? 1

Sleepers ● → T2 ●

But the sleepers queue is shared data and must be protected!



M

Held? 1

Sleepers ● → T1 ● → T4 ●

M’

Held? 1

Sleepers ● → T3 ●

M”

Held? 1

If M” is held, block in the kernel as a last resort!

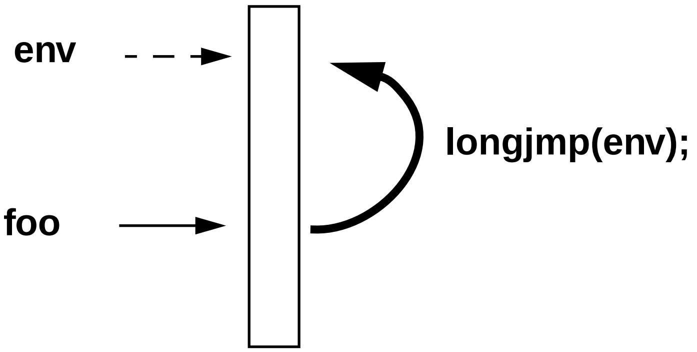
(Effective, not actual implementation)

# Using setjmp/longjmp

```
jmp_buf env;
```

```
baz(...)
{
 ...
 if (setjmp(&env) == 0)
   foo();
 else
   bar1();
 ...
}
```

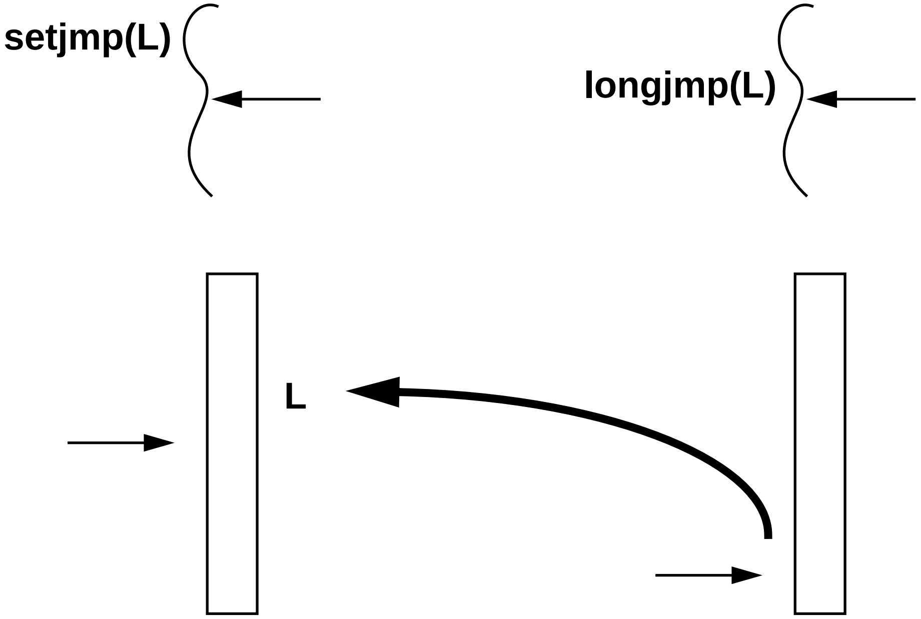
```
foo(...)
{...
 ...
 longjmp(env, 42);
 bar();
 ...
}
```



# Using setjmp/longjmp

**If you use these, you must be consistent in jumping to an active point in the same thread. (From a signal handler: `sigsetjmp()`, `siglongjmp()`.)**

**Bad Programmer!**



# Thread Stacks

**Solaris UI & POSIX:**

- **The main (first) thread always has a 2 meg stack allocated**
- **Default stack is 1 meg (on 32-bit Solaris), 2 meg (64-bit Solaris)**
- **You may specify stack size, but:**
    - **Rarely used. Be careful!**
    - **Window calls can get deep**
- **You may allocate the actual memory to be used yourself**
    - **Very unusual! Be very careful!**
    - **Mainly used for real-time applications which need to lock down physical memory**

**Java:**

- **All threads default to 500k (~10k minimal stack frames)**
- **% java -oss 1000000 (1MB, ~20k stack frames)**

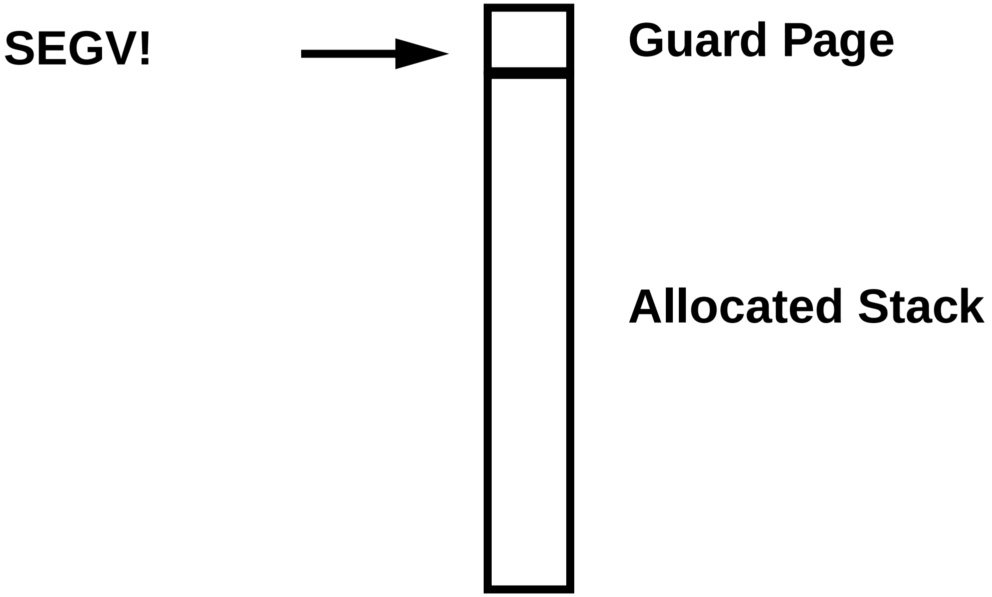
# Thread Stacks

**Win32:**

- **The main (first) thread always has a 1 meg stack allocated**
- **Default stack is also 1 meg**
- **You may specify stack size, but:**
  - **Rarely used. Be careful!**
  - **Window calls can get deep**
- **You may not allocate the stack memory yourself**

# Thread Stacks

**Solaris implementations of UI and POSIX allocate a *guard* page as the very last page of the stack to prevent stack overflow. Win32 does the same.**



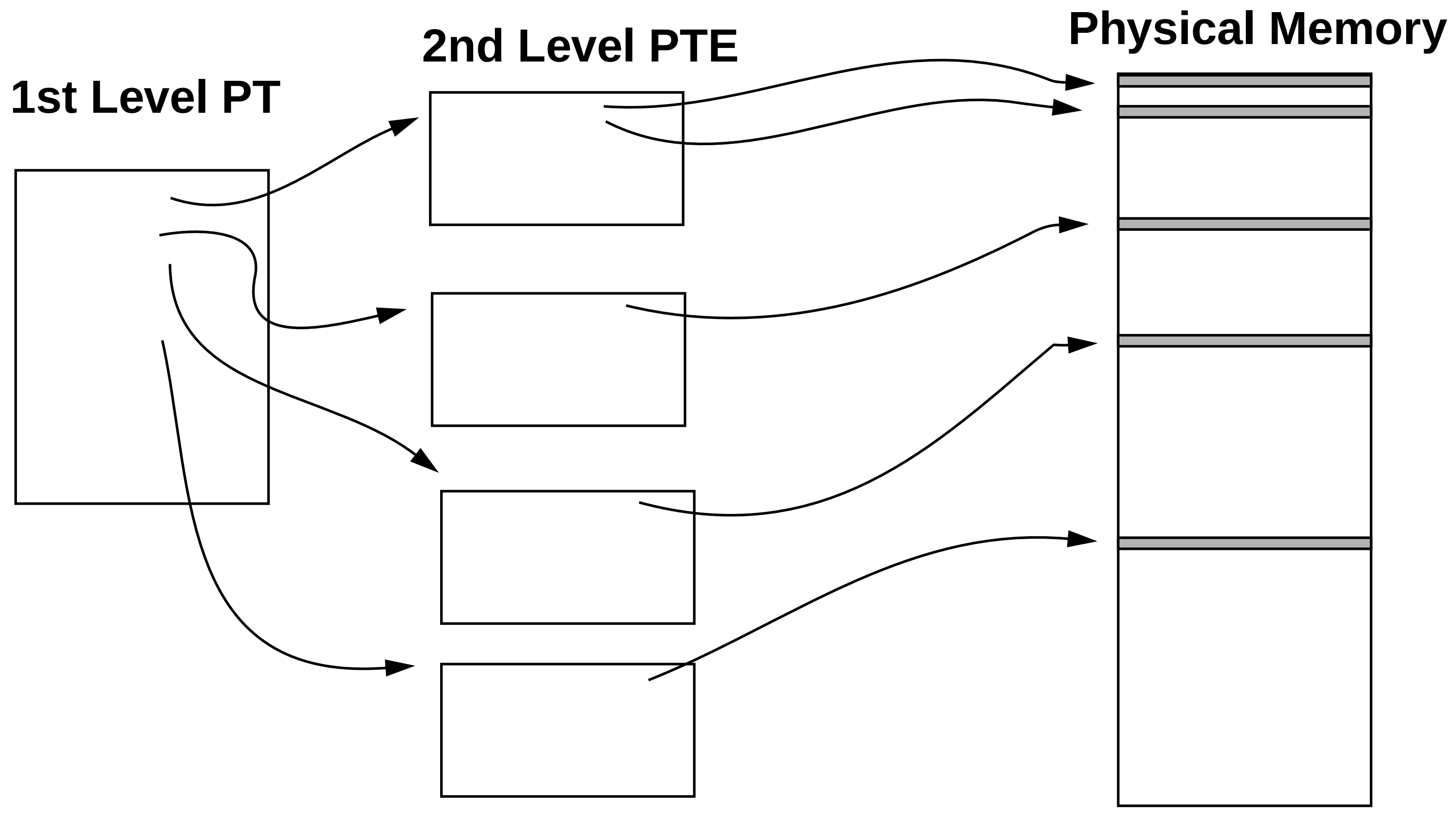
**See:** **`pthread_setguardsize()`** **(UNIX98)**

# Specifying Stack Sizes

In general it’s fine to use the default stack size. Should you create a very large number of threads (say 100 - 100,000), you will want to specify stack size:

- Minimum stack size is about 8k for all the libraries
- Verify the maximum stack depth your program can ever use
- All the libraries map in the stack as NORESERVE, meaning that no virtual memory is used until a page is touched (this is good)
- Multilevel page tables can have problems (they slow down a lot) if you allocate a large number of distant segments, even if those segments are NORESERVE (such as 500 1mb stacks)
- If you really do run completely out of virtual memory, your program can crash with a mysterious SIGBUS if it happens on a new stack reference (vs. the more familiar SIGSEGV)

# Multi-Level Page Tables



**Too many stacks far apart causes the second level page table entries to be paged themselves. (Bad thrashing on small memory machines.) You will only see this with > ~100 1meg stacks.**

# POSIX Optional Portions

- **In order to allow different implementations to do what they want, several portions of the POSIX definitions are optional.**
  - **A fully compliant Pthreads library does not need to implement those portions, and no one currently does implement all of the options.**
  - **Those portions include: “unbound” threads, FIFO and Round Robin real-time scheduling policies, real-time priority inheritance mutexes, shared memory mutexes, semaphores, CVs, stack size, stack allocation.**
- **See constants in `limits.h` and `sysconf()`**

# Other Scheduling Ideas

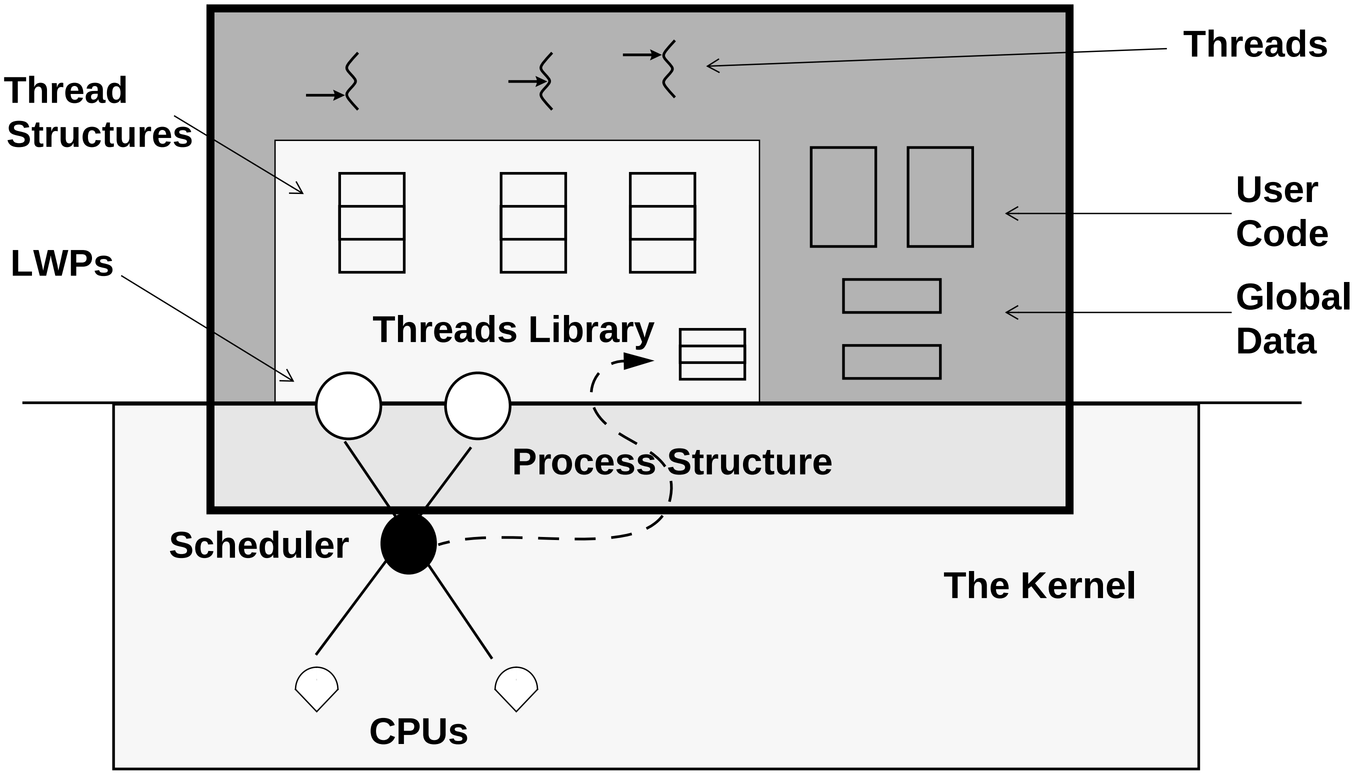
### Activations

- A specialized method of integrating user-level threads with kernel scheduling.
    - Allows CPU scheduling and thread context-switching to be controlled entirely in user space.
    - Used by SGI. Others?
    - Solaris 2.6+ has “Scheduler Controls” -- similar to “Activations”.

### Gang Scheduling

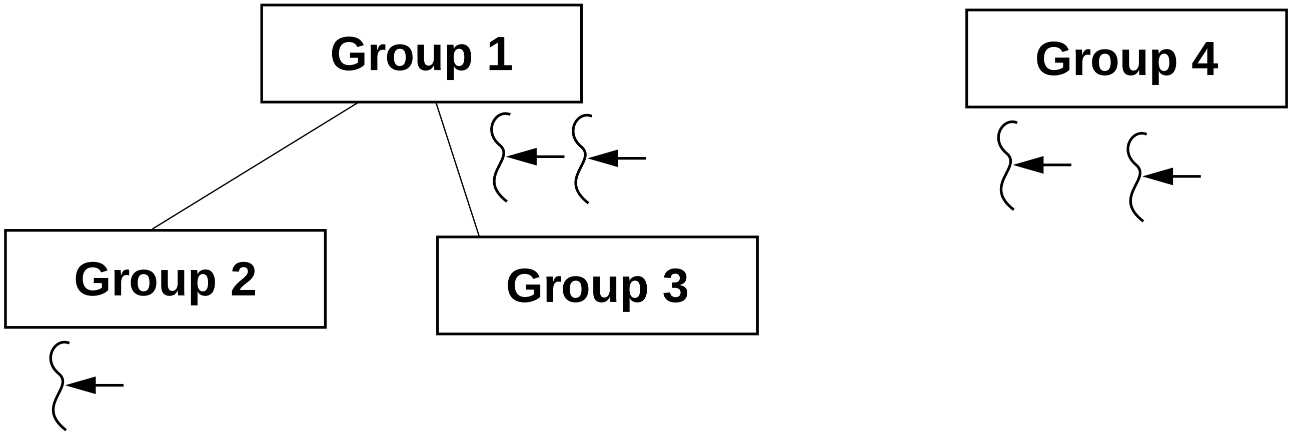
- Allows the user program to request CPUs be made available at the same time as a set.
    - Closely cooperating threads can be assured of running together.
    - Not implemented in any current systems?

# User Space Scheduler Input



- Upon context switching, the 2.6+ kernel updates a structure.
- Makes adaptive spin locks possible
- Allows user space input into the scheduling process.

# Java Thread Groups



A thread group is a set of threads and other thread groups. The primary idea behind these is that a single security manager will be responsible for all the threads in the group, allowing one set of threads one set of functions disallowed to another group.

In particular, threads running in one thread group in a browser will not have permission to touch threads running in other thread groups (e.g., the system thread group). A set of methods applied to a thread group may be applied to all its members. The primary methods in question are stop, suspend, and resume. All of which are deprecated.

# Java Thread Groups

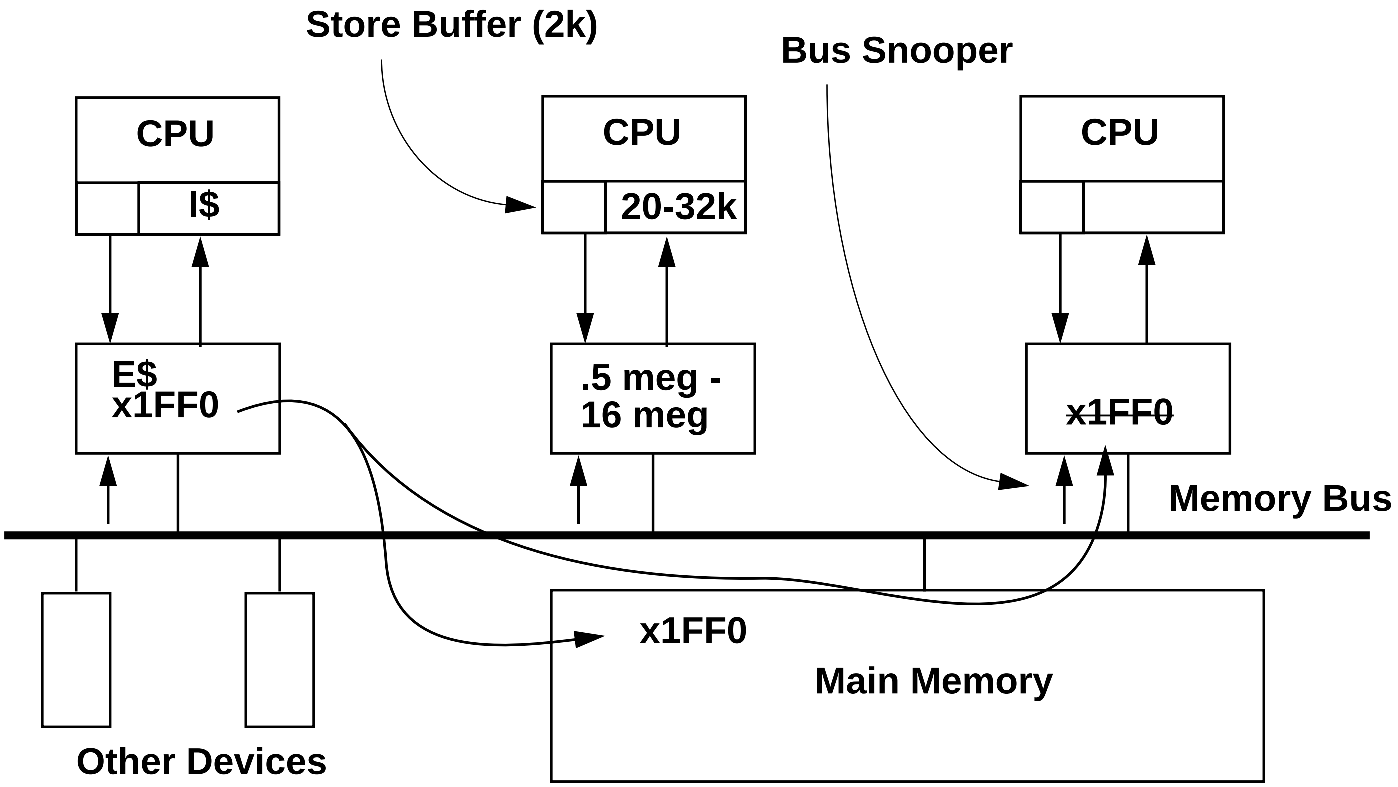
```
ThreadGroup(ThreadGroup parent, String name)
void tg.suspend()
void tg.interrupt()
void tg.resume()
void tg.stop()
void tg.destroy()
ThreadGroup tg.getParent()
boolean tg.parentOf(ThreadGroup g)
int tg.enumerate(Thread list[])
int tg.activeCount()
void tg.setMaxPriority(int p)
int tg.getMaxPriority()
void tg.setDaemon()

Thread(ThreadGroup g, String name)
ThreadGroup thread.getThreadGroup()
void securityManager.checkAccess(thread t)
void securityManager.checkAccess(ThreadGroup g)
```

# *Bus Architectures*

# Types of MP Machines

- **SMP: Shared Memory Symmetric Multiprocessors**
  - **Most common, from 2-CPU SS20 ($20,000) to 64-CPU CS6400 ($1,000,000)**
- **D-SMP: Distributed Shared Memory Multiprocessors**
  - **32 - 2048 CPUs (Cray T3D ~$1,000,000)**
- **MPP: Massively Parallel Multiprocessors**
  - **64 - 1,000,000 CPUs, > $5,000,000 (CM)**
  - ***Hennessy and Patterson should move MPPs to Chapter 11.*** **- Jim Gray**

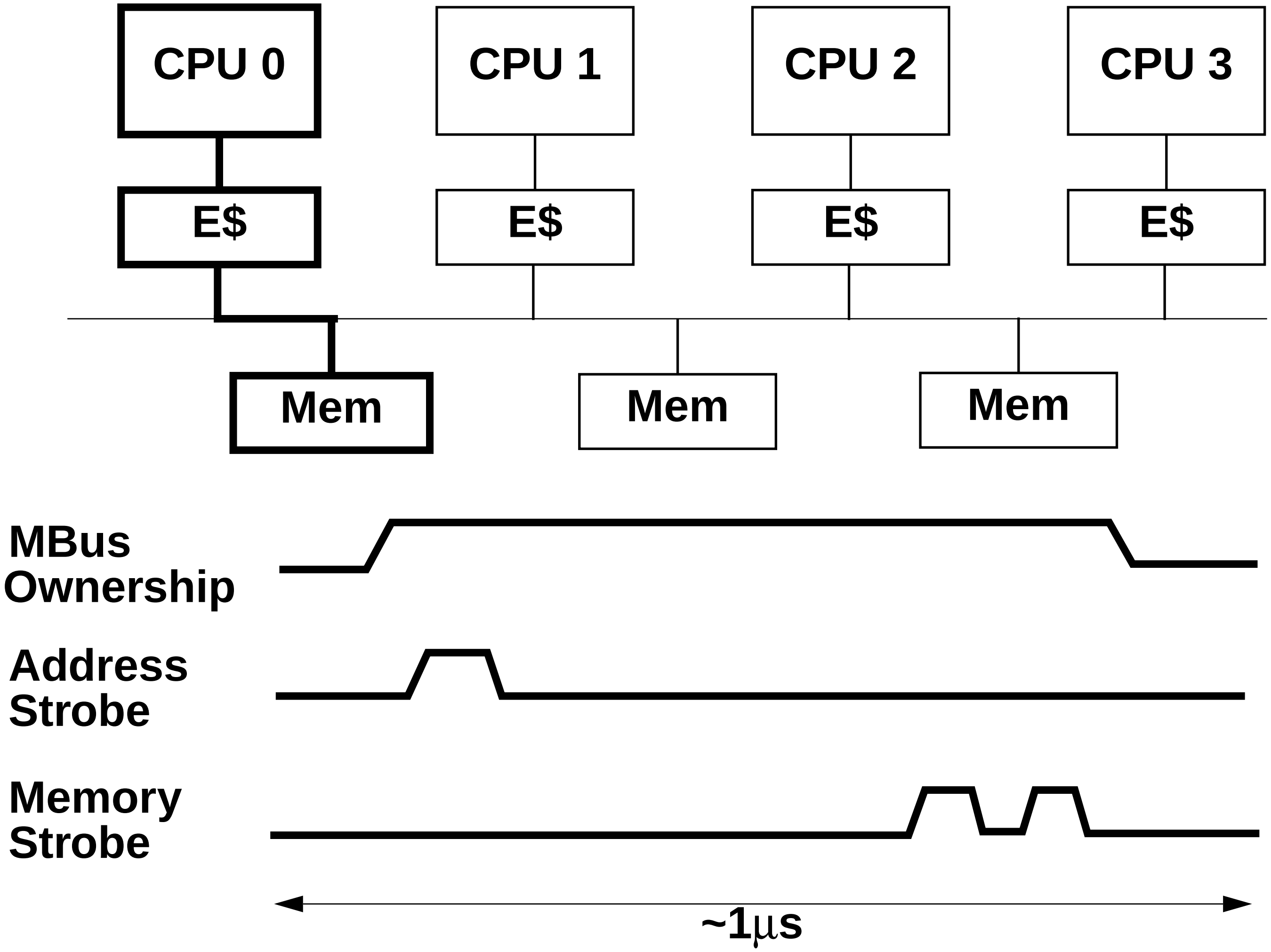
# Shared Memory SMP Bus Designs



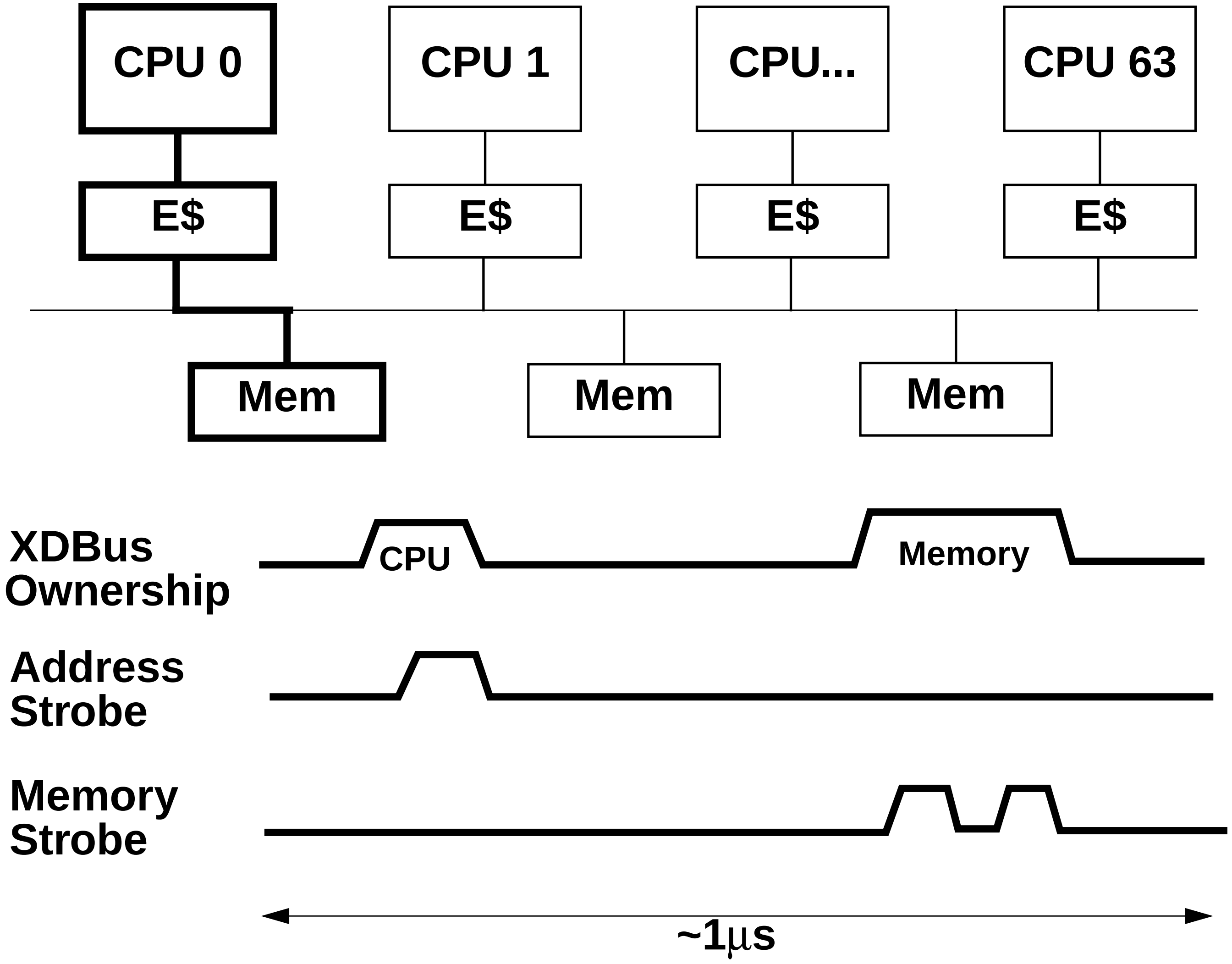
Must flush store buffers to keep memory coherent between CPUs after mutex unlocks.

Bus snooper invalidates old cache entries.

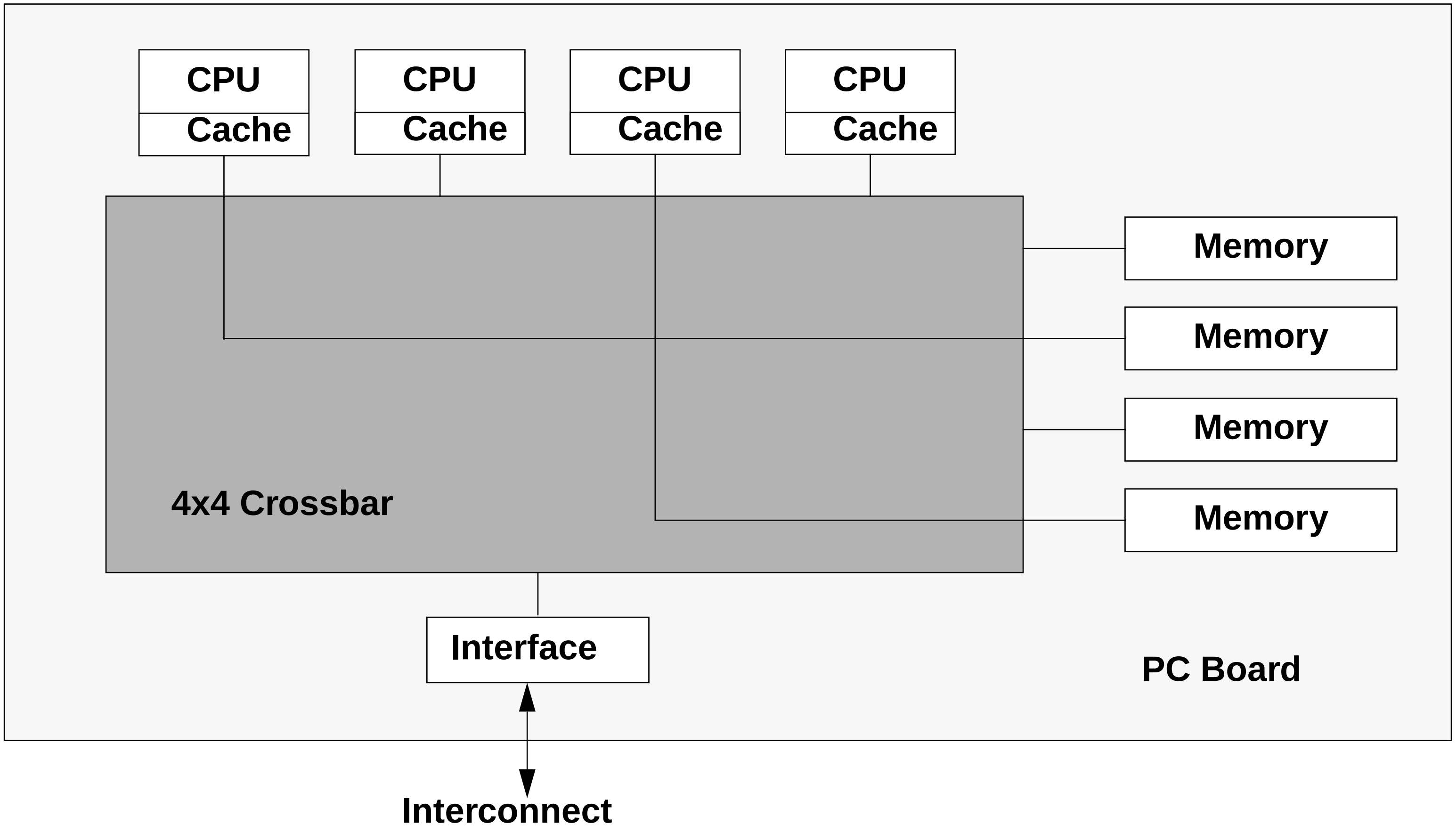
# Direct Switched Bus: MBUS



# Packet Switched Bus: XDBUS

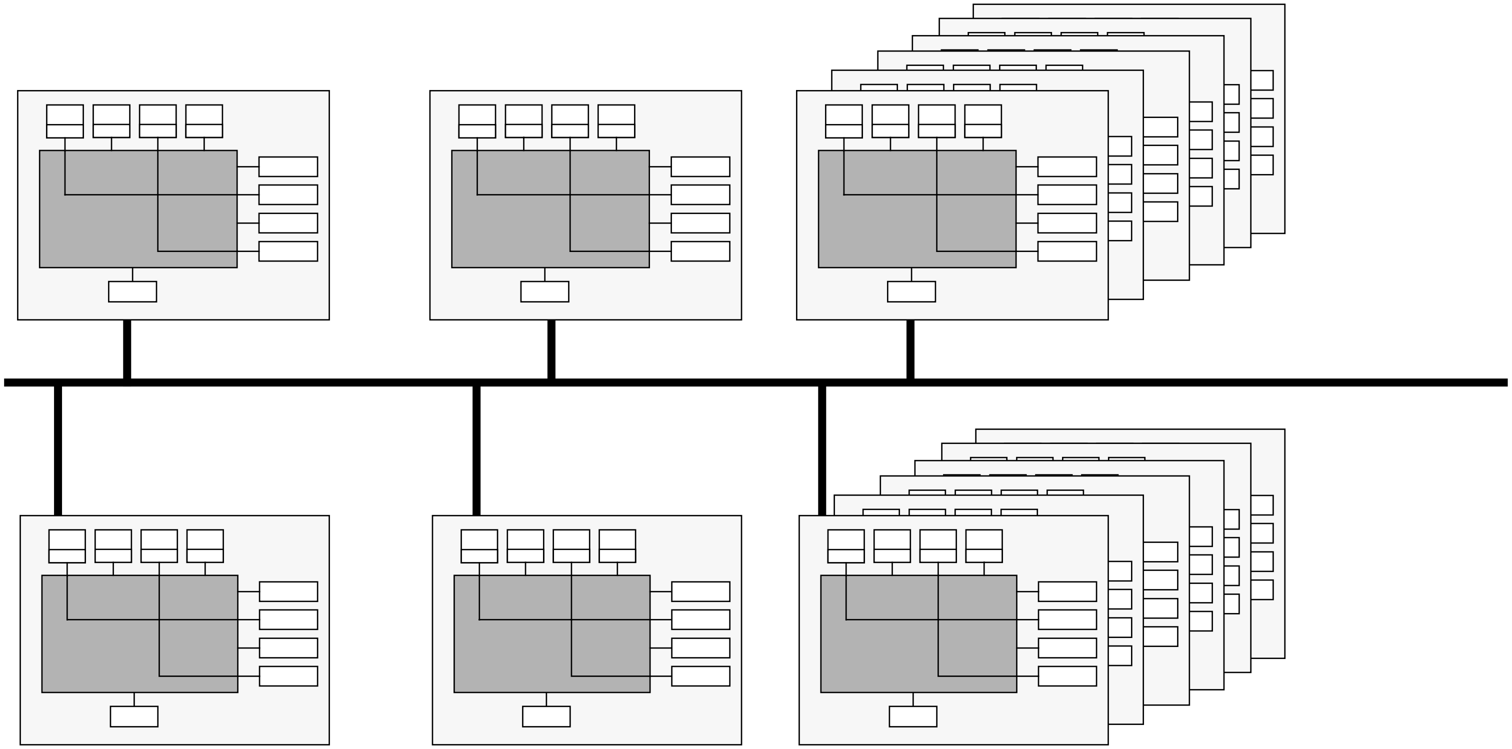


# Cross-Bar Switches



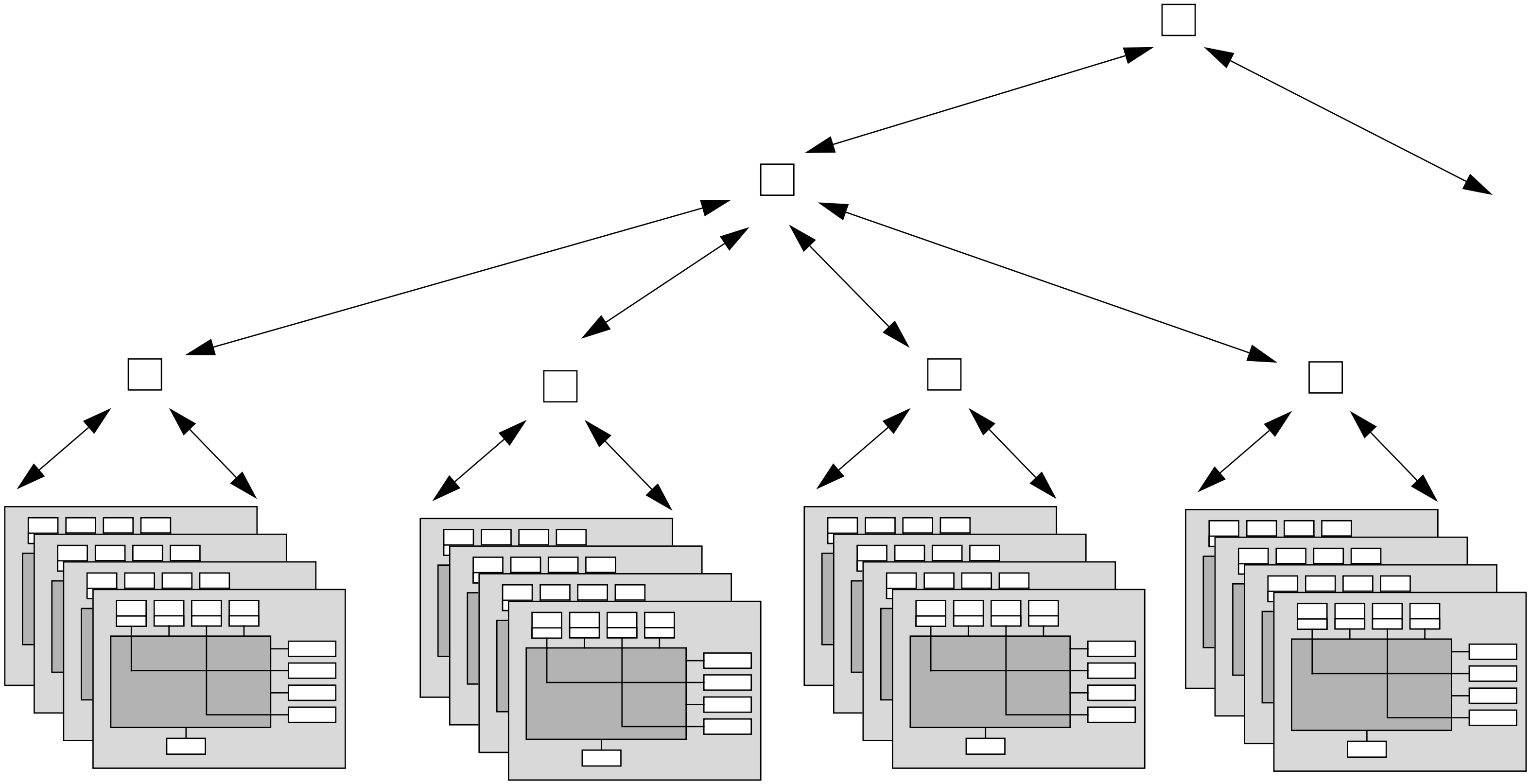
**Cross-bars are very fast, allow parallel reads, and are very expensive. Writes must be seen by all CPUs (for invalidations).**

# Sun Ultra Enterprise



**This design scales up to 64 CPUs. On-board access is fast, off-board access is slower. All writes propagate across all CPUs (for invalidations).**

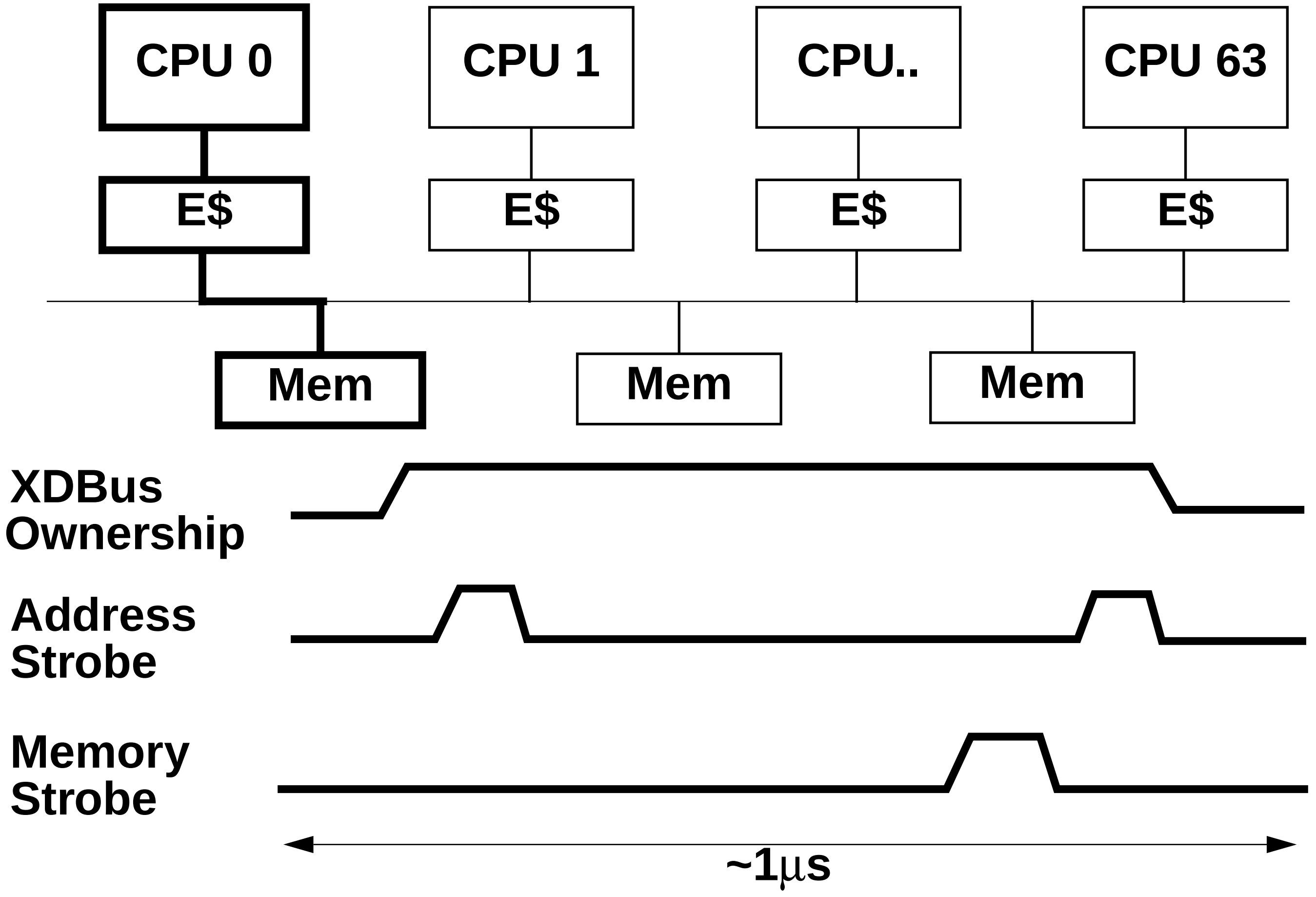
# SGI Origin



**This design will scale upwards to 4096 CPUs! Each jump across the interconnect costs ~150ns. An on-board directory keeps track of all CPUs caching a local word, and invalidates propagate only to those CPUs.**

# Packet Switched Bus: XDBUS

**`LDSTUB` is expensive!**



# Spin Locks Done Better (Solaris 2.5)

```
#include <thread.h>
#include <errno.h>
#include <synch.h>

int SPIN_COUNT;
mutex_t m;


void spin_lock(mutex_t *m)
{int i;

 for (i = 0; i < SPIN_COUNT; i++)
   {if (m->lock.owner64 == 0)            /* Check w/o ldstub */
         if (mutex_trylock(m) != EBUSY)
               return;                   /* Got it! */
    /* Didn’t get it, continue the loop */
   }

 mutex_lock(m);                          /* Give up and block */
}
```

**It is always best to use the vendor-provided spin lock (if any).**

# The “Thundering Herds” Problem

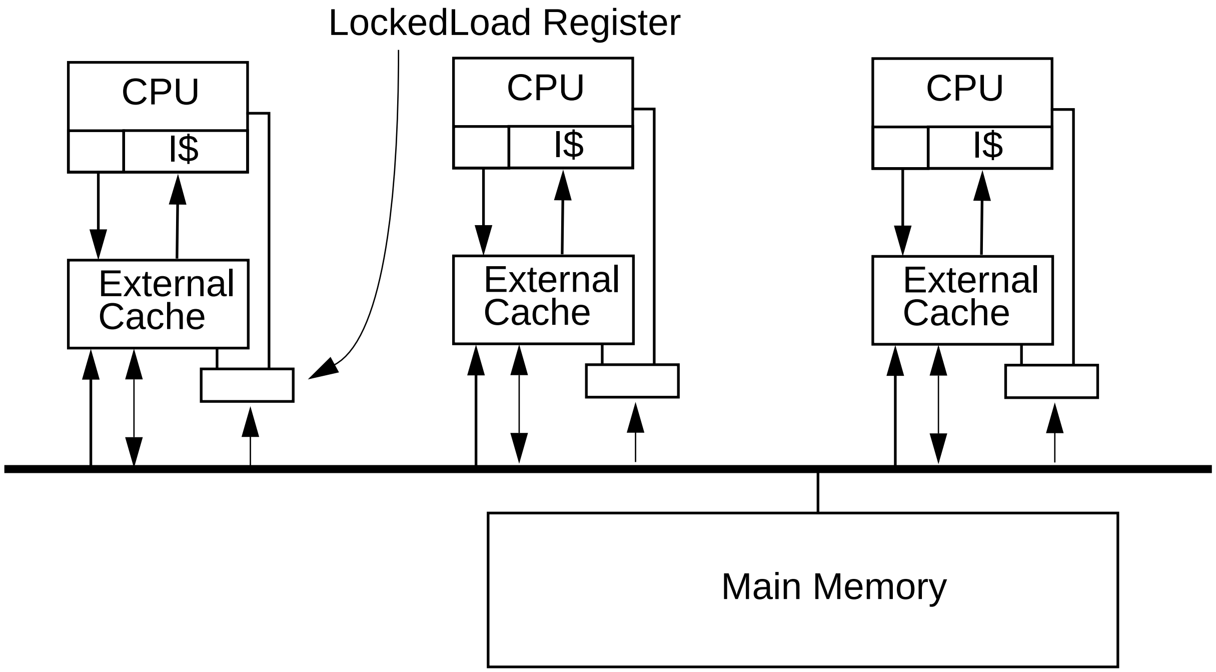
**With many CPUs all spinning on the same lock, the memory bus will get inundated with $-miss requests, followed by `ldstubs`. The $-misses must happen, but you can spread them out AND avoid the subsequent `ldstubs` with proper timing and luck. With high-contention spin locks we can slow down the peak speed (i.e., without contention), but improve the average speed under contention.**

```
for (j = 0; j < SPIN_COUNT; j++)
  {for (i = 0; i < DELAY_COUNT; i++) ;         /* Do Nothing! */
   if (m->held == HELD) continue;
   if (trylock(m) == EBUSY) continue;
   return;
  }
lock(m);
```

***Very, Very*** **unusual problem!**

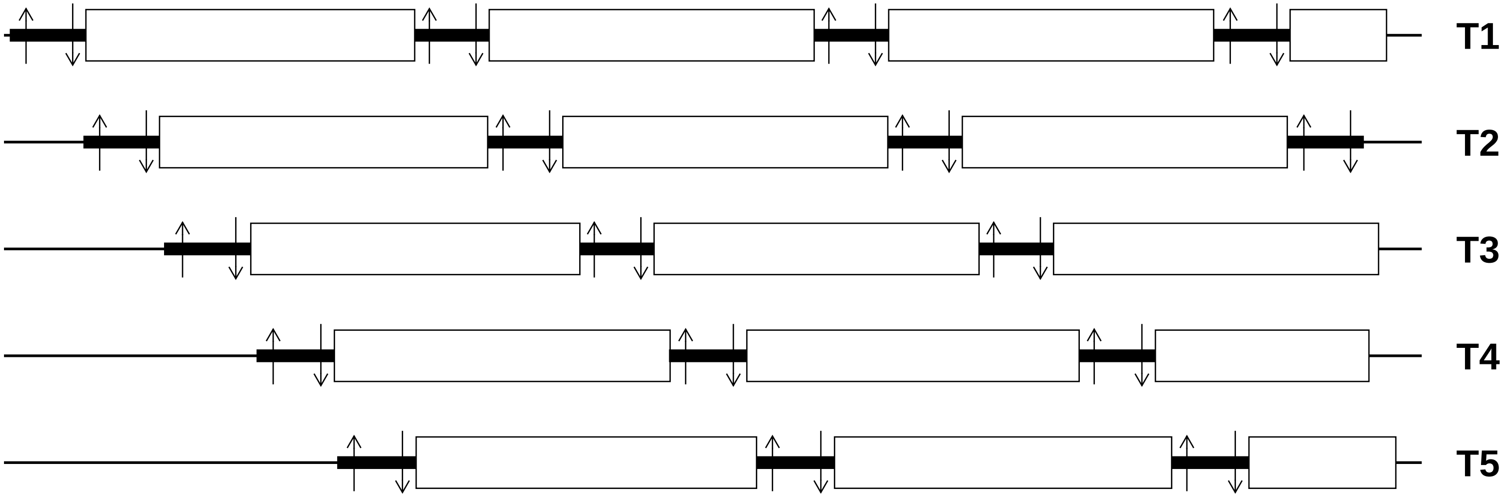
**Don’t bother with this unless you do *Very* detailed analysis.**

# Other Synchronization Instructions



**On the DEC Alpha and MIPS, there is a `LockedLoad` instruction which loads a word from main memory and keeps track of that word. When you subsequently issue a `StoreConditional` to that word, it will only succeed if no one else has written to it. This saves the bus locking which SPARC has to do. These instructions, and the SPARC v9 `CAS` (Compare and Swap if Equal) allow you do increment/decrement variables without an explicit mutex.**

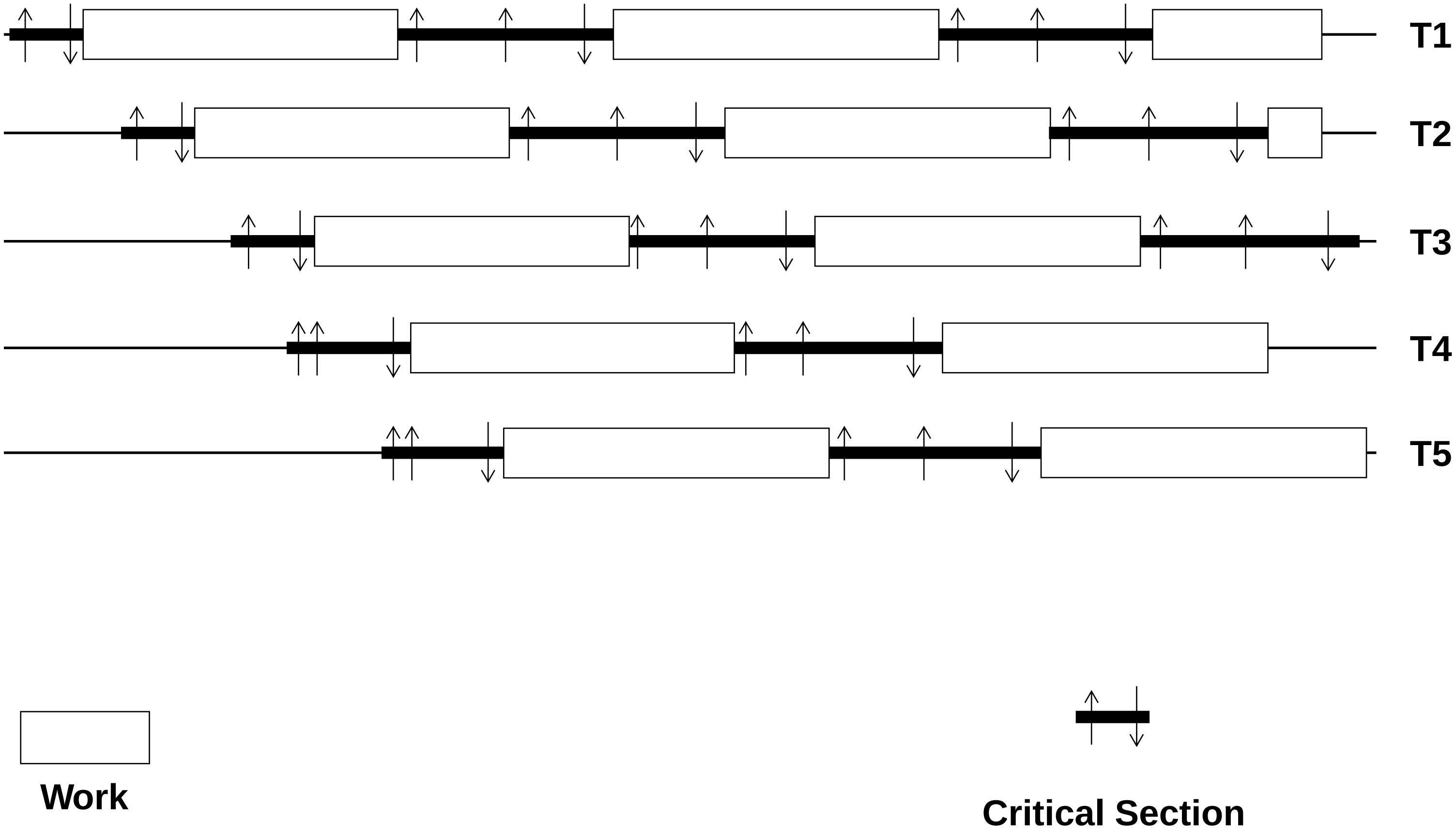
# Critical Section Contention



In Perfect Lockstep

Work (20µs)
Critical Section (5µs)

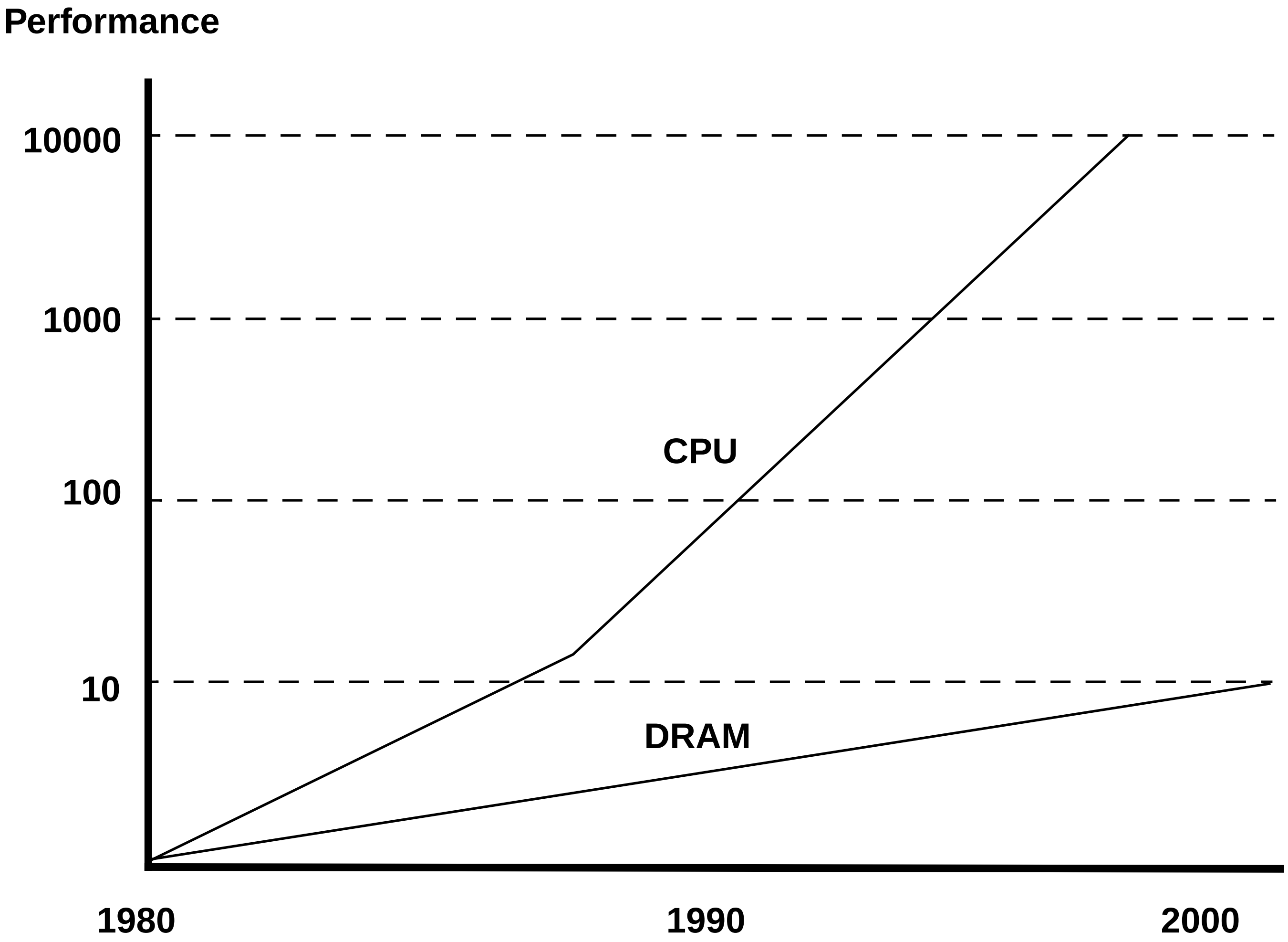
# Critical Section Contention



**Degradation due to the extra CPUs is normally gradual.**

# *Memory Systems*

# Memory Got Slower

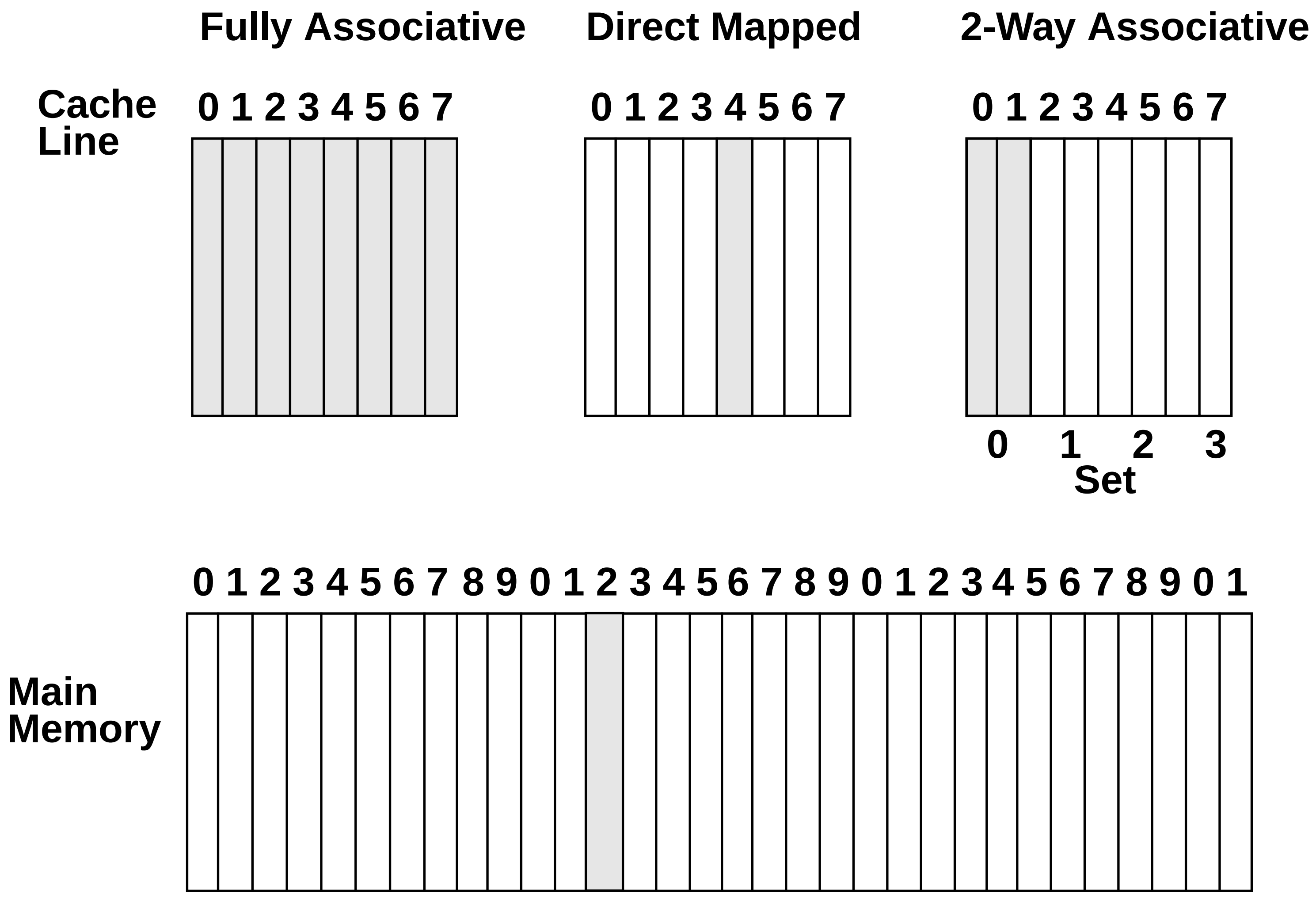


(The graphs in this section are from Hennessy & Patterson)

# Cache Memory

- Very fast, very expensive memory which resides very close to the CPU and provides either single cycle access (internal cache) or < 10 cycle access (external cache).
- Cache miss reasons:
    - Compulsory - first access
    - Capacity - not enough room in the cache
    - Conflict - all placement locations filled

# Cache Line Placement



Fully Associative

Direct Mapped

2-Way Associative

Cache Line

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

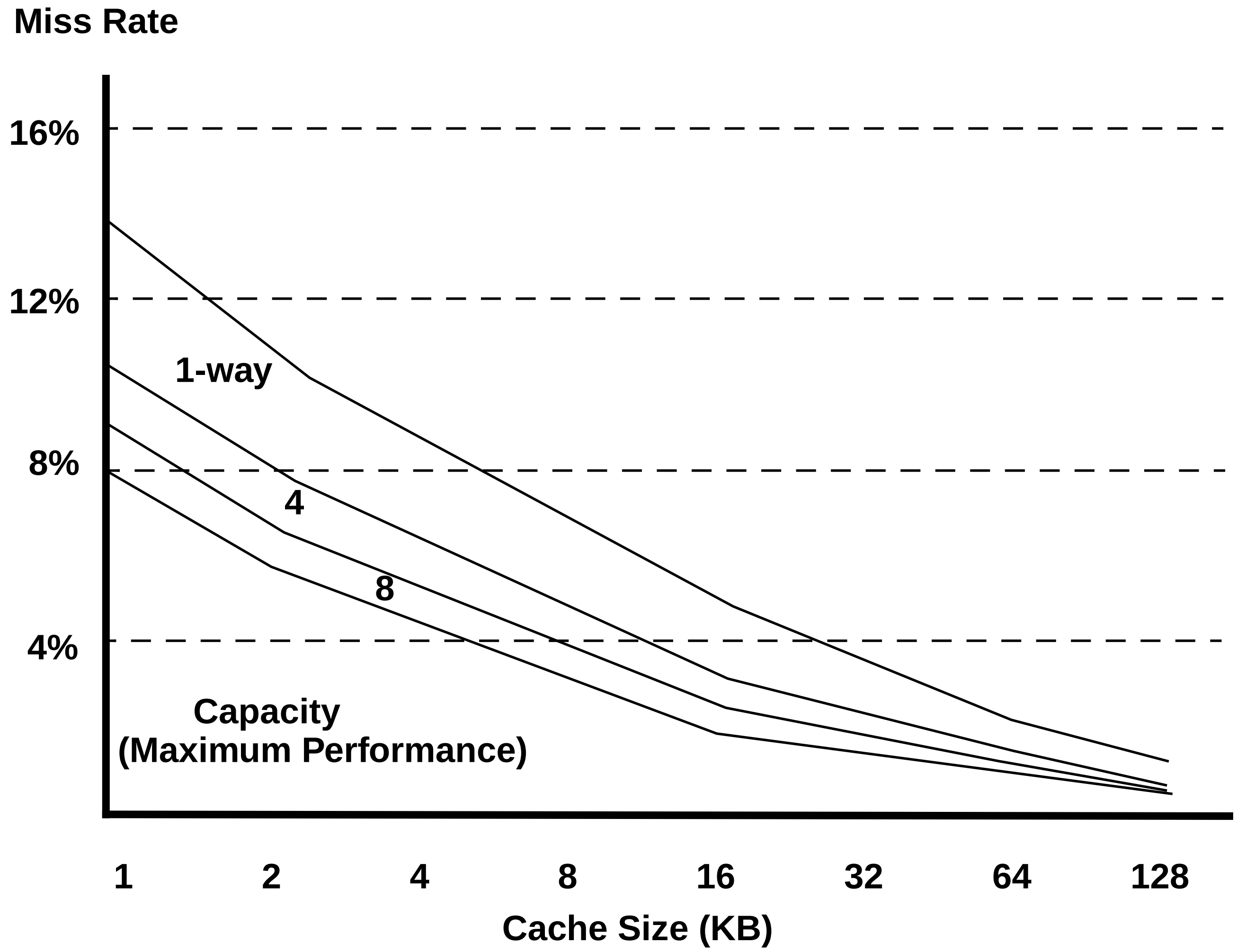
0 1 2 3

Set

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Main Memory

# SPEC92 Cache Miss Rates



# Out of Order Execution

```
ld add0, L0       <-- Cache miss here...
ld add1, L1
ld add2, L2
add L1, L2
st L2, add3
ld add4, L3
add L3, L2
mv L2, L4
add L4, L4
add L4, L0        <-- But not used until here!
```

- This is quite expensive in silicon, but allows for significant speedup in many programs.
- Compilers do not optimize for cache loading/placement.

# Volatile Data

**“But wait!” somebody says. “How do I know that a variable is even going to get written out to main memory? What if my data is kept in a register? Don’t I have to declare it volatile?”**

**No. Only variables local to a function are allowed to be kept in a register across a function call (and `pthread_mutex_unlock()` is a function call!)**

**Of course, if you’re trying to cheat and not use mutexes...**

# Synchronize Your Data!

**All of the preceding memory consistency issues become mute when you use synchronization properly. Unlocking a mutex always includes a call to a store barrier, ensuring that all data in the store buffer is flushed to main memory before the next instruction can complete.**

# Typical Memory Designs

| | Internal Cache | External Cache | Virtual Memory |
|---|---|---|---|
| Block Size | 4 - 64 bytes (64) | 32 - 256 bytes (32) | 4k - 16k (8k) |
| Hit time | 1 - 2 CPU cycles (1) | 5 - 15 CPU cycles (5) | 10 - 100 CPU cycles (~100) |
| Miss penalty | 5 - 66 CPU cycles (5) | 30 - 200 CPU cycles (~100) | 700,000 - 6,000,000 |
| Miss Rate | 15 - 30% | 0.5 - 20% | 0.00001 - 0.001% |
| Size | 1 - 128k (36k) | 256k - 16m (2m) | 16m - 8g (2g) |
| Block Placement | Direct Mapped (DM) | Direct Mapped or Set Associative (5-way) | Fully Associative |
| Block Replacement | N/A | Random | LRU |
| Write Strategy | Write-through or Write-back | Write-back | Write-back |

**(Actual UE Data)**

# Memory Buses

| | HP Summit | SGI Challenge | Sun XDBus | Sun UPA |
|---|---|---|---|---|
| Data Width | 128 | 256 | 144 (128) | 256 |
| Clock Rate | 60 MHz | 48 MHz | 66 MHz | 83 MHz |
| Bus Masters | Multiple | Multiple | Multiple | Multiple |
| Peak Bandwidth | 960 MB/sec | 1200 MB/sec | 1056 MB/sec | 1600 MB/sec |
| Type | Packet-switched | Packet-switched | Packet-switched | Cross-bar |

# Latency

Latency is the total amount of real (wall clock) time a task takes.

Throughput is the total number of tasks that can be done per second.

CPU time is how long the CPU was actually working on your task.

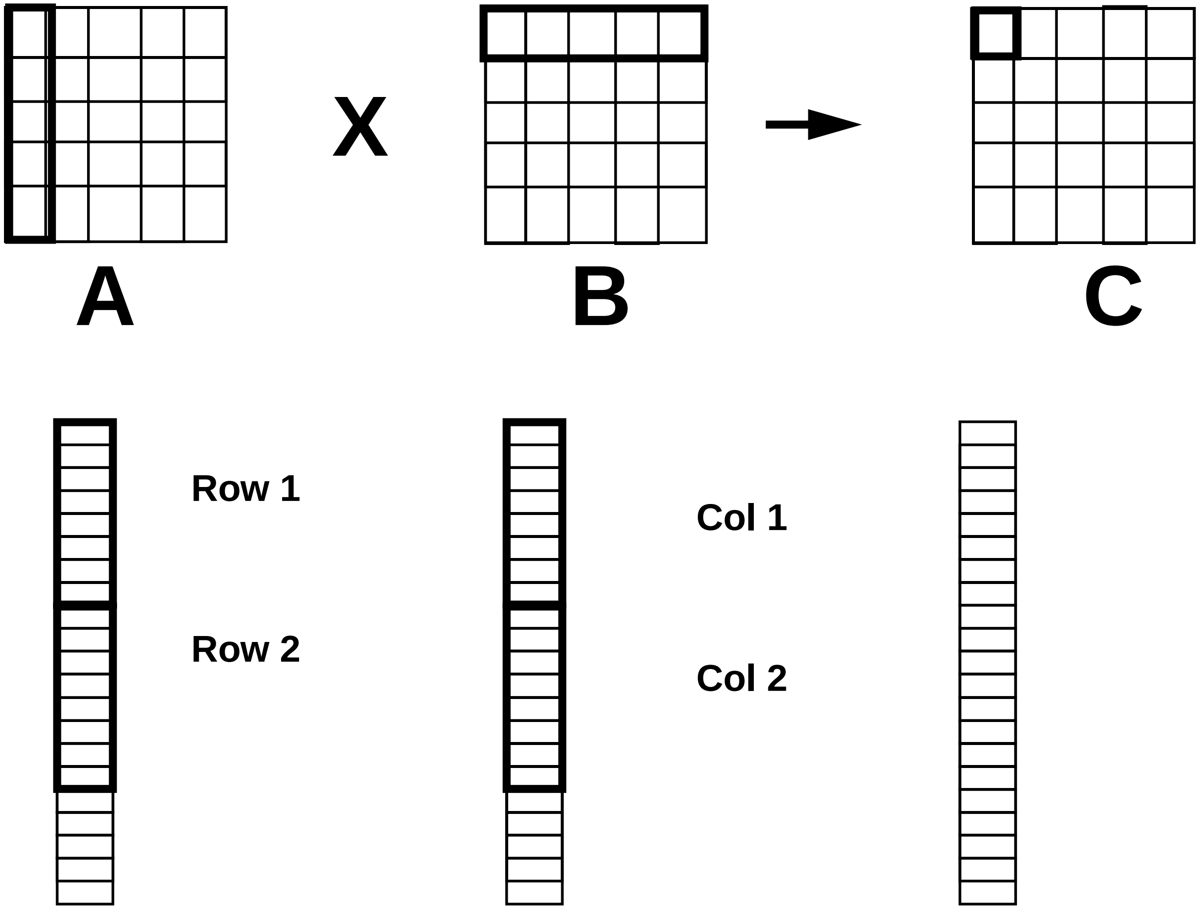
CPU time <= latency, often CPU time << latency.

Disk seek time, network delays, etc. often make up 99% of latency.

Systems are generally optimized to produce maximum throughput, even at the cost of greater latency.

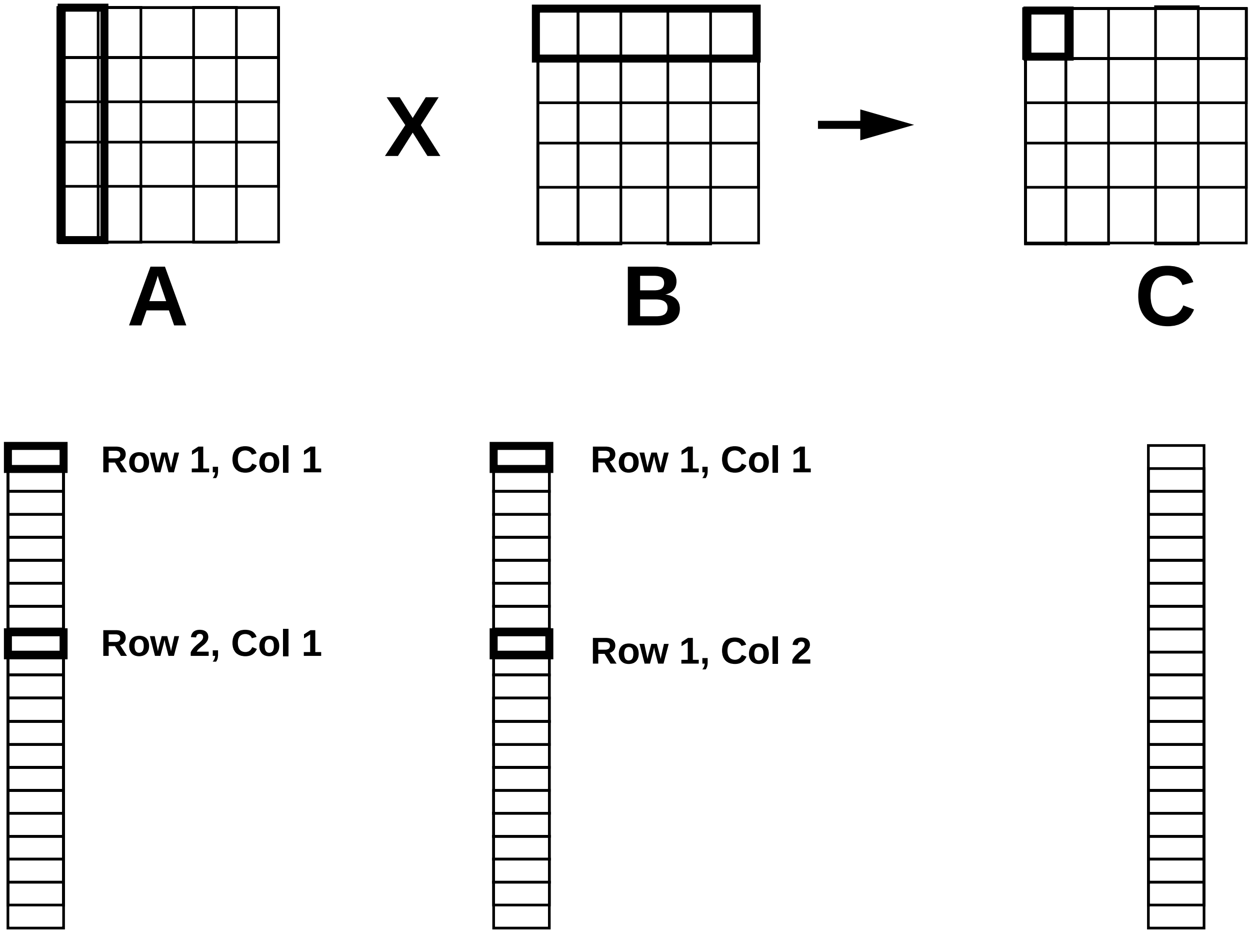
Cache miss latency is not recorded, just lumped into CPU time. :-(

# High Latency Cache Misses Cost!



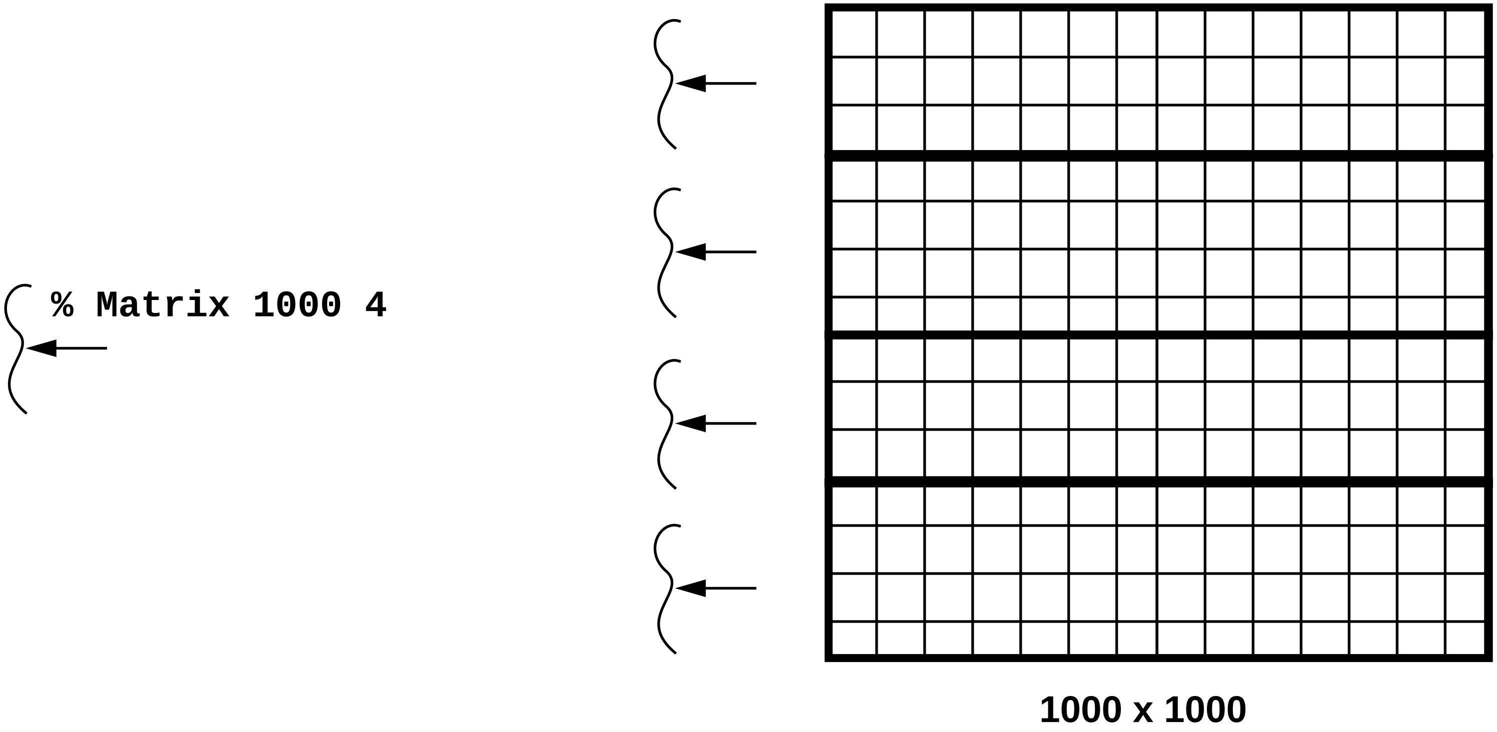
**Memory Layout, Best Case!**

# High Latency Cache Misses Cost!



**Memory Layout, Worst Case!**

# Naive Matrix Multiply



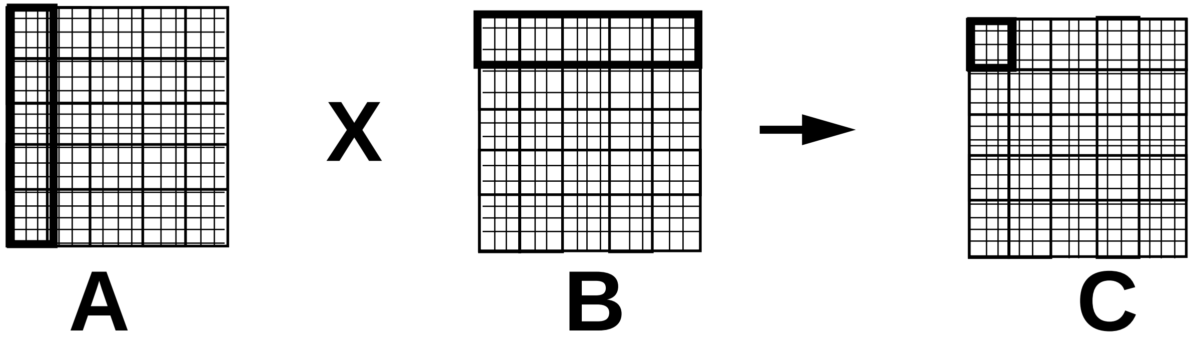
```
% Matrix 1000 4
```

**Total of 2 gflops, @ 10ns/flop -> 20 seconds**

**12mb compulsory $ loads -> 360 ms**

**4gb capacity $ loads -> 120 seconds (!)**

# $-Blocking Matrix Multiply



1000 x 1000

**Total of 2 gflops, @ 10ns/flop -> 20 seconds**

**12mb compulsory $ loads -> 360 ms**

**28mb capacity $ loads -> 840ms (assuming 1-mb $)**

# However...

# The TLB has only 64 entries!

Solaris 2 allows only 8k VM pages, so the TLB can map only 512k. (The UltraSPARC implementation allows for 8k, 64k, 512k, and 4m pages, but only 8k is used.)

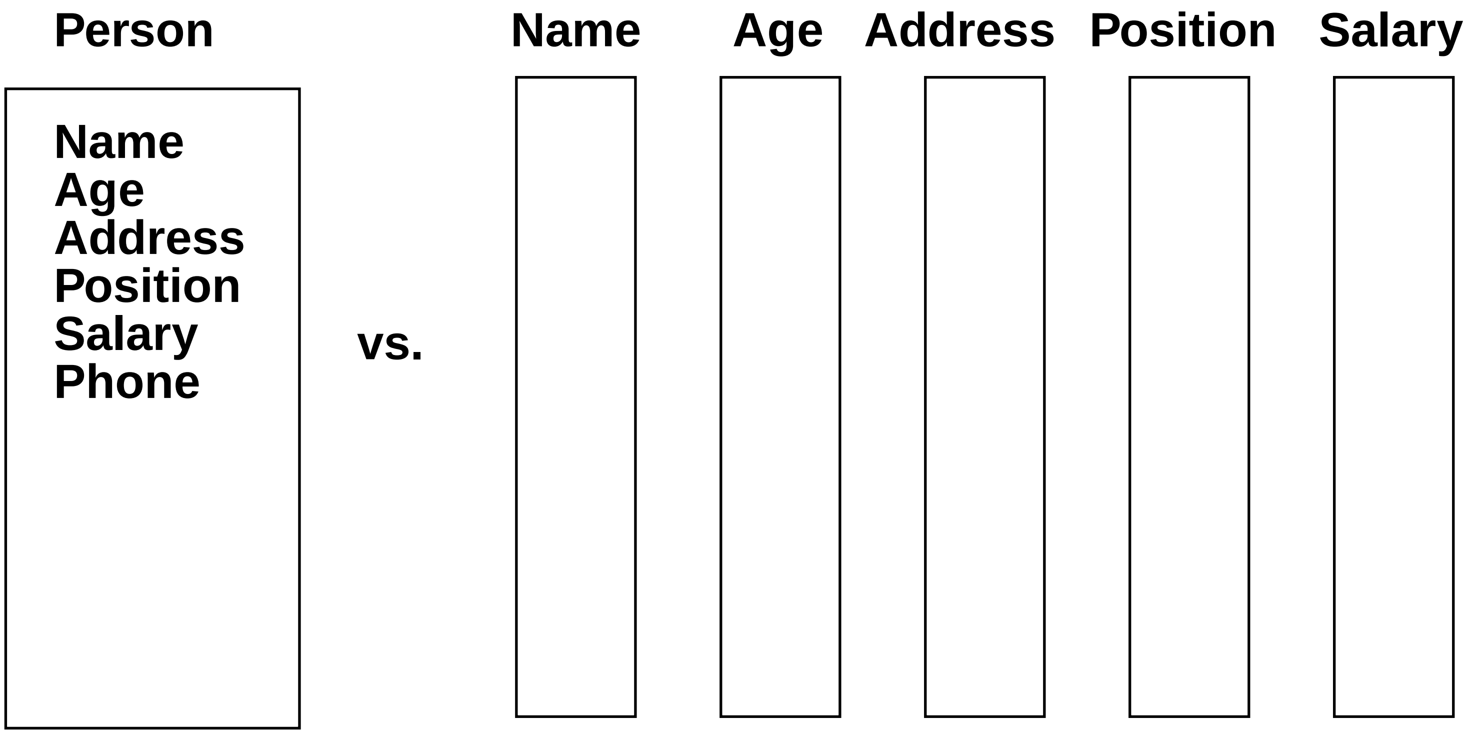
SPARC E$ maps physical addresses, so at least 1/2 of all $ entries in a 1 meg E$ require a TLB reload.

Optimal speed is obtained on a SuperSPARC or UltraSPARC system by blocking to 512k, and using the rest of the $ as a secondary store.

Oh, and you also have to account for E$ being write-back, hence optimize the target array first.

***Don’t do this at home!***

# Redefining Data to Fit Cache



Person

Name
Age
Address
Position
Salary
Phone

vs.

Name Age Address Position Salary

Fewer $ misses if using one field (e.g., searching for a name, averaging salaries).

# False Sharing

What if two different threads use two different (unshared) variables which are in the same cache line?

You wouldn’t need to use a mutex (they are not shared, after all), but your performance would be in the pits. Consider:

`int a[8];`

T1 (on CPU 1) writes `a[1]` a lot, and T2 (on CPU 2) writes `a[2]` a lot.

T1 writes `a[1]`, invalidating the entire cache line for CPU 2. T2 reads `a[2]`, taking a cache miss, and then writes `a[2]`. Now CPU 1’s cache line is invalidated. When T1 next reads `a[1]`...

You get the idea.

***Don’t do that!***

# *More Performance*

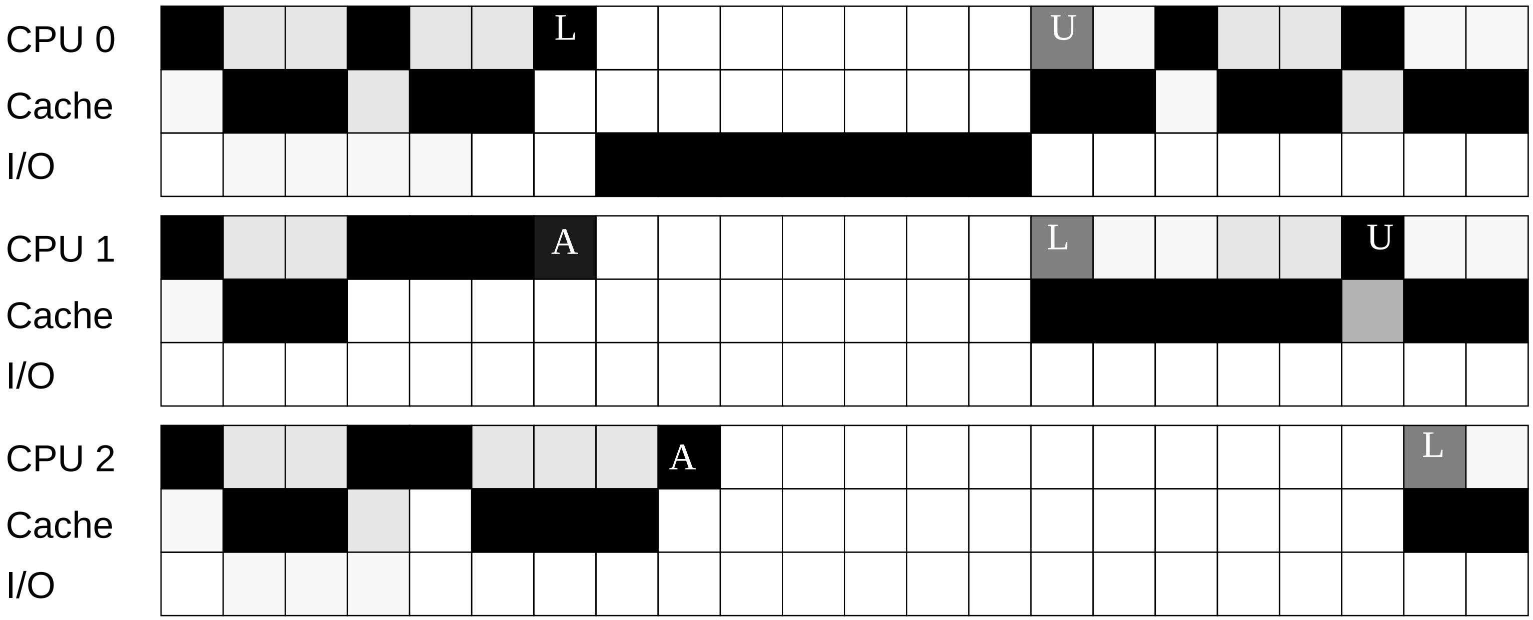
# Programming for Performance

- is the exception, not the rule
- Most programs don’t care if an operation takes 1µs or 100ms.

*Keep your eyes on the prize!*

- Only optimize what you need to optimize.

# Bottlenecks



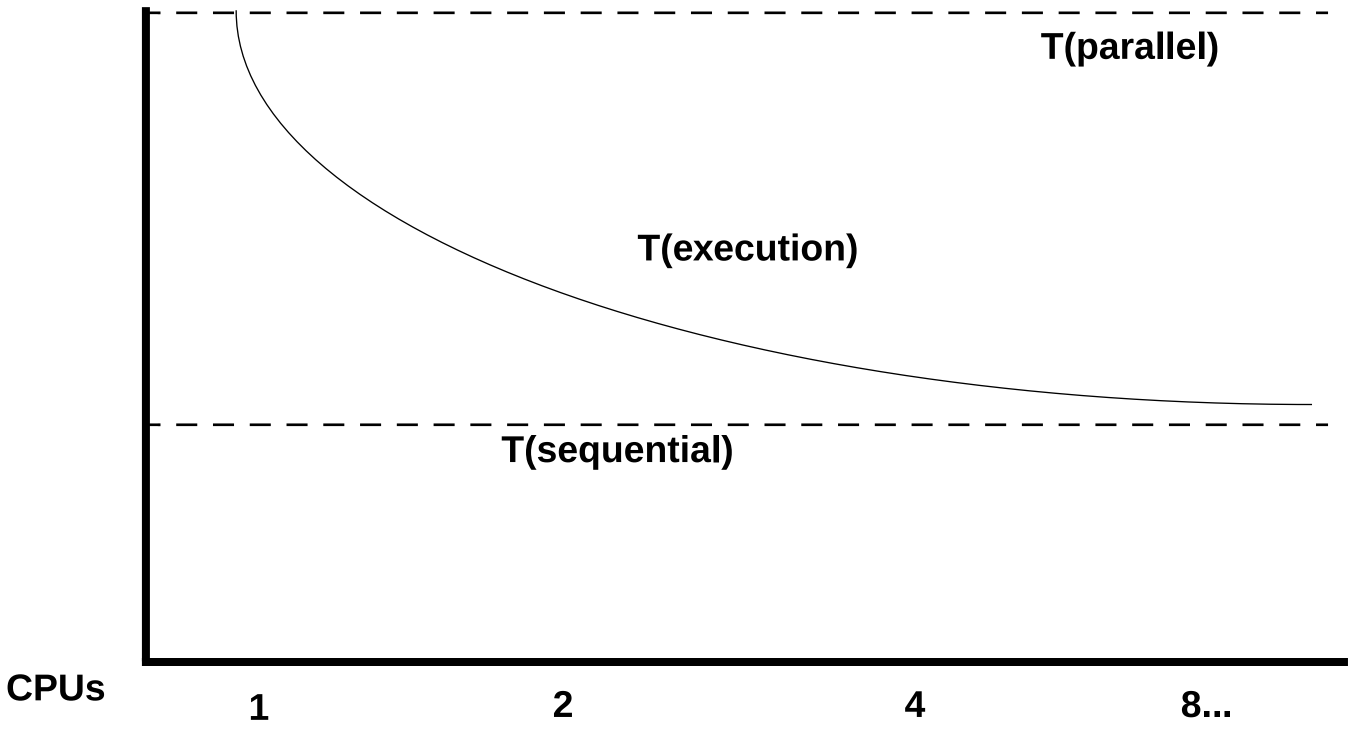
■ 100% Busy ▩ 75% ▨ 50% ░ 25% □ 0%

L = Mutex Locked Here U = Mutex Unlocked Here A = Mutex Lock Failed

**The “bottleneck” is that subsystem which is doing the work! Maybe you can speed it up, maybe you can’t.**

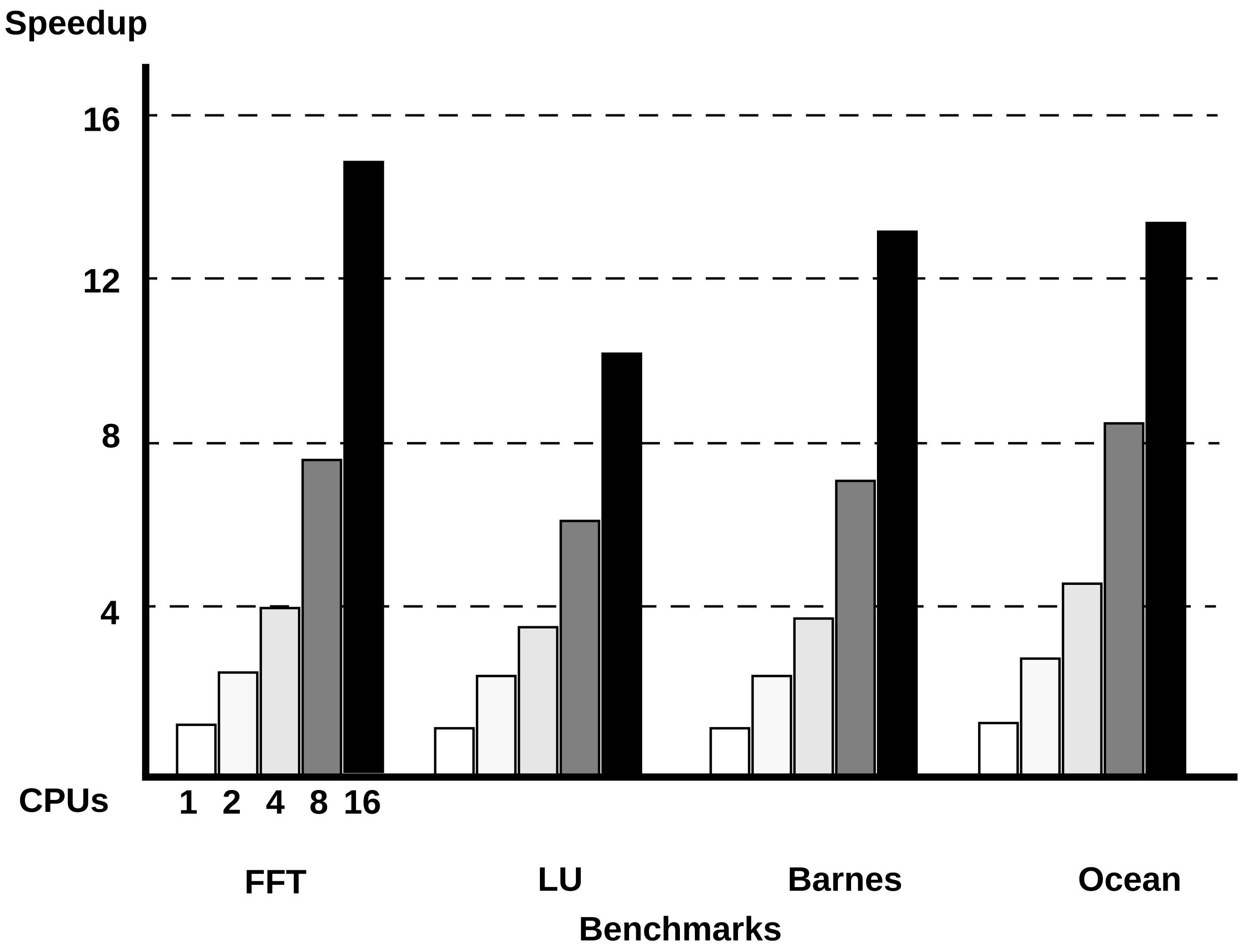
# Amdahl’s Law

**T(execution) = T(sequential) + T(parallel)/N_Processors**

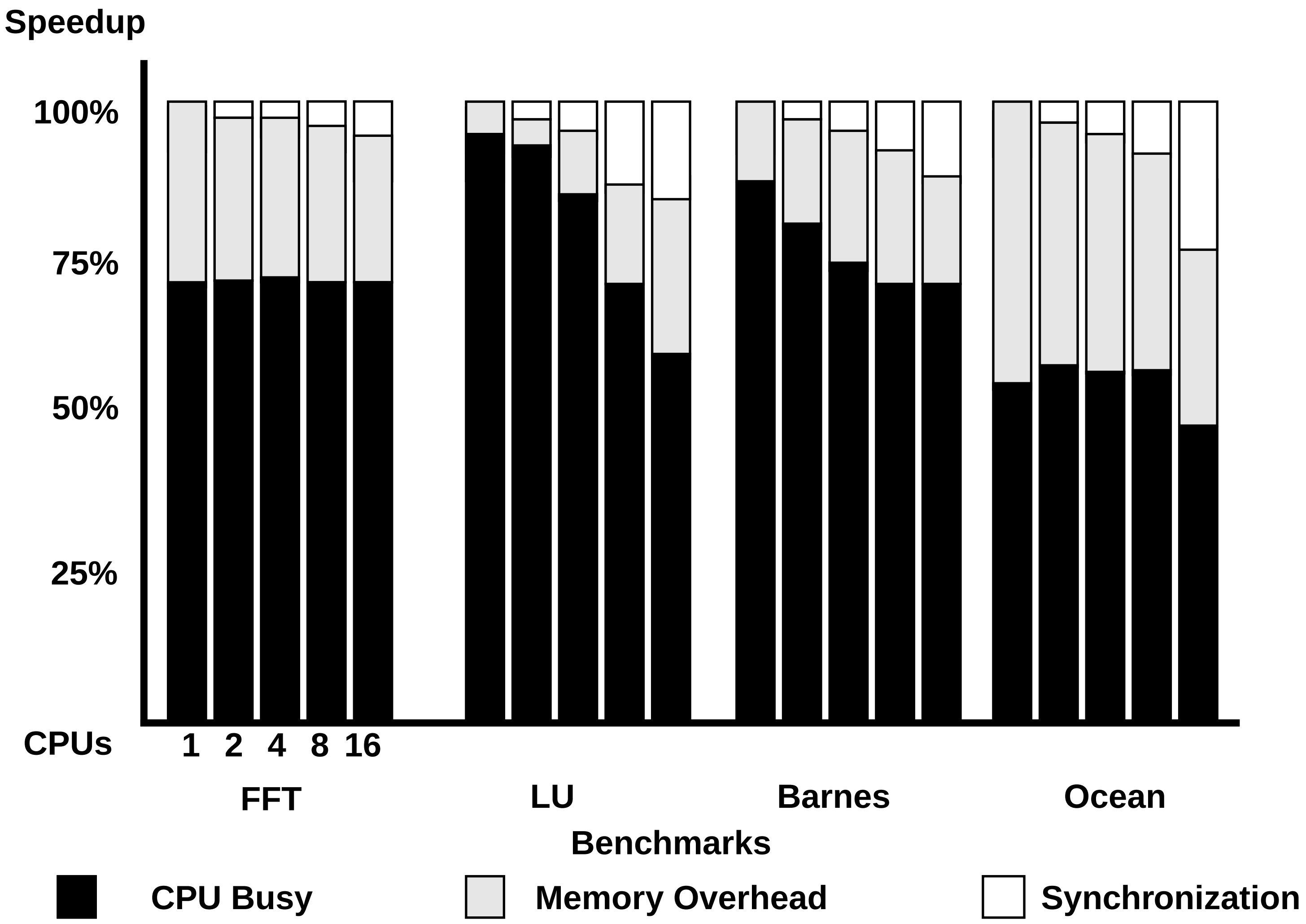


**This is always true, but not very important to most client/server type programs. Other factors are more important for them.**

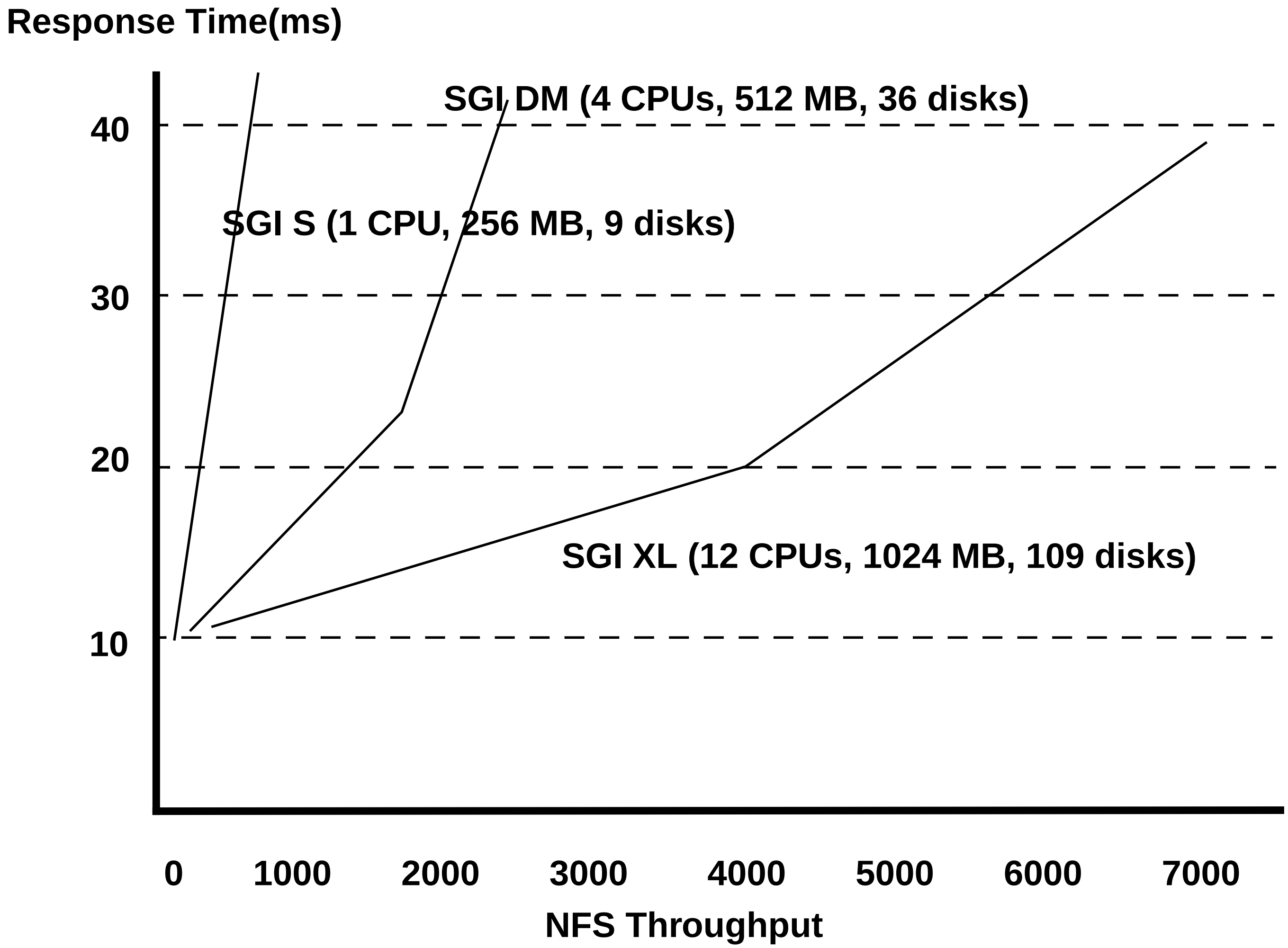
# Speedup for Parallel Benchmarks



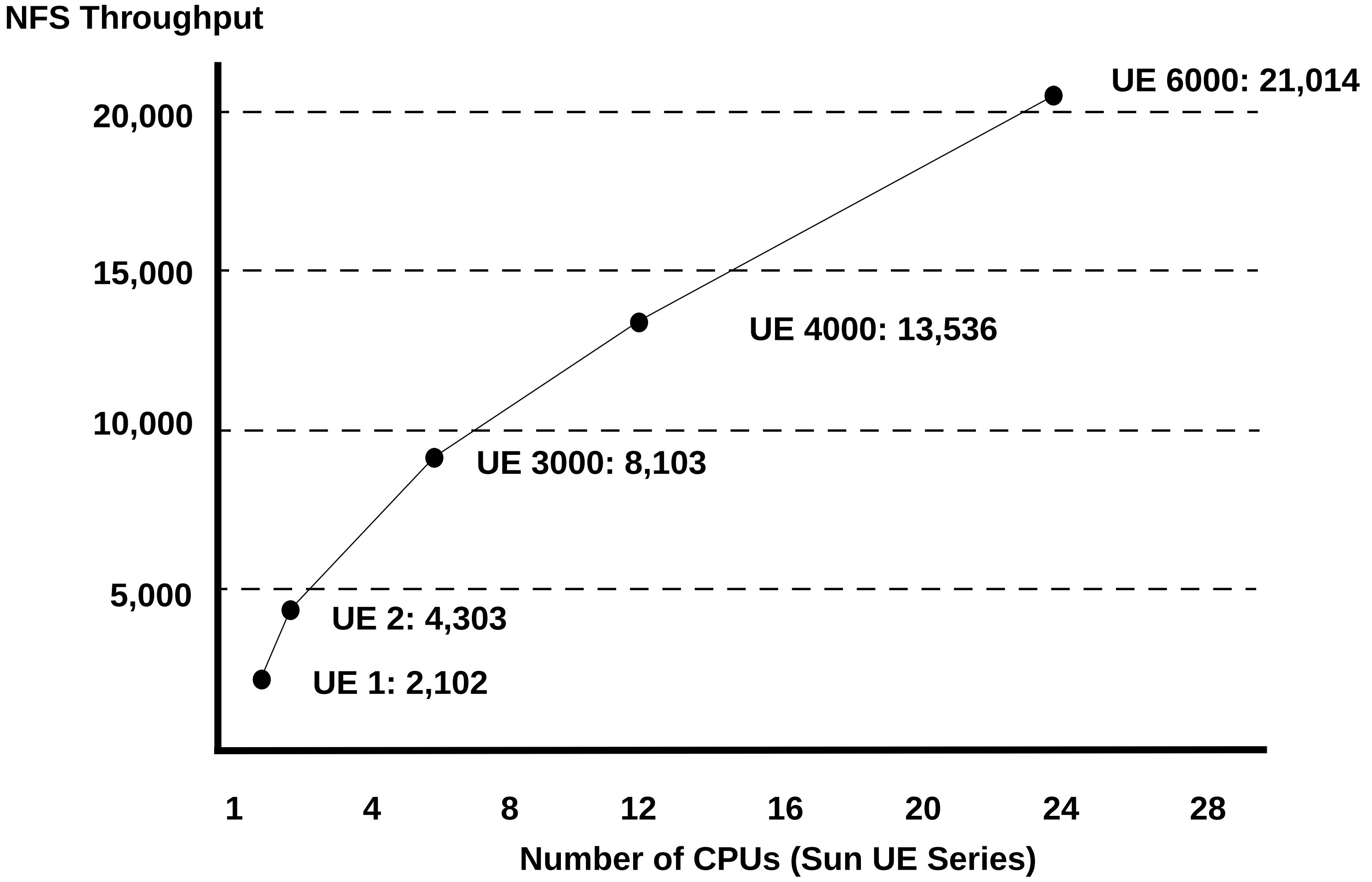
# Speedup for Parallel Benchmarks



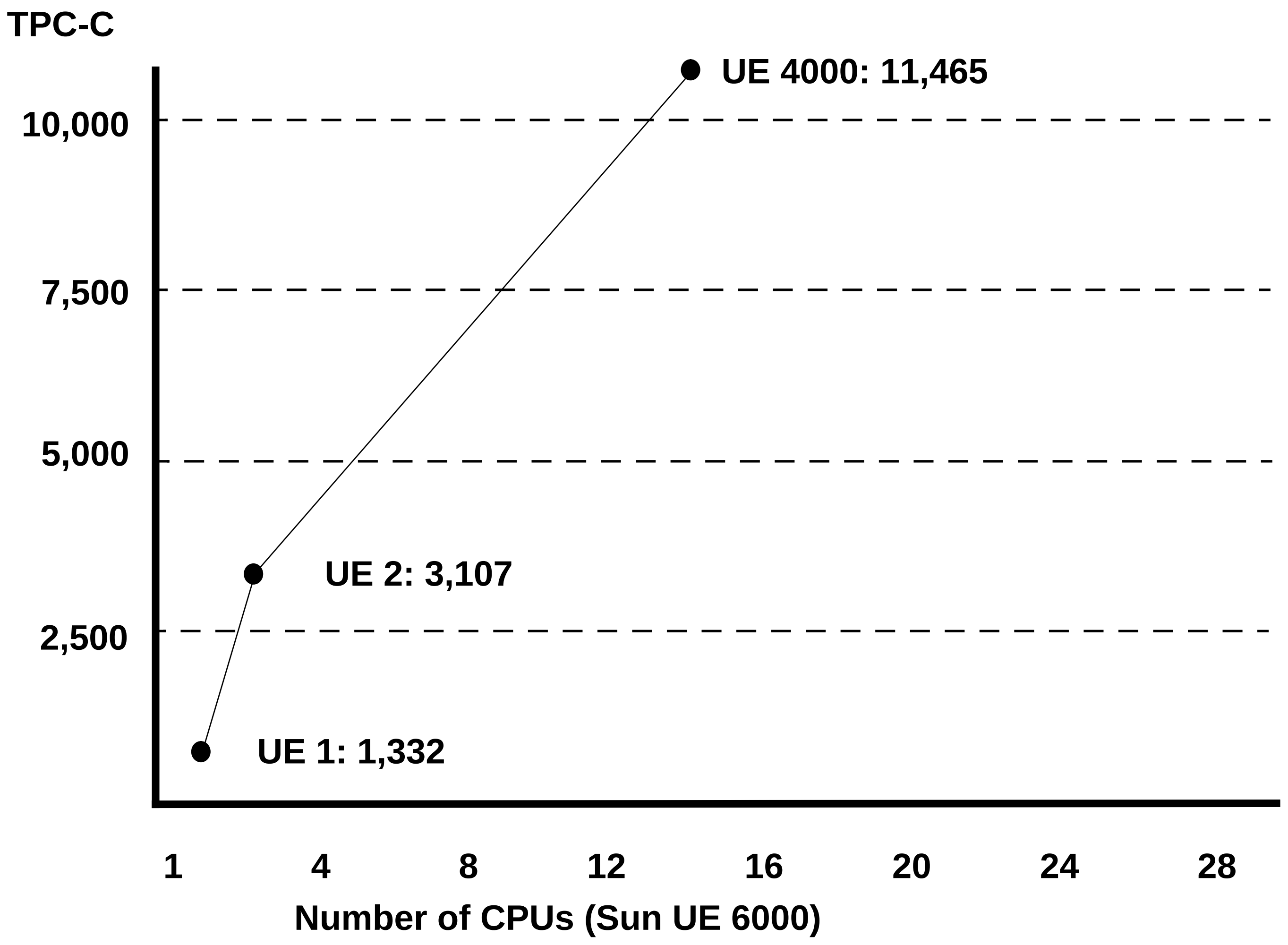
# SPEC LADDIS


Response Time(ms)
40
30
20
10
SGI DM (4 CPUs, 512 MB, 36 disks)
SGI S (1 CPU, 256 MB, 9 disks)
SGI XL (12 CPUs, 1024 MB, 109 disks)
0
1000
2000
3000
4000
5000
6000
7000
NFS Throughput

# SPEC LADDIS


NFS Throughput
20,000
15,000
10,000
5,000
UE 6000: 21,014
UE 4000: 13,536
UE 3000: 8,103
UE 2: 4,303
UE 1: 2,102
1
4
8
12
16
20
24
28
Number of CPUs (Sun UE Series)

# TPC-C Performance



# How Many LWPs?

- **CPU bound: 1 LWP per CPU**
  - **The Solaris 2.4 kernel is not so perfectly tuned that everything runs exactly as you’d want. Using bound (globally scheduled) threads may make a difference**
  - **The Solaris 2.5 kernel has not shown any such anomalies (yet).**
- **I/O bound:1 LWP per outstanding I/O request + #CPUs**
- **How to count CPUs:**

```
#include <unistd.h>

  printf(NCPUS: %d\n”,
sysconf(_SC_NPROCESSORS_ONLN));
```

# Lies, Damn Lies, and Statistics

**When you do your performance evaluation, comparing your program running on one CPU vs. many, be aware: It won’t be fair.**

**Comparing an MT program running one CPU vs. running on many is not the same as comparing with a single threaded program running on one CPU. The MT program *will* be slower (it’s locking mutexes, etc., unnecessarily), making your comparison look better.**

**Such is life.**

# How Many LWPs?

Only UI (and extensions to POSIX) has the notion of controlling the number of LWPs for locally scheduled threads to use.

- UI Threads:

```
int thr_getconcurrency (void);
int thr_setconcurrency (int concurrency);
```

- In UNIX98:

```
int pthread_getconcurrency (void);
int pthread_setconcurrency (int concurrency);
```

Tells the library how many LWPs the programmer wants for locally scheduled threads

The library will try to supply that number, but there are no guarantees. The program must work correctly anyway!

# How Many LWPs For Java?

**Java has no notion of LWPs (this is quite reasonable).**

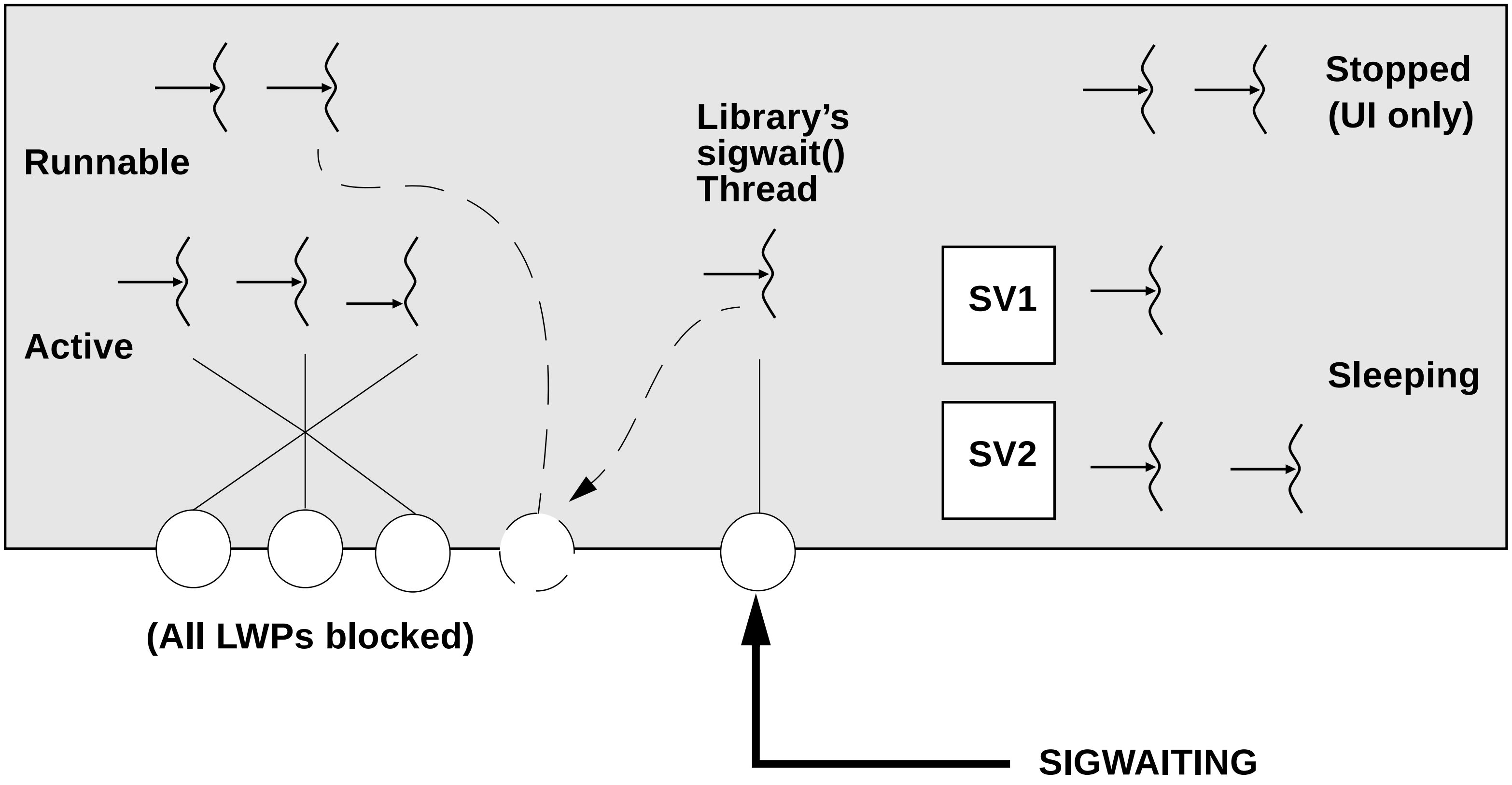
**Unfortunately, under Solaris, the default is one LWP. This is a real problem for many applications. The idea is that Solaris' automatic mechanism (SIGWAITING) will add the requisite LWPs when needed for I/O bound programs. Unfortunately it doesn't do anything for CPU-bound programs (and it doesn't work very well anyway).**

**Under Solaris (but not NT, Digital UNIX (Tru64), others?) you have to make a call to the native threads library to tell it to add LWPs. This makes your program non-portable!!**

**Starting with Solaris 8, there is a new PThreads library which provides *only* bound threads, thus solving this problem.**

**CF: `TimeDisk`, which does exactly this.**

# SIGWAITING

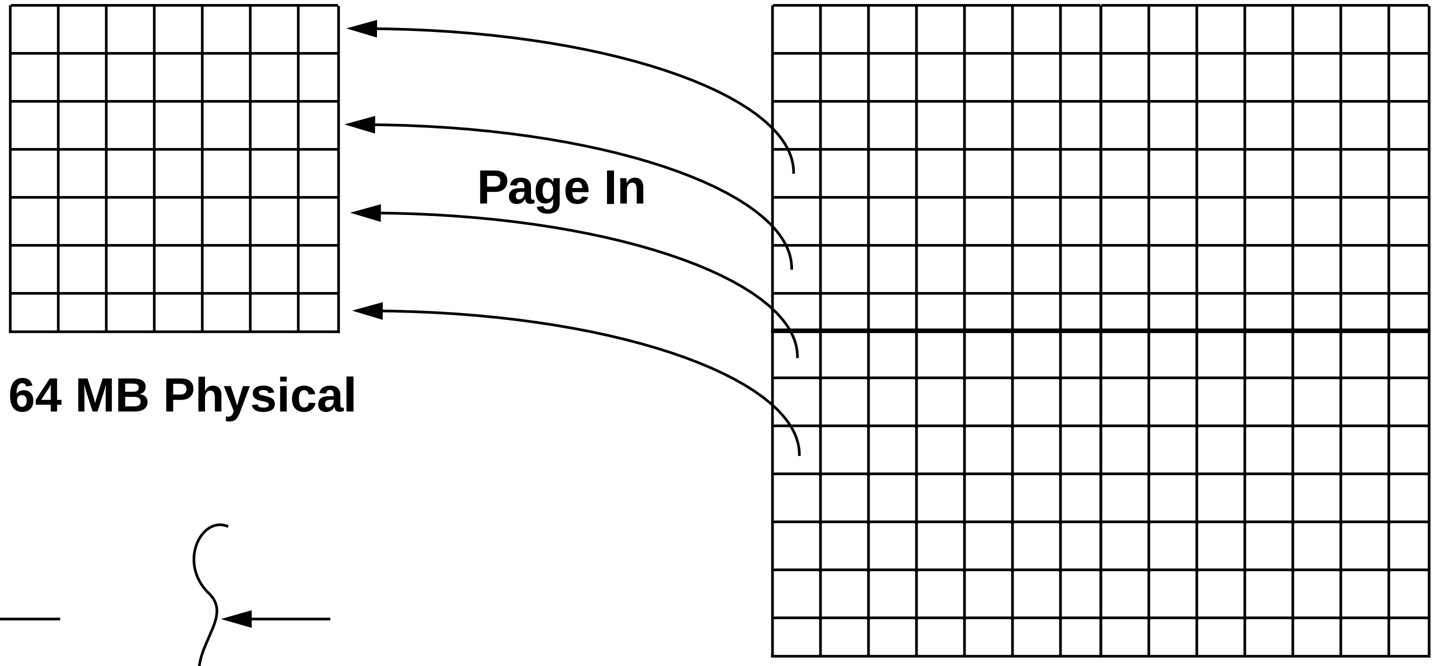


**If all LWPs are blocked, Solaris will send SIGWAITING to see if the process might like to create another one.**

# Short-Lived Threads

- 10 x 10 Matrix Multiply
    - 2000 FP Ops @ 100MFLOPS = 20µs
- 100 x 100 Matrix Multiply
    - 2M FP Ops @ 100MFLOPS = 20ms
- Solaris Thread Create ~ 100µs (SS10/41)
- Bad Example: EDA simulation, containing thousands of 10µs tasks
- Good Example: NFS, containing hundreds of 40ms tasks

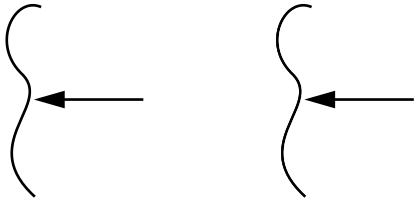
# Finessing Page Faults



Page In

64 MB Physical

256 MB Image
in Virtual Memory



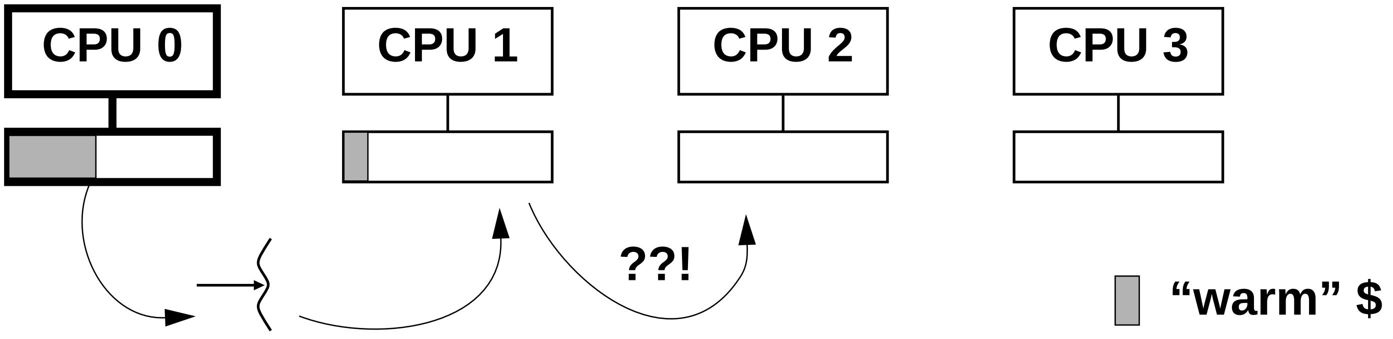
Work
Page In
Work
Page In
Work

Page In
Work
Page In
Work
Page In
Work

E.g., Adobe, Scitex

Page Faults are per-LWP!

# Processor Affinity



To completely load a 1 MB cache: 32ms

Default Solaris time slice (aka clock tick) = 10ms.

Therefore, it is very likely that the CPU which a given LWP last ran on still has an 80% valid cache for that LWP.

Solaris will delay the scheduling of an LWP for up to 3 clock ticks in order to regain the CPU which that LWP last ran on. Other systems have similar designs.

# When to Use Global Scheduling

- **In a perfect world with flawless CPU, system, and O/S design, you would only use global scheduling for:**
  - **Real time tasks**
  - **Signal handling (which is sort of a real-time task!)**
- **In the current world you *may* find global scheduling useful for:**
  - **CPU intensive applications (typically numerical programs)**
  - **Imposed time sharing**

## *The old Advice!!*

**(Before committing to global scheduling, make doubly sure that you really need it!)**

# When to Use Global Scheduling

- **The majority of programs I have seen create threads that they intend to use either as computational engines, or as I/O waiting threads. Global scheduling makes sense here.**

***The new Advice!!***

# When to Bind to a CPU

- **Never**
- **When hacking kernel code**
- **When you know more than I do**

**`processor_bind()` is the Solaris interface.**

# TSD Performance Considerations.

| Function | Microseconds | Ratio |
|---|---|---|
| Reference a Global Variable | 0.02 | 1 |
| Get Thread-Specific Data | 0.79 | 40 |

In one example, a company (unnamed!) decided to directly replace every global reference with a call to TSD. They had used a lot of globals, and the program went much slower.

110MHz SPARCstation 4 (This machine has a SPECint of about 20% of the fastest workstations today.) running Solaris 2.5.1.

# TSD Hacks

## Normal Use of TSD

:

```
{for (i ...)
  {v = (int) pthread_getspecific(v_key);
  s += f(v);
  }
```

## Cached Use of TSD

```
{v = (int) pthread_getspecific(v_key);
  for (i ...) s += f(v);
}
```

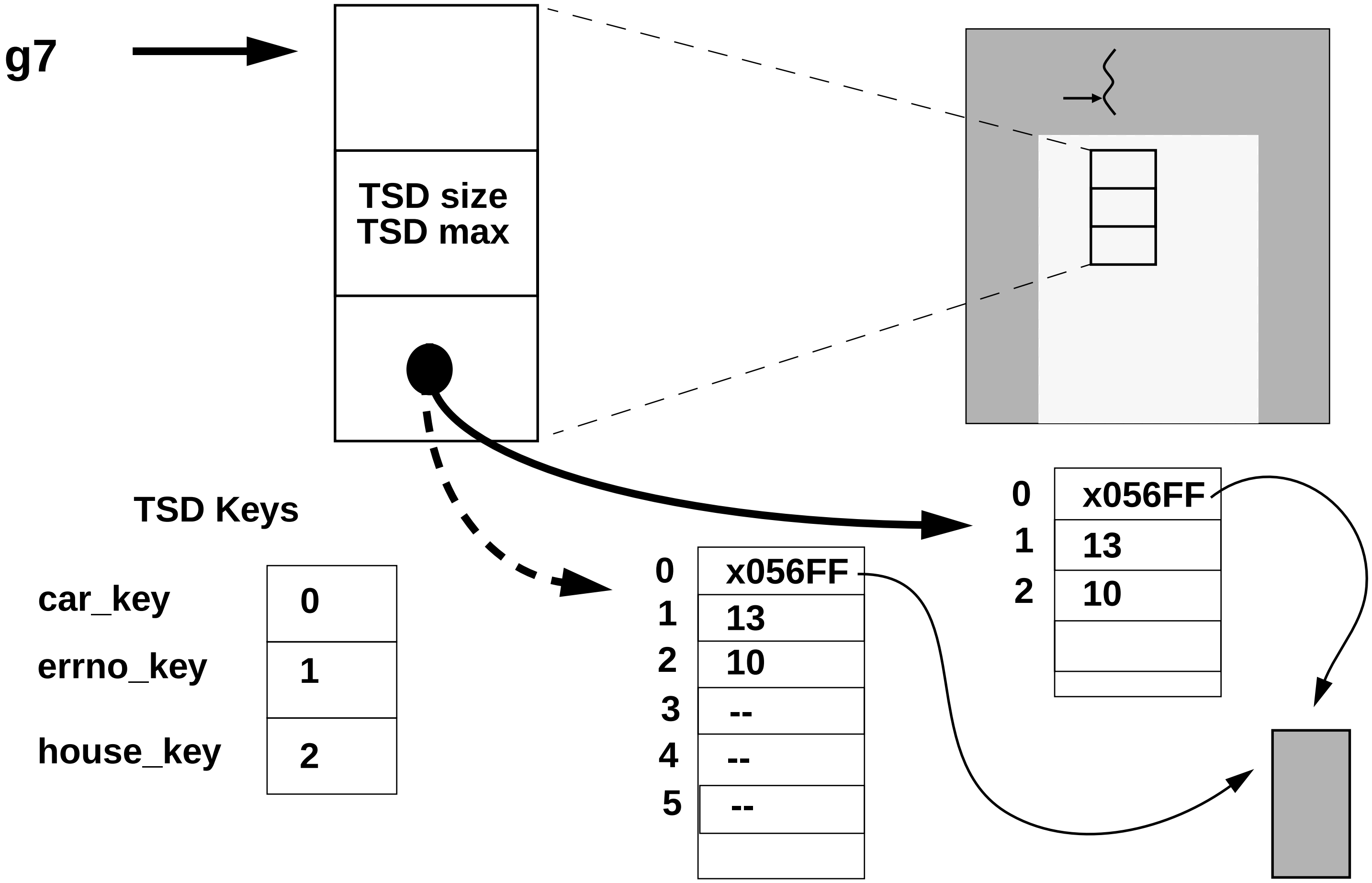
***(Be sure you need to do this!)***

# TSD Hacks

Create a structure that contains all the data you wish to be thread specific and pass that structure to every function that is going to access it. *(Be sure you need to do this!)*

```
struct MY_TSD
{ int a
  int b; }

start_routine()
{struct MY_TSD *mt;
  ...
  mt = malloc(sizeof(MY_TSD));
  mt->a = 42; mt->b = 999;
  foo(x, y, z, mt);
  bar(z, mt);
  ... }

void foo(x, y, z, struct MY_TSD *mt)
{int answer = mt->a;
...}
```

# TSD Access Technique



- Limited size TSD array *may* be dynamically expanded (Solaris). POSIX definition is minimum of 128 entries.

# Faster TSD

Hey! Why not just inline the `pthread_getspecific()` call? Then TSD could run in a couple instructions, instead of requiring a function call.

Yes, but...

That would make the thread structure part of the exported interface and then it couldn't be changed. Otherwise a program compiled on 2.5 might not work on 2.6. :-(

# Is It *Really* Faster?

**Even “simple, determistic programs” show variation in their runtimes. External interrupts, CPU selection, VM page placement, file layout on disks, etc. can cause wide variation in runtimes.**

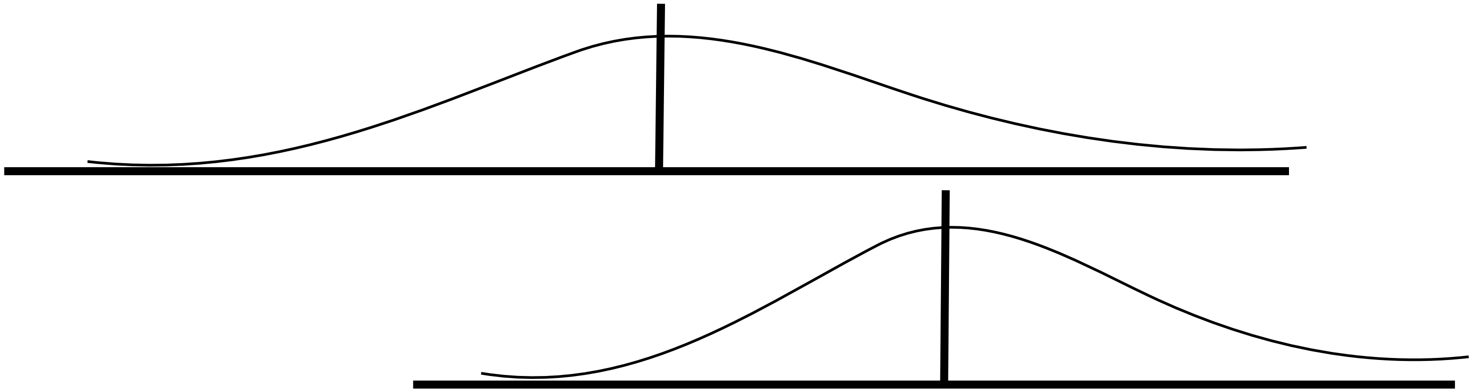
```
      Run #1                               Run #2
rate: 27.665667/sec            rate: 28.560094/sec
rate: 23.503779/sec            rate: 28.000473/sec
rate: 20.414748/sec            rate: 25.274012/sec
rate: 20.653608/sec            rate: 35.249477/sec

Mean rate: 23.05/sec           Mean rate: 28.27/sec
Standard Deviation: 3.34       Standard Deviation: 4.20
```

**The Question is: “How sure are we that the difference we measured is the difference between the actual means?”**

$$\mu_1 - \mu_2 =? x_1 - x_2$$

| | |
|---|---|
| $x_1 = 23.05$ | $s_1 = 3.34$ |
| $x_2 = 28.27$ | $s_2 = 4.20$ |

# Statistics is Your Friend



$\mu_1 - \mu_2 = x_1 - x_2$ +- $1.96\ SE(X_1 - X_2)$ (1.96 for 95% confidence)

$SE(X_1 - X_2) = (s_1^2 / n_1 + s_2^2/n_2)^{1/2}$

$( 3.34^2 / 4 + 4.20^2 / 4 )^{1/2}$ ~= 2.7

$\mu_1 - \mu_2 = 5.22$ +- 5.4

The interval includes 0. We are not 95% certain that they are different!

# Informal Statistics

**For “large” data sets (10 is a nice number) with the same standard deviation, we can eyeball the confidence interval:**

$\mu_1 - \mu_2 = x_1 - x_2 +- 3 s / (n^{1/2})$

**= 5.22    +- 3 * 3.5 / 2    ~ 5 +- 5    Insignificant difference**

```
N_PROD = 1 N_CONS = 4                 N_PROD = 1 N_CONS = 5
rate: 85.965975/sec                   rate: 89.984372/sec
rate: 86.802915/sec                   rate: 91.710778/sec
rate: 88.528658/sec                   rate: 91.075302/sec
rate: 85.411582/sec                   rate: 91.741185/sec
rate: 85.957945/sec                   rate: 87.995095/sec
rate: 84.514983/sec                   rate: 93.661803/sec
rate: 86.732842/sec                   rate: 89.505427/sec
rate: 84.284994/sec                   rate: 89.262953/sec
rate: 85.024726/sec                   rate: 89.611914/sec
rate: 85.602694/sec                   rate: 91.972079/sec
Mean rate: 85.88/sec                  Mean rate: 90.65/sec
Standard Deviation: 1.25              Standard Deviation: 1.67
```

**= 4.77    +- 3 * 1.5 / 3    ~ 5 +- 1.5    Significant difference**

# Informal Statistics

**For quick, informal estimates I will run 10 iterations, guess at the means and standard deviations, then see if the difference between the means is greater than the (average) standard deviation. If it’s much bigger, then I’m happy. If not, then it’s back to the drawing board.**

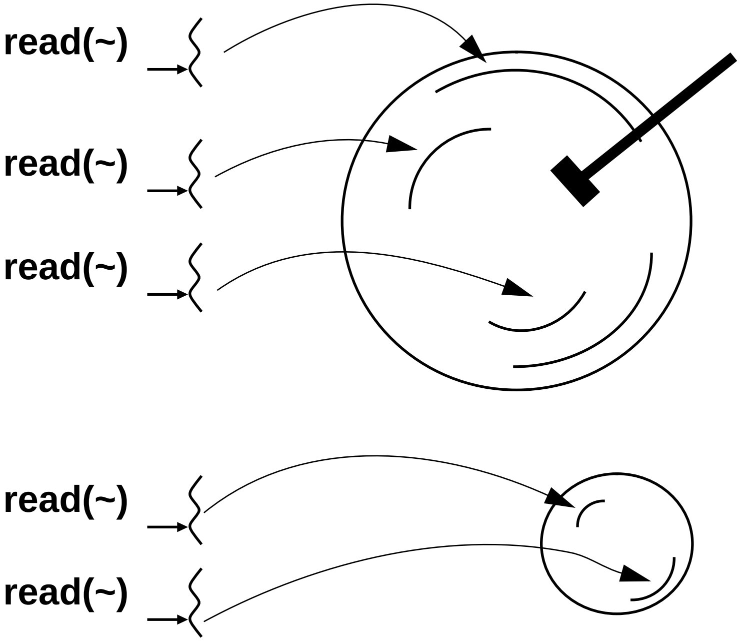
$$\mu_1 - \mu_2 = x_1 - x_2 +- 3\ s\ /\ (10^{1/2}) \quad \sim \quad \mathit{x_1 - x_2 +- s}$$

**When I’m happy with the results and have done everything I can think of, then I’ll do formal statistics for the final program.**

# *Advanced Topics*

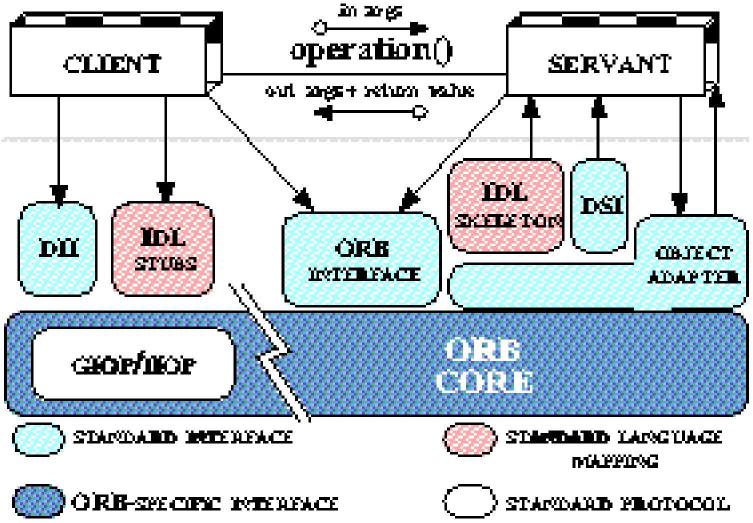
# Disk Access Rates

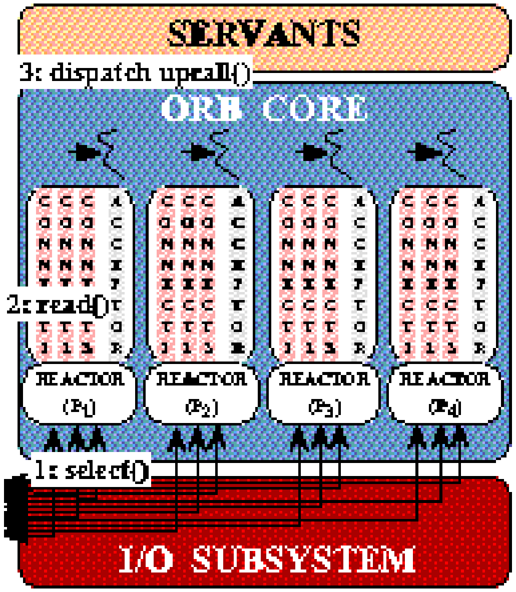
- Disk requests first check to see if the data is buffered, then go out to the physical disk.
- Individual read requests always block (typically ~20ms). Multiple requests can overlap and give better throughput.



- Rotation rates vary from 3600 rpm to 7400 rpm. (16-8ms per revolution)
- Seek times average around 10ms.
- Best overlap efficencies for random access are about 2x. Hence, ~32 threads.

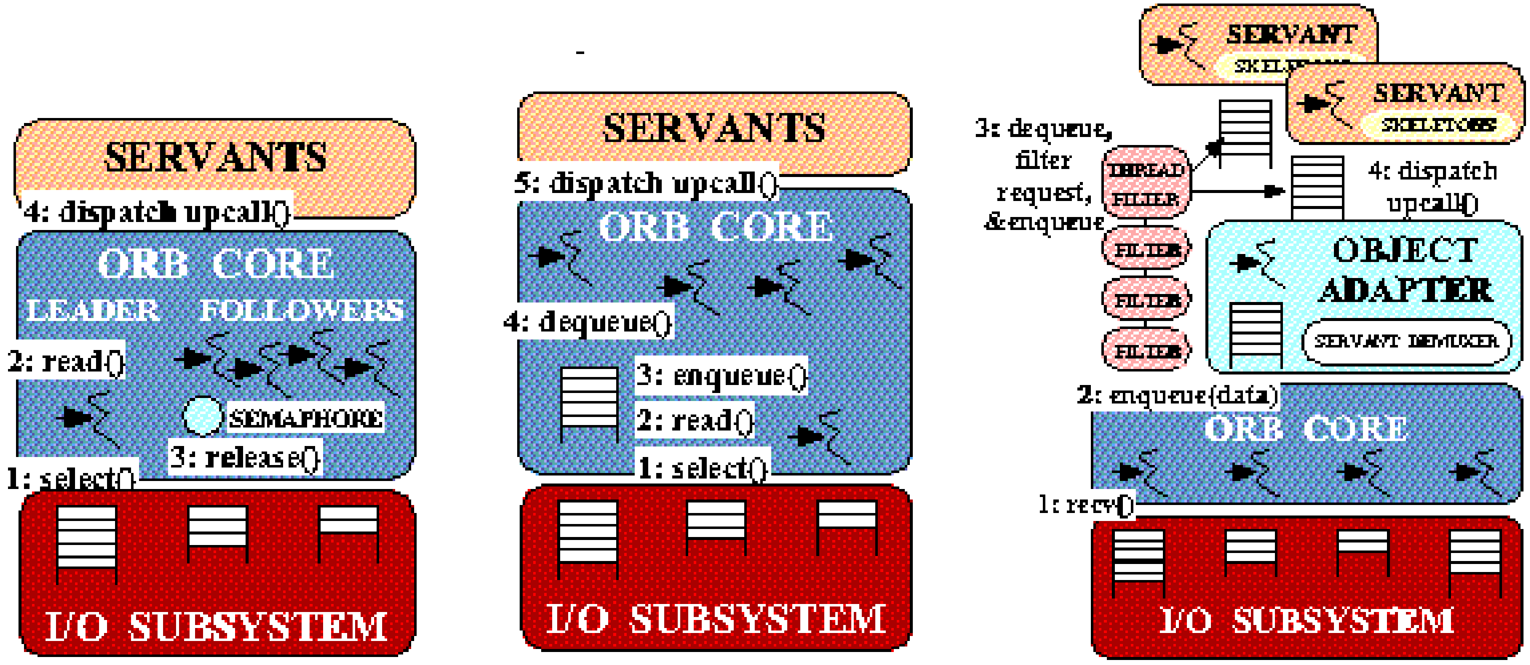
# CORBA Designs





**From *Realtime CORBA Architectures (Doug Schmidt)***

# CORBA Designs



# CORBA Performance

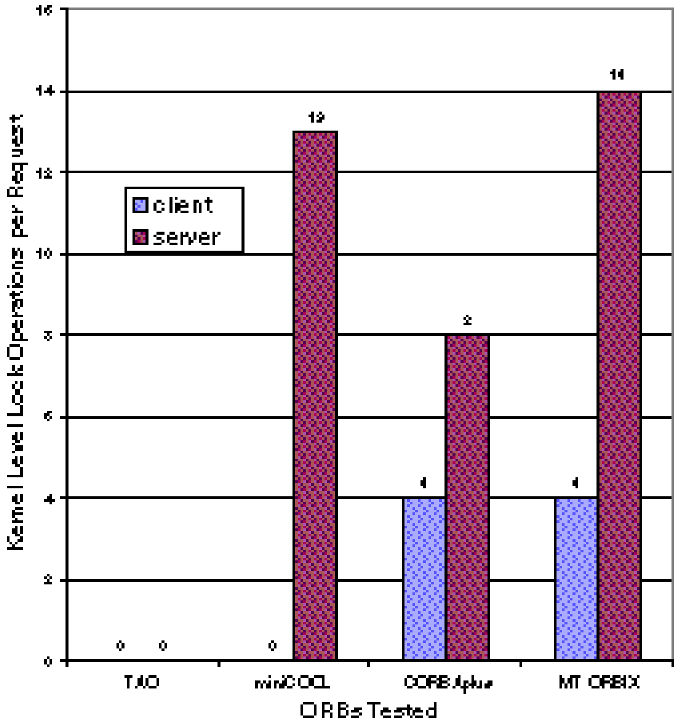


Figure 14: Kernel-level Locking Overhead in ORBs

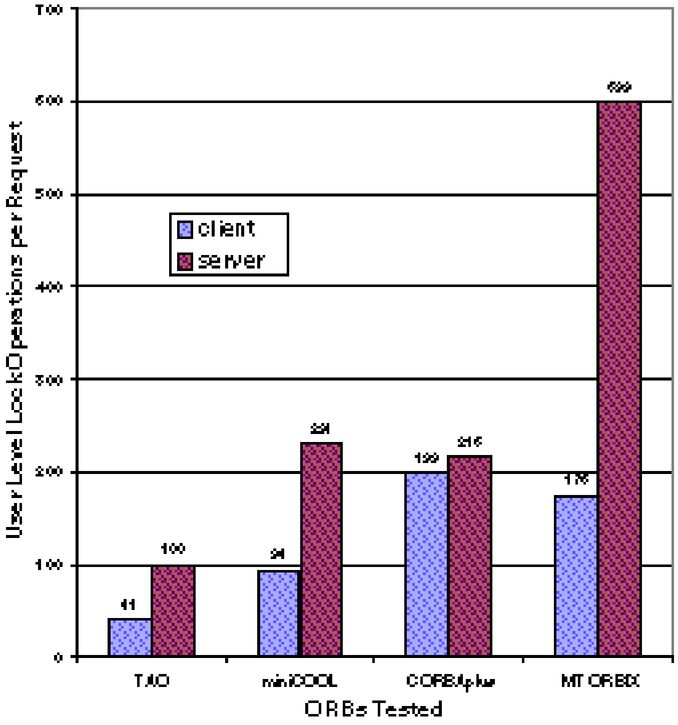
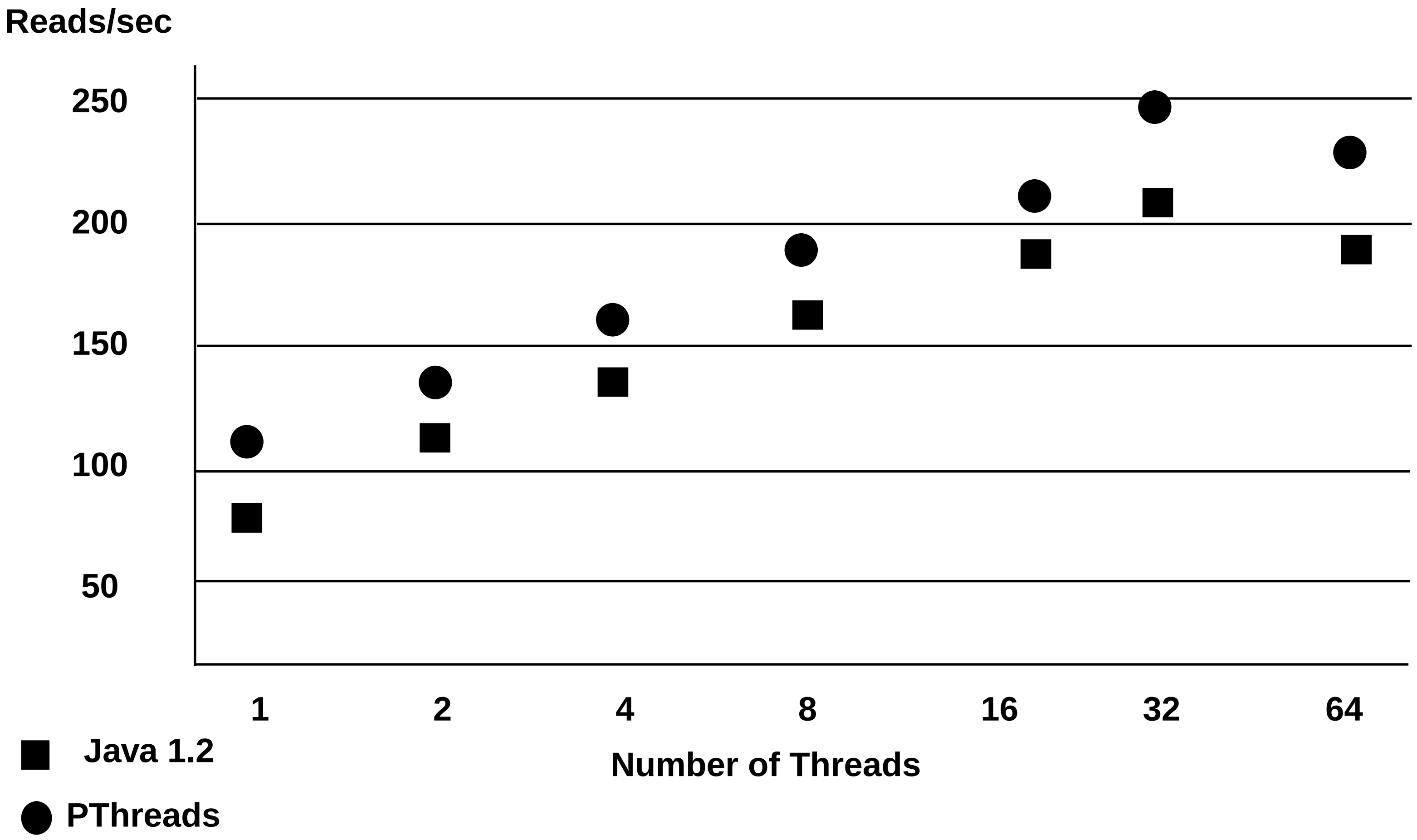


Figure 13: User-level Locking Overhead in ORBs

# Disk Performance on an SS4

**Increasing the number of threads accessing a single disk.**



# *The APIs*

# Comparing the Different Thread Specifications

| Functionality | UI Threads | POSIX Threads | Win32 Threads | Java Threads |
|---|---|---|---|---|
| Design Philosophy | Base Primitives | Near-Base Primitives | Complex Primitives | Complex Primitives |
| Scheduling Classes | Local/Global | Local/Global | Global | Vendor |
| Mutexes | Simple | Simple | Complex | Complex |
| Counting Semaphores | Simple | Simple | Buildable | Buildable |
| R/W Locks | Simple | Buildable | Buildable | Buildable |
| Condition Variables | Simple | Simple | Buildable | Buildable |
| Multiple-Object Synchronization | Buildable | Buildable | Complex | Buildable |
| Thread Suspension | Yes | Impossible | Yes | Yes |
| Cancellation | Buildable | Yes | Not usable | Not usable |
| Thread-Specific Data | Yes | Yes | Yes | Buildable |
| Signal-Handling Primitives | Yes | Yes | n/a | n/a |
| Compiler Changes Required | No | No | Yes | N/A |
| Vendor Libraries MT-safe? | Most | Most | All? | All? |
| ISV Libraries MT-safe? | Some | Some | Some | Some |

# POSIX Return Values

**POSIX allows a number of different return values from many of the functions, some of these are “problematic”. Most you would never test for. At most, write a macro that core dumps.**

- **EINVAL** — **You passed a pointer to an invalid structure.**  
  **(Either uninitialized, or you cast wrong)**  
  **No viable recovery, CRASH!**
- **EFAULT** — **You passed a pointer to an illegal address.**  
  **No viable recovery, CRASH!**
- **EINTR** — **The operation was interrupted by a signal/fork**  
  **99.9% of the time: retry action.**
- **EBUSY** — **A “try” function returned a “no”.**  
  **(Not an error.)**
- **ETIMEDOUT** — **Time limit has expired.**  
  **(Not an error.)**  
  **(NB: Solaris 2.5 bug uses ETIME)**

# POSIX Return Values

- **EPERM** — **No permission to execute privileged function**
  **Best action: CRASH! or?**
  **(Then go back and fix the program!)**
- **EAGAIN** — **Insufficient system resources (temporary).**
  **Best action: ?**
  **(e.g., `sem_trywait()`)**
- **ESRCH** — **No such thread. (Not an error.)**
- **EDEADLK** — **Causes a deadlock. (Best action: CRASH!)**
- **ENOSPC** — **Insufficient system resources.**
  **(Best action: CRASH?)**
- **ENOMEM** — **Memory is exhausted (Best action: CRASH?)**
- **ENOTSUP** — **The requested option is not supported.**
  **Best action: forget the operation and run the program without it (less efficiently).**
- **ENOSYS** — **Function not supported. (CRASH?)**

# POSIX Constants

```
Via sysconf()                          in <unistd.h>          Solaris Value
sysconf(_SC_THREADS) -> _POSIX_THREADS          1


_POSIX_THREAD_ATTR_STACKSIZE                    1
        (Can you set the stack size?)
_POSIX_THREAD_ATTR_STACKADDR                    1
        (Can you allocate the stack yourself?)
_POSIX_THREAD_PRIORITY_SCHEDULING               1
        (Can you use realtime?)
_POSIX_THREAD_PRIO_INHERIT                      --
        (Are there priority inheriting mutexes?)
_POSIX_THREAD_PRIO_PROTECT                      --
        (Are there priority ceiling mutexes?)
```

# Wrappers

```
int PTHREAD_CREATE(pthread_t *new_thread_ID,
   const pthread_attr_t *attr,
   void * (*start_func)(void *), void *arg)
{int err;
 pthread_t tid;

 if (err = pthread_create(new_thread_ID, attr, start_func, arg))
   {printf("%s\n", strerror(err));
    abort();
  }}

int PTHREAD_ATTR_INIT(pthread_attr_t *a)

int PTHREAD_CONDATTR_INIT(pthread_condattr_t *a)

int PTHREAD_MUTEXATTR_INIT(pthread_mutexattr_t *a)

int SEM_INIT(sem_t *sem, int pshared, unsigned int value)


int SEM_WAIT(sem_t *arg)/* Ignore signal interruptions */
{while (sem_wait(arg) != 0) {}}
```

# POSIX Constants

```
_POSIX_THREAD_PROCESS_SHARED                   1
        (Are there cross-process locks & CVs?)
_POSIX_THREAD_SAFE_FUNCTIONS                   1
        (Are there posix MT-safe libraries?)
```

# POSIX Constants

**in <limits.h>**

**_POSIX_THREAD_DESTRUCTOR_ITERATIONS 4**
**(Iterations if TSD destructors are circular)**

**_POSIX_THREAD_KEYS_MAX 128**
**(Maximum number of TSD keys *required***
**per process, actual is > 1,000)**

**_POSIX_THREAD_STACK_MIN 8k**
**(Smallest possible thread stack)**

**_POSIX_THREAD_THREADS_MAX 64**
**(Maximum number of threads *required***
**per process, actual is > 10,000)**

# Attribute Objects

**UI and Win32 all use flags and direct arguments to indicate what the special details of the objects being created should be. POSIX requires the use of *attribute objects*:**

```
thr_create(NULL, NULL, foo, NULL, THR_DETACHED);
```

**vs:**

```
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, foo, NULL);
```

# Attribute Objects

**Although a bit of a pain in the *** compared to passing all the arguments directly, attribute objects allow the designers of the threads library more latitude to add functionality without changing the old interfaces. (If they decide they really want to, say, pass the signal mask at creation time, they just add a function `pthread_attr_set_signal_mask()` instead of adding a new argument to `pthread_create()`.)**

**There are attribute objects for:**

- **Threads**
  - **stack size, stack base, scheduling policy, scheduling class, scheduling scope, scheduling inheritance, detach state**
- **Mutexes**
  - **Cross process, priority inheritance**
- **Condition Variables**
  - **Cross process**

# Attribute Objects

Attribute objects (AOs) must be:

- allocated
- initialized
- values set (presumably)
- used
- destroyed (only required if they are to be free’d)

```
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, foo, NULL);
pthread_create(&tid, &attr, bar, NULL);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
pthread_create(&tid, &attr, baz, NULL);
pthread_attr_destroy(&attr);
```

# SV and AO Allocation

You normally declare these statically:

```
pthread_mutex_t           m;
pthread_cond_t            c;
sem_t                     s;
pthread_mutexattr_t       mattr;
pthread_condattr_t        cattr;
pthread_attr_t            tattr;
```

But you may wish to allocate some dynamically also:

```
pthread_mutex_t     *mp;

mp = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(mp, NULL);
...
pthread_mutex_lock(mp);
...
```

(I can’t think of a good reason ever to allocate attribute objects dynamically!)

# SV Initialization

Condition variables and mutexes both have attribute objects. Semaphores don’t.

All SVs must be initialized. Condition variables and mutexes allow static initialization also:

```
pthread_mutex_t    m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t     c = PTHREAD_COND_INITIALIZER;

                     versus

pthread_mutexattr_t mattr;
pthread_condattr_t cattr;

pthread_mutexattr_init(&mattr);
pthread_condattr_init(&cattr);
pthread_mutex_init(&m, &mattr);          /* mattr could be NULL */
pthread_cond_init(&c, &cattr);           /* cattr could be NULL */
```

No static initialization for any attribute objects.

# SV and AO Destruction

**If (and only if) you allocated these dynamically and you are about to call `free()` on the pointer, you must destroy them. There may be other storage pointed to by the objects themselves which you don’t know about, and that would cause a leak.**

```
pthread_mutex_t    *mp;

foo()
{
  mp = (pthread_mutex_t *) malloc(sizeof pthread_mutex_t);
  pthread_mutex_init(mp, NULL);
  ...
  pthread_mutex_lock(mp);
  ...
  pthread_mutex_unlock(mp);                  /* Must unlock! */
  pthread_mutex_destroy(mp);
  free(mp);
}
```

# Semaphores

**POSIX semaphores existed before the creation of the threads standard, and are not “really” part of pthreads. Hence their inconsistent naming conventions and use of flags instead of attribute objects.**

**There are also named semaphores, which allow you to have unrelated processes use the same semaphore.**

```
Unnamed Semaphores

sem_init(&sem, shared, initial_value);
sem_getvalue(&sem);


Named Semaphores

sem_t *sem_open(const char *name, int oflag);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

# Java Thread-Related Exceptions

There are a small number of exceptions which the various functions can throw.

| | |
|---|---|
| InterruptedIOException | Someone called `interrupt()`. |
| InterruptedException | Someone called `interrupt()`. |
| IllegalThreadStateException | Method is only valid if the thread is (or isn’t) running/suspended. |
| IllegalArgumentException | `setPriority(p)` called,<br>p out of range. |
| IllegalMonitorStateException | `wait()` or `notify()` called without holding the lock. |
| SecurityException | `setPriority(p)` called, p too high.<br>Or `checkAccess()` failed. |

# *Compiling*

# Solaris Libraries

- Solaris has three libraries: `libthread.so` (UI threads), `libpthread.so` (POSIX threads), `libposix4.so` (POSIX semaphores)
- Corresponding new include files: `synch.h` (UI synchronization variables), `thread.h` (UI threads), `pthread.h` (POSIX threads), `posix4.h` (POSIX semaphores)
- Bundled with all O/S releases/bundles (e.g., user bundle)
  - Running an MT program requires no extra effort
  - Compiling an MT program requires only a compiler (any compiler!)
  - Writing an MT program requires only a compiler (but a few MT tools will come in very handy)
  - The .h files are only included in the developer bundle (of course).

# Win32 Libraries

- has one library: `libthread`?
- Corresponding New Include Files: `thread.h`?
- Bundled with all O/S releases?
    - Running an MT program requires NO extra effort
    - Compiling an MT program requires only a compiler (Win32 requires a special compiler)
    - Writing an MT program requires only a compiler (but a few MT tools will come in very handy)

# Compiling UI under Solaris

- Compiling is no different than for non-MT programs
    - libthread is just another system library in /usr/lib
    - Example:

```
%cc -o sema sema.c -lthread -D_REENTRANT
%cc -o sema sema.c -mt
```

- All multithreaded programs should be compiled using the `_REENTRANT` flag
    - Applies for every module in a new application
    - If omitted, the old definitions for `errno`, `getc`, etc. would be used, which you don’t want
    - `#define _REENTRANT` in the file is fine.
- All MT-safe libraries should be compiled using the `_REENTRANT` flag, even though they may be used single in a threaded program

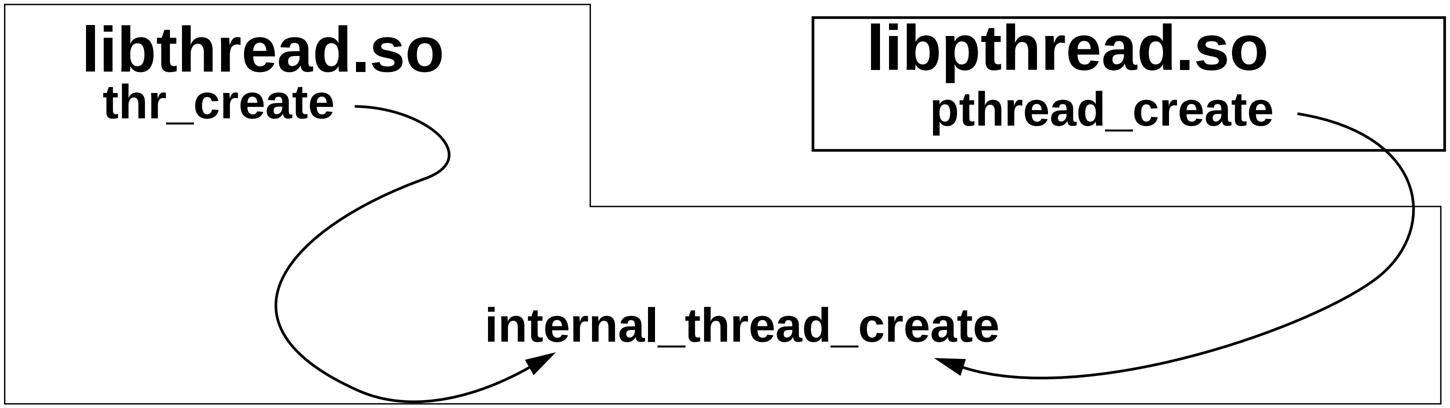
# Compiling POSIX (Solaris)

- **Compiling is no different than for non-MT programs**
  - **`libpthread.so` is just another system library in `/usr/lib`**
  - **Example:**

```
%cc -o sema sema.c -lpthread -lposix4
     -D_POSIX_C_SOURCE=199506L
```

- **All multithreaded programs should be compiled using the `_POSIX_C_SOURCE=199506L` flag**
  - **Applies for every module in a new application**
  - **If omitted, the old definitions for `errno`, `getc`, etc. would be used, which you don't want**
  - **`#define _POSIX_C_SOURCE=199506L` in the file OK.**
  - **All libraries should use the flag, even though they may be intended for a single-threaded program.**

# Compiling mixed UI/POSIX (Solaris)



libthread.so

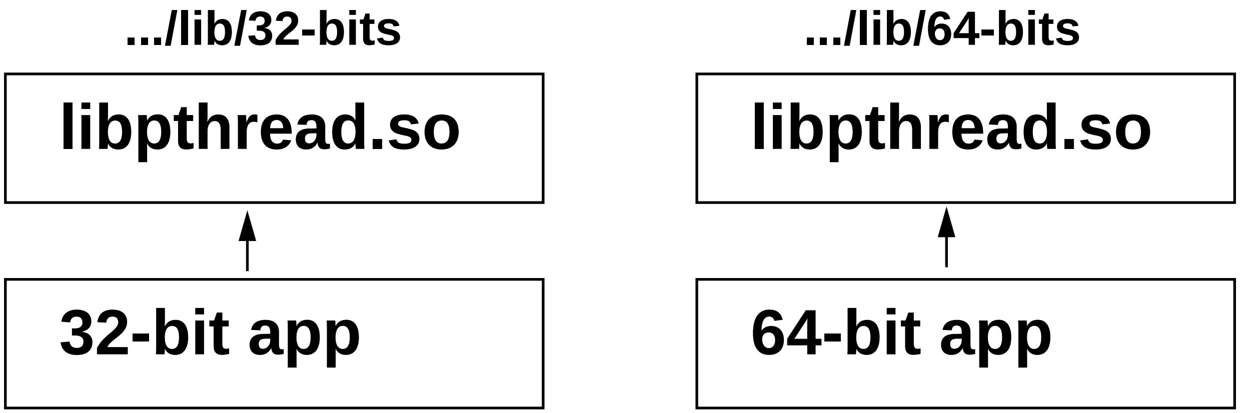
thr_create

libpthread.so

pthread_create

internal_thread_create

# 64-bit Libraries (Solaris 2.7)



.../lib/32-bits

.../lib/64-bits

libpthread.so

libpthread.so

32-bit app

64-bit app

# Compiling mixed UI/POSIX under Solaris

- **If you just want to use the UI thread functions (e.g., `thr_setconcurrency()`)**

```
%cc -o sema sema.c -lthread -lpthread -lposix4
     -D_REENTRANT -D_POSIX_PTHREAD_SEMANTICS
```

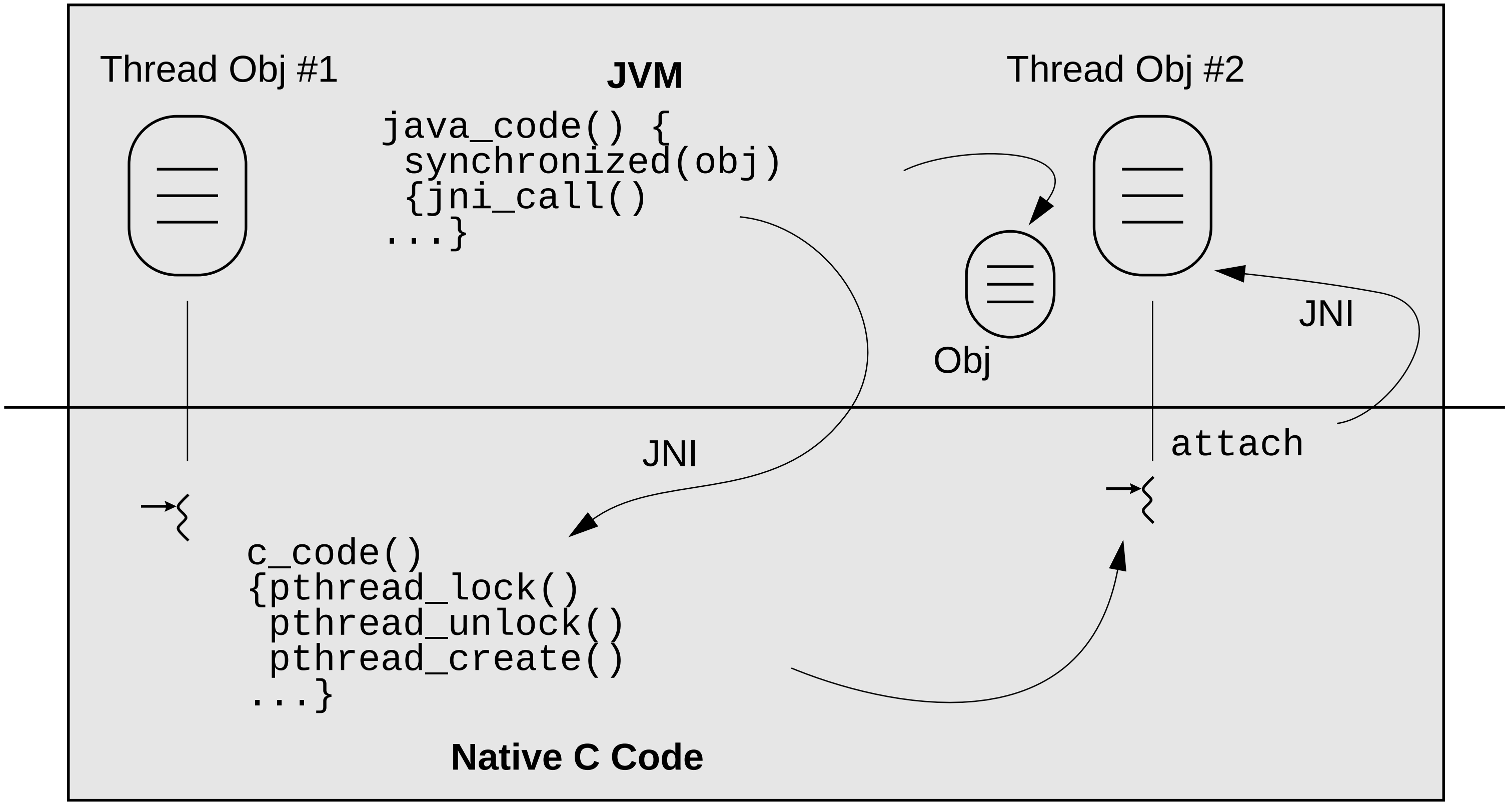
- **If you also want to use the UI semantics for `fork()`, alarms, timers, `sigwait()`, etc*.**

```
%cc -o sema sema.c -lthread -lpthread -lposix4
     -D_REENTRANT
```

**(Don’t do this!)**

**`* (etc: ctime_r, ftrylockfile, getgrnam_r, getpwnam_r, readdir_r, asctime_r, getlogin_r, getgrgid_r, getpwuid_r, ttyname_r)`**

# Java Threads Use Native Threads



```
java_code() {
 synchronized(obj)
 {jni_call()
...}
```

```
c_code()
{pthread_lock()
 pthread_unlock()
 pthread_create()
...}
```

**You can make JNI calls to native C code, which can make JNI calls back to the JVM.**

# *Tools*

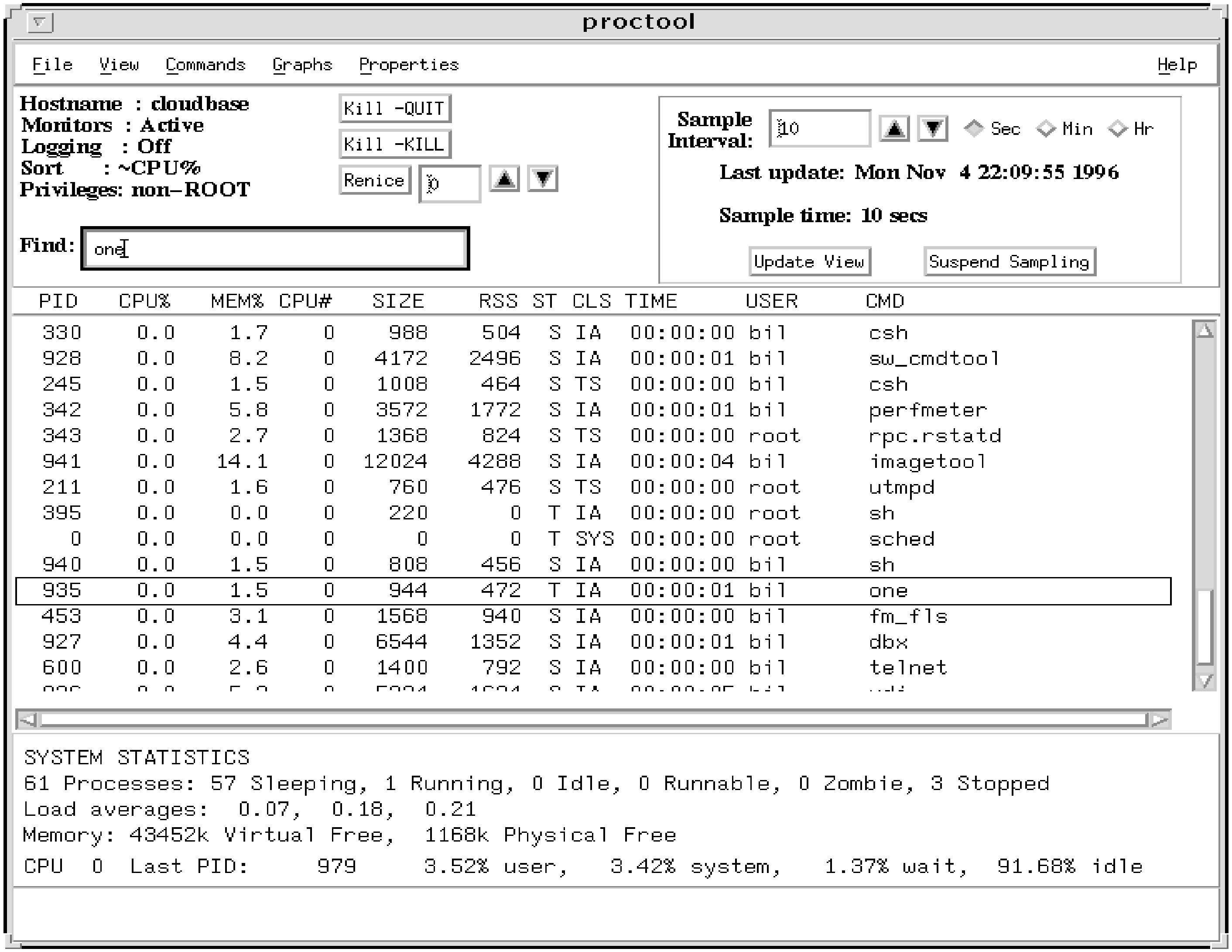
# De Bugs in Debugger

**In Sun’s older debugger, there are some problems...**

```
()^C Cannot interrupt in Critical Section

() step (into a locked mutex!)

^c ^c ^c !%@$%&#$^&#$!!!!!
```

```
% ps -el | grep my_program

352 console  0:02 perfmete
348 console  0:01 imagetoo
804 pts/1    0:00 telnet
412 pts/3    0:01 my_program

%kill -9 412
```

# Proctool: System Window



Hostname : cloudbase
Monitors : Active
Logging : Off
Sort : ~CPU%
Privileges: non-ROOT

Renice: 0

Find: one

Sample Interval: 10 ◆ Sec ◇ Min ◇ Hr

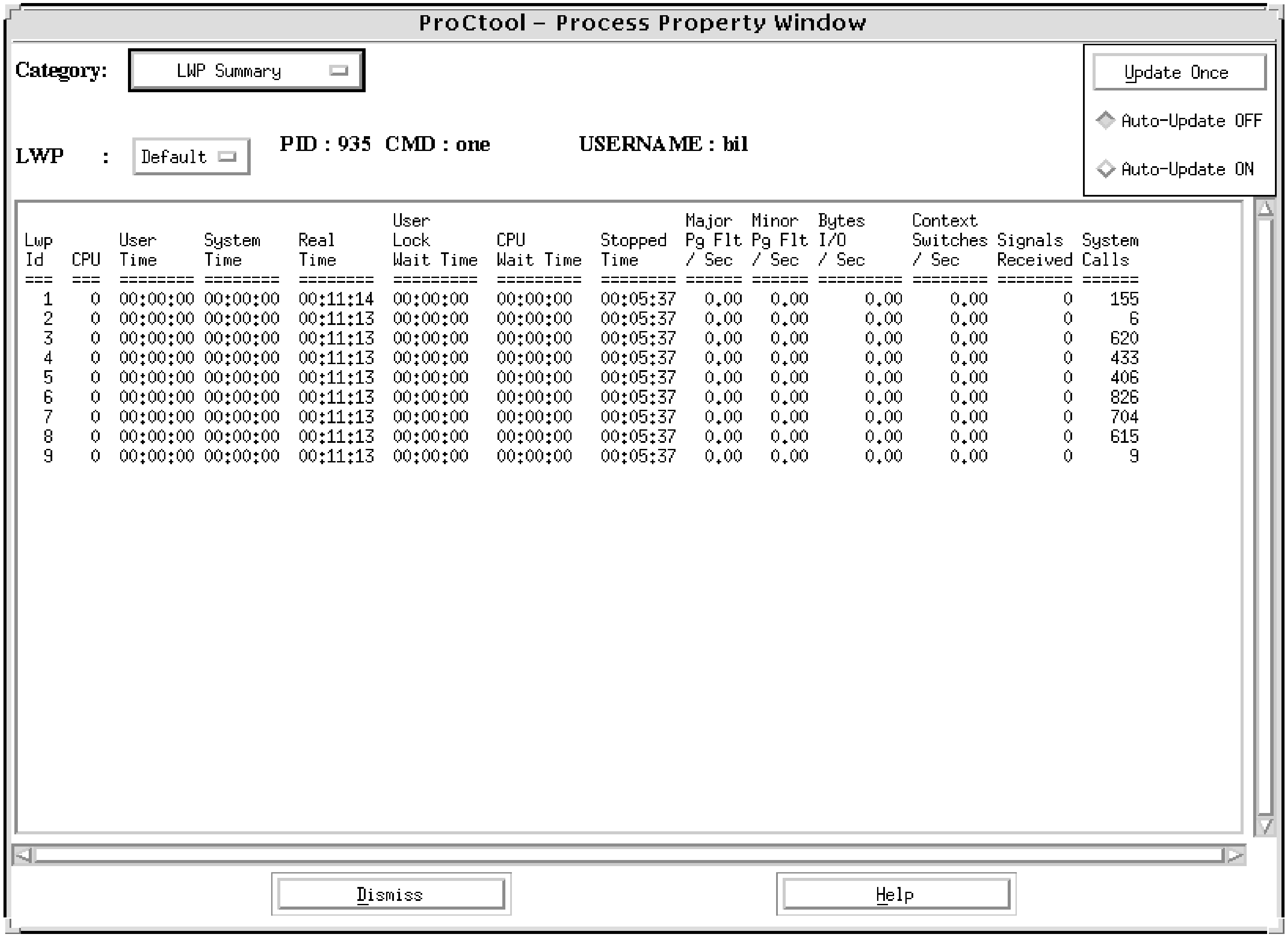
Last update: Mon Nov 4 22:09:55 1996

Sample time: 10 secs

```
 PID    CPU%    MEM% CPU#   SIZE    RSS ST CLS TIME      USER     CMD
 330     0.0     1.7    0    988    504  S IA  00:00:00 bil      csh
 928     0.0     8.2    0   4172   2496  S IA  00:00:01 bil      sw_cmdtool
 245     0.0     1.5    0   1008    464  S TS  00:00:00 bil      csh
 342     0.0     5.8    0   3572   1772  S IA  00:00:01 bil      perfmeter
 343     0.0     2.7    0   1368    824  S TS  00:00:00 root     rpc.rstatd
 941     0.0    14.1    0  12024   4288  S IA  00:00:04 bil      imagetool
 211     0.0     1.6    0    760    476  S TS  00:00:00 root     utmpd
 395     0.0     0.0    0    220      0  T IA  00:00:00 root     sh
   0     0.0     0.0    0      0      0  T SYS 00:00:00 root     sched
 940     0.0     1.5    0    808    456  S IA  00:00:00 bil      sh
 935     0.0     1.5    0    944    472  T IA  00:00:01 bil      one
 453     0.0     3.1    0   1568    940  S IA  00:00:00 bil      fm_fls
 927     0.0     4.4    0   6544   1352  S IA  00:00:01 bil      dbx
 600     0.0     2.6    0   1400    792  S IA  00:00:00 bil      telnet
```

```
SYSTEM STATISTICS
61 Processes: 57 Sleeping, 1 Running, 0 Idle, 0 Runnable, 0 Zombie, 3 Stopped
Load averages:  0.07,  0.18,  0.21
Memory: 43452k Virtual Free,  1168k Physical Free
CPU  0  Last PID:     979     3.52% user,   3.42% system,   1.37% wait,  91.68% idle
```

# Proctool: Process Window (LWPs)



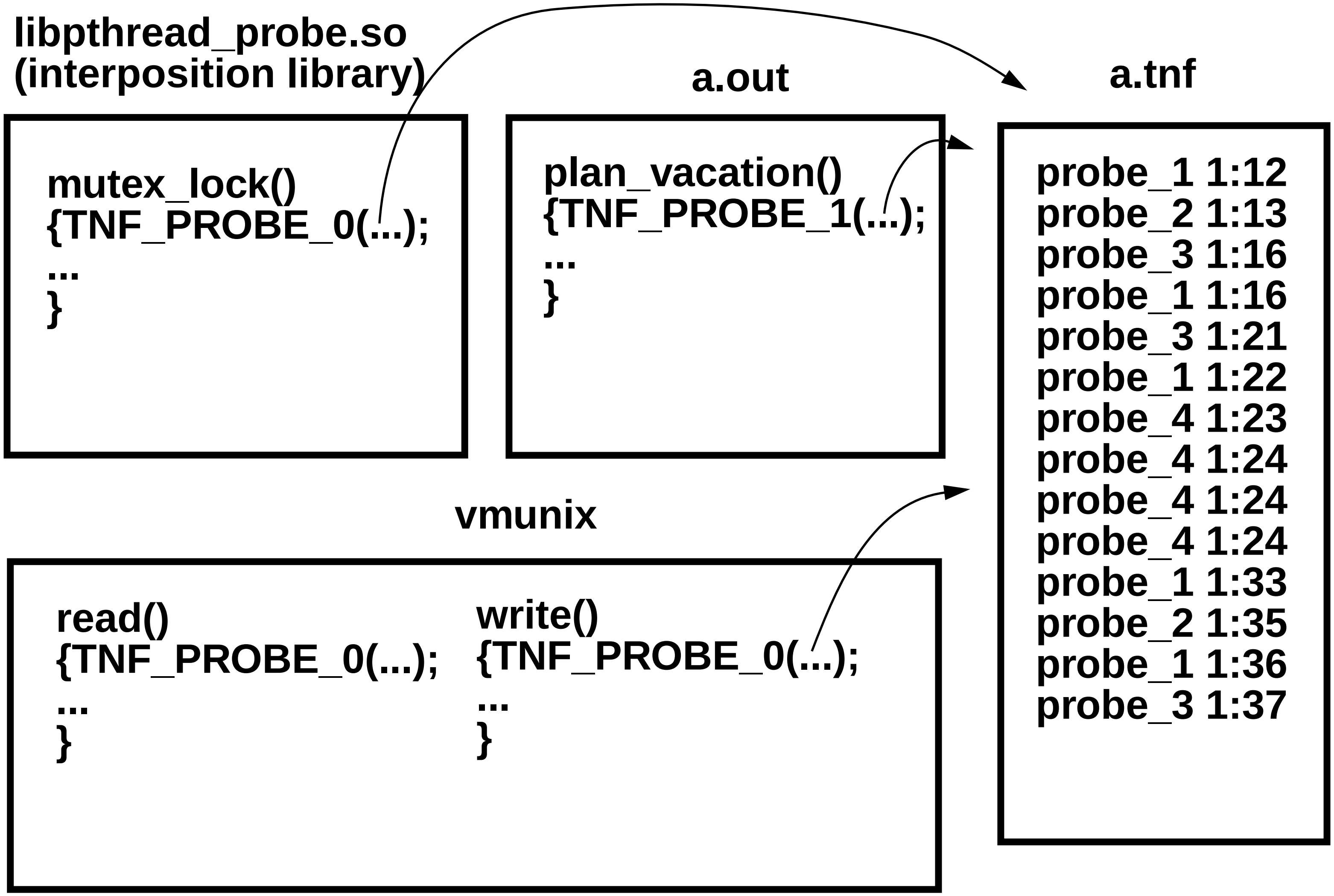
Category: LWP Summary

LWP : Default

PID : 935 CMD : one USERNAME : bil

```
                                       User                           Major  Minor  Bytes     Context
Lwp       User     System    Real      Lock       CPU        Stopped  Pg Flt Pg Flt I/O       Switches Signals  System
Id   CPU  Time     Time      Time      Wait Time  Wait Time  Time     / Sec  / Sec  / Sec     / Sec    Received Calls
===  ===  ======== ========  ========  =========  =========  ======== ====== ====== ========= ======== ======== ======
  1    0  00:00:00 00:00:00  00:11:14  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0    155
  2    0  00:00:00 00:00:00  00:11:13  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0      6
  3    0  00:00:00 00:00:00  00:11:13  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0    620
  4    0  00:00:00 00:00:00  00:11:13  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0    433
  5    0  00:00:00 00:00:00  00:11:13  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0    406
  6    0  00:00:00 00:00:00  00:11:13  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0    826
  7    0  00:00:00 00:00:00  00:11:13  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0    704
  8    0  00:00:00 00:00:00  00:11:13  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0    615
  9    0  00:00:00 00:00:00  00:11:13  00:00:00   00:00:00   00:05:37   0.00   0.00      0.00     0.00        0      9
```

# TNF (Solaris 2.5+)



libpthread_probe.so
(interposition library)

```
mutex_lock()
{TNF_PROBE_0(...);
...
}
```

a.out

```
plan_vacation()
{TNF_PROBE_1(...);
...
}
```

a.tnf

```
probe_1 1:12
probe_2 1:13
probe_3 1:16
probe_1 1:16
probe_3 1:21
probe_1 1:22
probe_4 1:23
probe_4 1:24
probe_4 1:24
probe_4 1:24
probe_1 1:33
probe_2 1:35
probe_1 1:36
probe_3 1:37
```

vmunix

```
read()
{TNF_PROBE_0(...);
...
}
```

```
write()
{TNF_PROBE_0(...);
...
}
```

# Creating TNF Probes

```
void *give_friends_raise(void *arg)
{person_t *p, *friends = (person_t *) arg;

 TNF_PROBE_0(give_friends_raise_start,  "lgl", "");

  while(friends != NULL)
    {pthread_mutex_lock_conflicts(&people_lock);
    TNF_PROBE_0(give_one_friend_raise_start,  "lgl", "");
     p = find_person(friends->name);
    TNF_PROBE_0(give_one_friend_raise_middle,  "lgl", "");
     give_raise(p);
    TNF_PROBE_0(give_one_friend_raise_end,  "lgl", "");
     pthread_mutex_unlock(&people_lock);
     friends = friends->next;
     sched_yield();
   }
 TNF_PROBE_0(give_friends_raise_end,  "lgl", "");
  sem_post(&barrier);
}
```

# Collecting TNF Information: prex

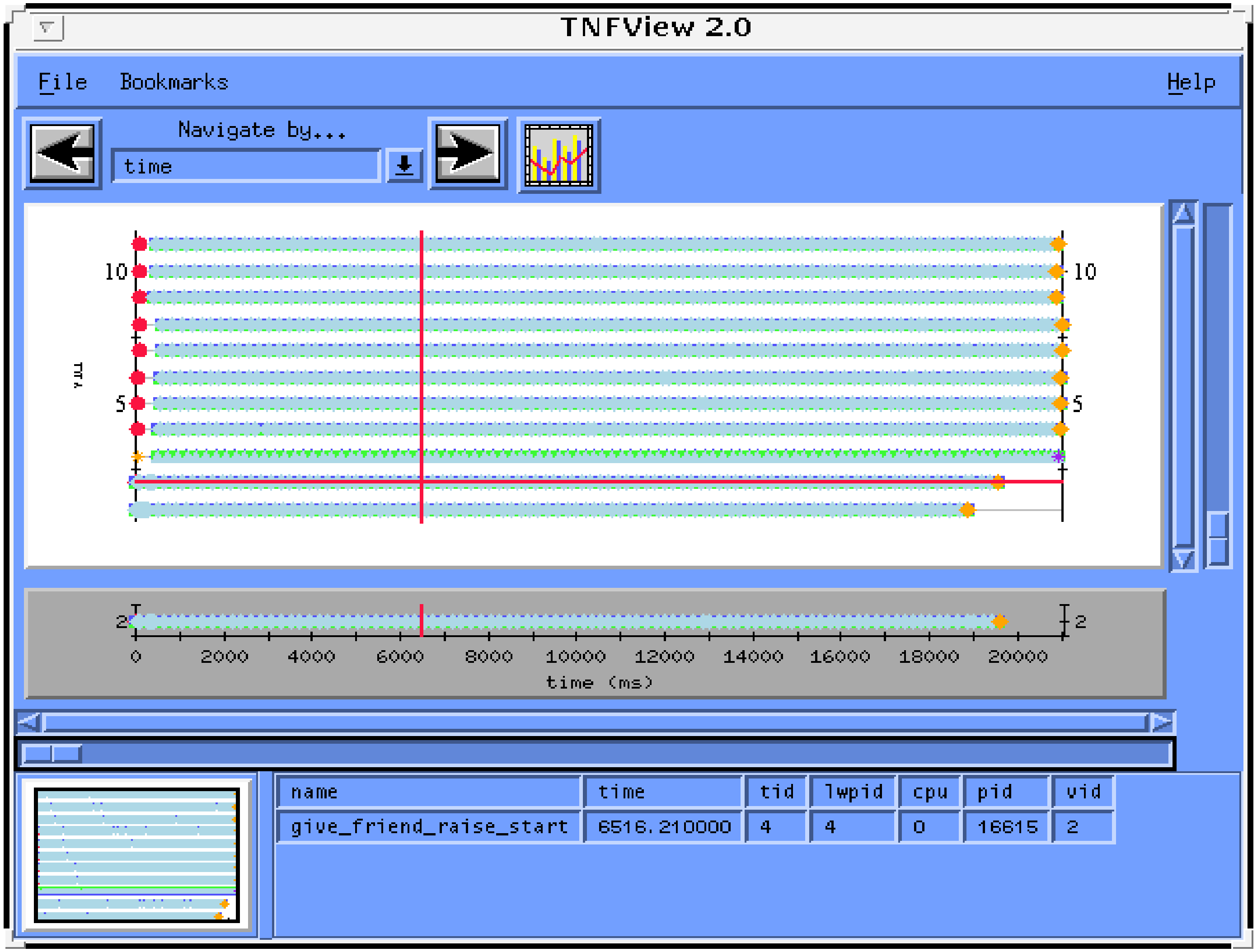
```
bil@cloudbase[87]: prex -lpthreadprobe.so tnf_list_global_lock 10 10
Target process stopped
Type "continue" to resume the target, "help" for help ...

prex> create $l1 /lgl/              Make a list of our probes...
prex> enable $l1                    And turn only them on.
prex> continue                      Or:  enable $all

LIQUIDATE=10 RAISE=10 N_PEOPLE=1000 N_FRIENDS=100 E_THREADS=1
Total raises + liquidations: 1054
Conflicts: 1084

Process info:
  elapsed time  21.0842
  CPU time      1.31127...
6.3% CPU usage

prex: target process exited
```

# Preparing TNF Files

```
bil@cloudbase[88]: tnfmerge -o /tmp/t.tnf /tmp/trace-45132
Counting events records...Found 3322 events.
Creating internal tables... Done.
Sorting event table... Done.
Processing events... Done.
Writing events to output file... Done.
bil@cloudbase[89]: tnfview /tmp/t.tnf
```

```
bil@cloudbase[89]: tnfdump /tmp/trace-45132
probe tnf_name: "give_friend_raise_middle" tnf_string: "keys lgl;file tnf_list_global_lock.c;line 157;"
probe tnf_name: "give_friend_raise_end" tnf_string: "keys lgl;file tnf_list_global_lock.c;line 159;"
probe tnf_name: "liquidate_enemies_start" tnf_string: "keys lgl;file tnf_list_global_lock.c;line 186;"
probe tnf_name: "liquidate_enemies_end" tnf_string: "keys lgl;file tnf_list_global_lock.c;line 198;"
probe tnf_name: "give_friends_raise_end" tnf_string: "keys lgl;file tnf_list_global_lock.c;line 164;"
---------------- ---------------- ----- ----- ---------- --- ------------------------ ------------------
    Elapsed (ms)     Delta (ms)   PID LWPID   TID    CPU Probe Name              Data / Description . . .
---------------- ---------------- ----- ----- ---------- --- ------------------------ ------------------
        0.000000         0.000000 16615     5         5   - give_friends_raise_start
        0.695500         0.695500 16615     5         5   - give_friend_raise_start
        0.955000         0.259500 16615     5         5   - give_friend_raise_middle
        1.447000         0.492000 16615     4         4   - give_friends_raise_start
       16.150000        14.703000 16615     5         5   - give_friend_raise_end
       16.703000         0.553000 16615     4         4   - give_friend_raise_start
       17.311000         0.608000 16615     4         4   - give_friend_raise_middle
       36.163000        18.852000 16615     4         4   - give_friend_raise_end
       36.418500         0.255500 16615     5         5   - give_friend_raise_start
```

# TNFview: Main Window



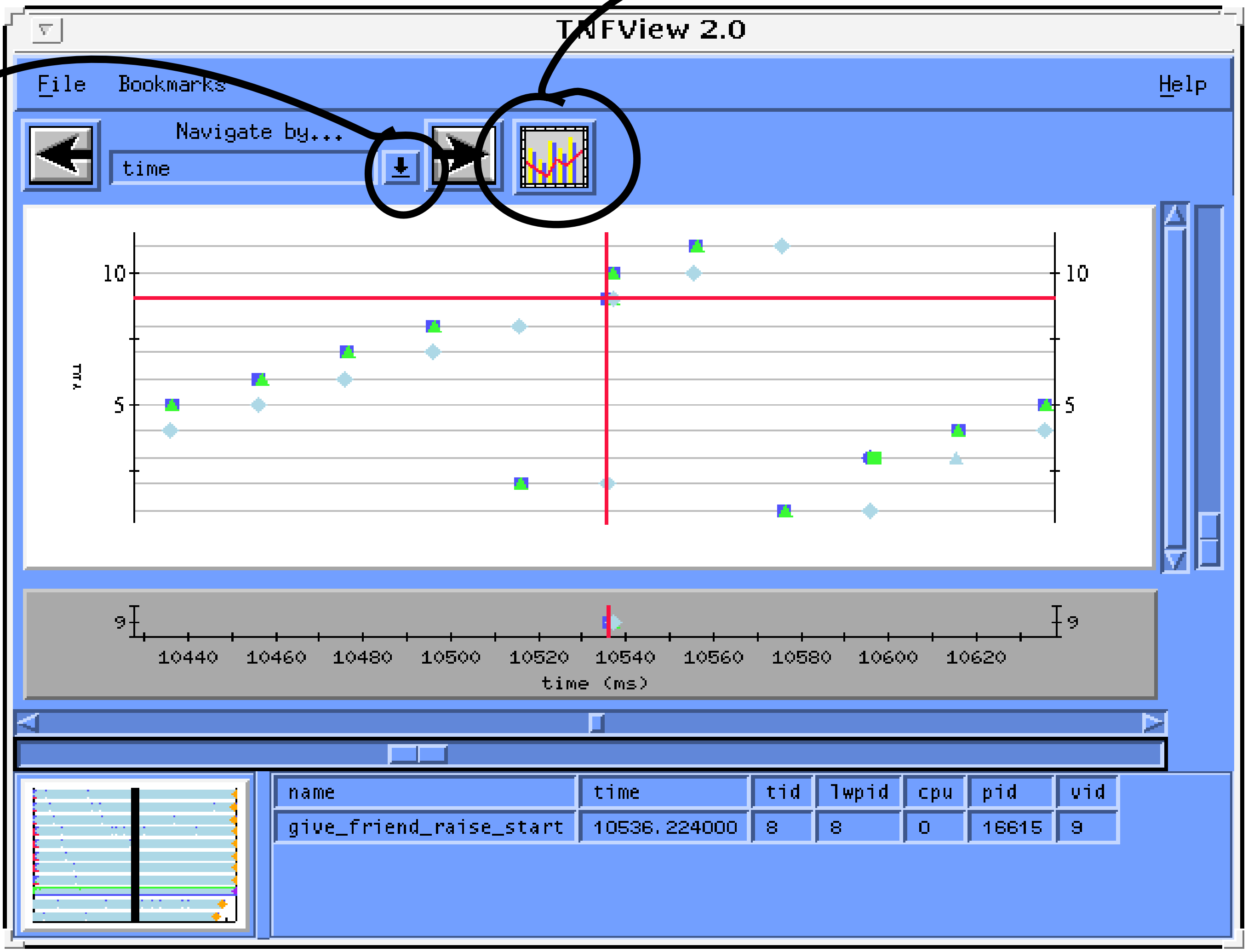
Navigate by... time

| name | time | tid | lwpid | cpu | pid | vid |
|---|---|---|---|---|---|---|
| give_friend_raise_start | 6516.210000 | 4 | 4 | 0 | 16615 | 2 |

# TNFview: Main Window, Zoomed

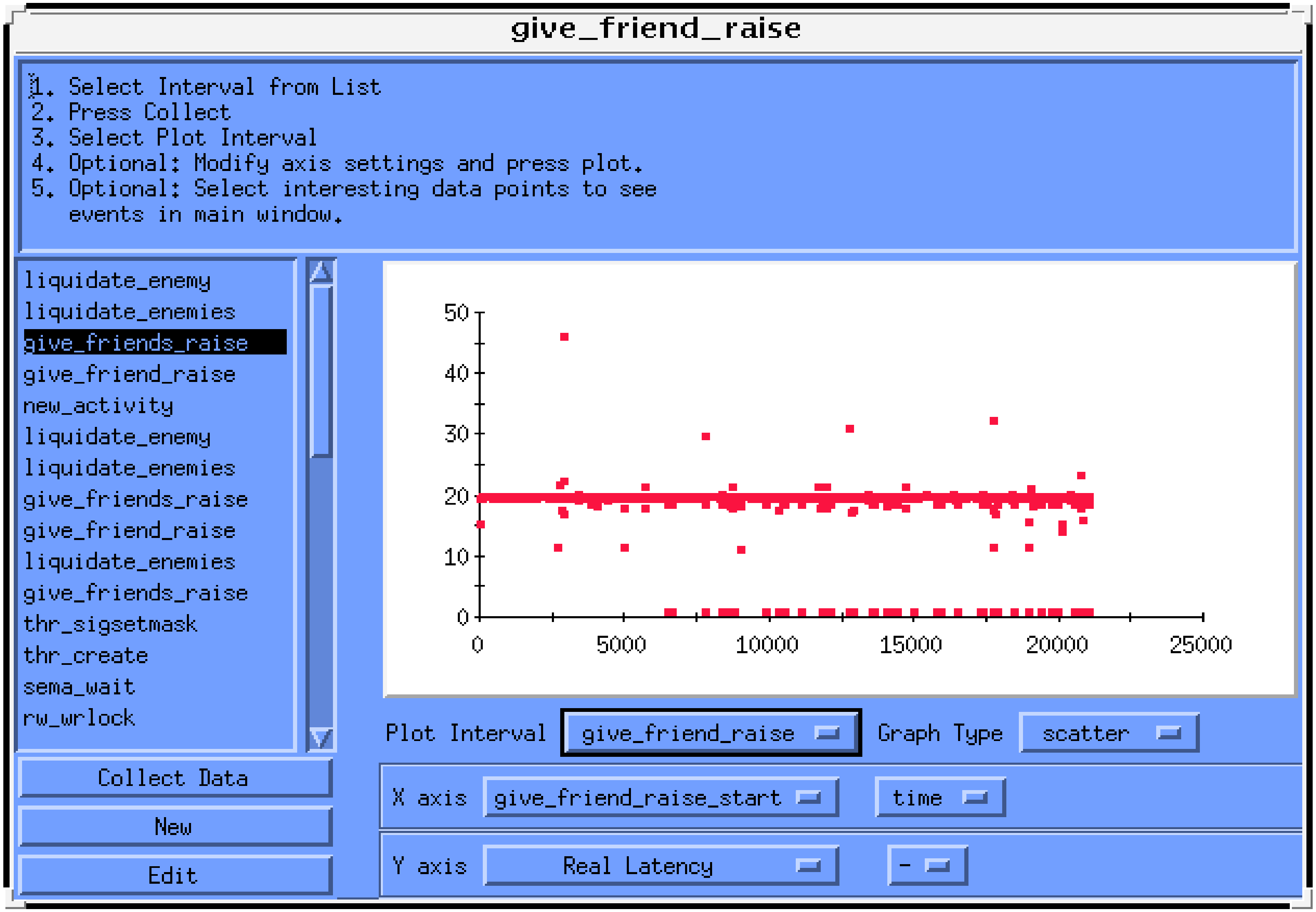
Graph

Same Probe
Same Thread



| name | time | tid | lwpid | cpu | pid | vid |
|---|---|---|---|---|---|---|
| give_friend_raise_start | 10536.224000 | 8 | 8 | 0 | 16615 | 9 |

# TNFview: Latency Graph



give_friend_raise

1. Select Interval from List
2. Press Collect
3. Select Plot Interval
4. Optional: Modify axis settings and press plot.
5. Optional: Select interesting data points to see
   events in main window.

Plot Interval: give_friend_raise

Graph Type: scatter

X axis: give_friend_raise_start, time

Y axis: Real Latency, -

# *Typical Problems*

# Diatribe on the Failings of C

- Incomplete type hierarchy
  - Forces you to use (void *) and casts
- Types are not self-identifying
  - Forces you to use casts
- Persuasive use of pointers as a datatype
  - Makes for complex code
- No garbage collection
  - You spend too much time managing memory
- Lack of an exception handling system
  - Adds complexity and an unfortunate tendency to skip over error checks
- Lack of closures
  - Encourages unrestrained use of globals

# Diatribe on the Failings of C

- Lack of a module system to define the interface
- Too close to the hardware
  - Some functions have ridiculous limitations (e.g., longjmp doesn’t restore FP registers)
  - Too much porting effort required between platforms
- Equates pointers and arrays
  - Allows truly ugly programming and gives the compiler fits.

You don’t have any choice about these problems, but it’s useful to know that there are solutions in other languages which simplify the programming and allow you to concentrate on your problem and not the programming constraints.

(In defense of C, it was not intended to be an applications programming language. As a hardware abstraction layer, it’s quite reasonable.)

# Typical Problems

- Failure to Check Return Values for Errors
- Using `errno` Without Checking the Return Value

```
Good Programmer                     Bad Programmer!

err = system_call();                system_call()
if (err)                            if (errno)
  printf(“Bug: %d”, errno);           printf(“Bug: %d”, errno);
```

- Not Joining on Nondetached Threads
- Failure to Verify that Library Calls Are MT Safe
- Falling off the Bottom of `main()`

# Typical Problems

- **Using Spin Locks**
- **Depending upon Scheduling Order**
- **Using `errno` for Threads Library Functions**
- **Not Recognizing Shared Data**
- **Assuming Bit, Byte or Word Stores are Atomic**
- **Not Blocking Signals When Using `sigwait()`**
- **Forgetting (`-D_REENTRANT` and `-lthread`)
  or `-mt` (UI threads)**
- **Forgetting (`-D__POSIX_C_SOURCE=199506L and` `-lpthread`)
  (POSIX threads)**
  - **The stub functions in `libc.so` can fool you because your program will compile and run (incorrectly)**

# Typical Problems

- **Passing Pointers to Data on the Stack to Another Thread**

```
Good Programmer                     Bad Programmer!

main()                              main()
{my_struct *s;                      {my_struct s;
...                                 ...
s = (my_struct *) malloc(...);      ...
s->data = get_data();               s.data = get_data();
pthread_create(... s, ...);         pthread_create(... &s, ...);
pthread_exit();                     pthread_exit();
}                                   }
```

- **Believing the Documentation too Much.**
  - **The Solaris 2.5 man pages will tell you POSIX semaphores are not implemented, but they actually are.**
  - **And `sem_trywait()` also sets `errno` to `EBUSY` (should be `EAGAIN`).**

# *Don’t do That!*

- **Avoid `sched_yield() / thread.yield()`**
- **Avoid suspension**
- **Don’t return status from `pthread_exit()`**
- **Don’t wait for threads: `pthread_join() / thread.join()`**
- **Avoid cancellation (`thread.stop()`)**
- **Never `fork()` without calling `exec()` immediately**
- **Never try to recover from a deadlock**
- **Don’t use spin locks (at least time the program with and w/o!)**
- **Don’t use RW locks (at least time the program with and w/o!)**
- **Never create TSD keys dynamically**
- **Never delete TSD keys**
- **Avoid `pthread_once()`**

# Threads Comparison

| POSIX Pthreads | Solaris Threads | Java Threads |
|---|---|---|
| pthread_create() | thr_create() | new Thread()<br>t.start() |
| pthread_exit() | thr_exit() | stop() |
| pthread_join() | thr_join() | join() |
| sched_yield() | thr_yield() | yield() |
| pthread_self() | thr_self() | currentThread |
| pthread_kill() | thr_kill() | interrupt() |
| pthread_sigmask() | thr_sigsetmask() | - |
| pthread_setschedparam() | thr_setprio() | |
| pthread_getschedparam() | thr_getprio() | |
| - | thr_setconcurrency() | - |
| - | thr_getconcurrency() | - |
| - | thr_suspend() | suspend() |
| - | thr_continue() | resume() |
| pthread_key_create() | thr_keycreate() | Via Subclass |
| pthread_key_delete() | - | - |
| pthread_setspecific() | thr_setspecific() | var = |
| pthread_getspecific() | thr_getspecific() | thread.var |
| pthread_once() | - | - |
| pthread_equal() | - | - |
| pthread_cancel() | - | |
| pthread_testcancel() | - | stop() |
| pthread_cleanup_push() | - | |

| | | |
|---|---|---|
| pthread_cleanup_pop() | - | via finally |
| pthread_setcanceltype() | - | - |
| pthread_setcancelstate() | - | - |
| pthread_mutex_lock() | mutex_lock() | synchronized |
| pthread_mutex_unlock() | mutex_unlock() | (implicit) |
| pthread_mutex_trylock() | mutex_trylock() | - |
| pthread_mutex_init() | mutex_init() | - |
| pthread_mutex_destroy() | mutex_destroy() | - |
| pthread_cond_wait() | cond_wait() | wait() |
| pthread_cond_timedwait() | cond_timedwait() | wait(long) |
| pthread_cond_signal() | cond_signal() | notify |
| pthread_cond_broadcast() | cond_broadcast() | notifyAll |
| pthread_cond_init() | cond_init() | - |
| pthread_cond_destroy() | cond_destroy() | - |
| - | rwlock_init() | - |
| - | rwlock_destroy() | - |
| - | rw_rdlock() | - |
| - | rw_wrlock() | - |
| - | rw_unlock() | - |
| - | rw_tryrdlock() | - |
| - | rw_trywrlock() | - |
| sem_init() POSIX 1003.4 | sema_init() | - |
| sem_destroy() POSIX 1003.4 | sema_destroy() | - |
| sem_wait() POSIX 1003.4 | sema_wait() | - |

|  |  |  |
|---|---|---|
| sem_post() POSIX 1003.4 | sema_post() | - |
| sem_trywait() POSIX 1003.4 | sema_trywait() | - |
| pthread_mutex_setprioceiling() | - | - |
| pthread_mutex_getprioceiling() | - | - |
| pthread_mutexattr_init() | - | - |
| pthread_mutexattr_destroy() | - | - |
| pthread_mutexattr_setpshared() | - | - |
| pthread_mutexattr_getpshared() | - | - |
| pthread_mutexattr_setprioceiling() | - | - |
| pthread_mutexattr_getprioceiling() | - | - |
| pthread_mutexattr_setprotocol() | - | - |
| pthread_mutexattr_getprotocol() | - | - |
| pthread_condattr_init() | - | - |
| pthread_condattr_destroy() | - | - |
| pthread_condattr_getshared() | - | - |
| pthread_condattr_setshared() | - | - |
| pthread_attr_init() | - | - |
| pthread_attr_destroy() | - | - |
| pthread_attr_getscope() | - | - |
| pthread_attr_setscope() | THR_BOUND flag | - |
| pthread_attr_getstacksize() | - | - |

**Many of the missing pieces can be constructed, e.g., the Semaphore and ConditionVar classes in Extensions.java.**

# *Information, Books, Products*

# Sources of Information

- New Man Pages in UNIX
- New Solaris Manual: *Multithreaded Programming Guide*
- Programming Courses from Sun Education (800) 422-8020 (Other vendors?)
- Threads.h++ course from RogueWave (800) 487-3217
- To order a copy of the standard, call 1-800-678-IEEE (+1 908 981-1393 from outside the U.S). Ask for POSIX.1c [formerly named POSIX.4a], the amendment to POSIX.1-1990 for threads. (130 USD when I bought mine.)

# Internet Resources

- **comp.programming.threads**
- **http://www.sun.com/workshop/threads DEAD**
  **(Pointer to POSIX doc, other threads pages, UI Threads FAQ)**
- **http://opcom.sun.ca/toolpages/tnftools.html DEAD**
  **(Solaris 2.5: TNFview, SPARC only)**
- **http://www.sun.com/developers/driver/tools/toc.html DEAD**
  **(Solaris 2.6: TNFview2, SPARC only)**
- **ftp://sunsite.unc.edu/pub/sun-info/mde/proctool DEAD**
  **(proctool, SPARC & x86)**
- **http://www.LambdaCS.com**
  **(newsgroup FAQ, example code)**
- **www.alphaworks.ibm.com/formula/jinsight**
  **Performance Analysis Tool for Windows NT**
- **http://www.sun.com/solaris/java/wp-java/4.html**
  **http://www.sun.com/software/whitepapers.html**

# Books

- ***Threads Primer*, Bil Lewis & Daniel J Berg. SunSoft Press (1995)**
    - **Heavy concentration on the foundations and programming issues, small examples. Mainly UI & POSIX; comparisons to OS/2, Win32.**
- ***Programming with Threads*, Devang Shah, Steve Kleiman, & Bart Smaalders. SunSoft Press (1996)**
    - **It covers POSIX threads, concentrating on the Solaris implementation. It has a small, but adequate introduction, then concentrates on more advanced programming issues. The examples are good because they are very realistic and show you what to expect. They are bad because they are very realistic and obscure the main points in the text.**
- ***Thread Time*, Scott J. Norton, Mark D. Dipasquale, Prentice Hall (Nov, 1996) ISBN 0-13-190067-6**
    - **Describes POSIX threads with concentration on the HP-UX implementation. Excellent introduction, computer science descriptions, and standards discussion.**
- ***Programming with POSIX Threads*, Dave Butenhof; Addison Wesley, May `97 (380 pages, source on web).**
    - **Concentrates more on architecture than any specific implementation of POSIX threads. Lucid exposition of concepts and discussion of standards from one of the guys on the committee.**

- *Pthreads Programming,* Bradford Nichols, &c., O'Reilly & Associates, Inc (1996)
    - Concentrates on the Digital implementation of POSIX. It gives a good explanation of the concepts, but is a little too condensed to do them justice. Includes a major section comparing the final standard to draft 4, DCE.
- *Programming with UNIX Threads*, Charles J Northrup, Wiley (March, 1996)
    - It covers the UI threads library, focusing on the UNIXware implementation. The presentation is oriented around the API and contains numerous examples.
- *Effective Multithreading in OS/2*, Len Dorfman, Marc J Neuberger. McGraw-Hill (1995)
    - It gives a brief introduction, then focuses the rest of the discussion on the API and examples. It covers the OS/2 API.
- *Multithreaded Programming with Windows NT*, Thuan Q Pham & Pankaj K Garg. Prentice Hall (1996)
    - It focuses on the Win32 library, and gives some comparison with other libraries. It describes concepts and designs well, but lacks many of the practical details and problems.
- *Multithreading Applications in Win32*, Jim Beveridge and Robert Wiener; Addison-Wesley, Jan `97 (368 pages, source on diskette).
    - It describes Win32 threads (NT and Win95). Includes some comparison to POSIX. Excellent discussion of the practical aspects of programming Win32. Many insightful comments on both the good parts and the more problematic parts.

- *Multithreading Programming Techniques*, Shashi Prasad; McGraw-Hill, Jan. `97 (410 pages, source on diskette and web).
  - Describes and contrasts the multithreading libraries of POSIX, UI, Mach, Win32, and OS/2. Each library has its own chapters and its own code examples. This means that the introduction and presentation of concepts is lighter, but the examples are ported across the different platforms, making this a good reference for porting.
- *Concurrent Programming in Java*, Doug Lea; Addison Wesley, `97 (240 pages, source on web).
  - Describes how to write multithreaded programs in Java, using design patterns. Well-written from a computer science point-of-view, though perhaps overwhelming for the hacker-oriented. Familiarity with design patterns is a necessity.
- *Java Threads*, Scott Oaks and Henry Wong; O'Reilly, 97 (252 pages, source on web).
  - Describes how to write multithreaded programs in Java in a more conventional, programmer-oriented style. Explanations are clear, though often simplistic. The programs illustrate the points well, yet tend to gloss over problem areas in Java.
- *Multithreaded Programming with Pthreads*, Bil Lewis & Daniel J Berg. SunSoft Press (1997)
  - Heavy concentration on the foundations and programming issues, small examples. Mainly POSIX; comparisons to OS/2, Win32, Java. This course is based on this book (and vice-versa).

- *Multithreaded Programming with Java*, Bil Lewis & Daniel J Berg. SunSoft Press (1999)
    - Heavy concentration on the foundations and programming issues, small examples. Mainly Java; comparisons to Pthreads, and Win32. This course is based on this book (and vice-versa).
- *Concurrent Programming: The Java Programming Language*, Stephan J. Hartley; Oxford University (1998)
    - Designed as a classroom text. It spends a lot of time on understanding the underlying issues and alternative choices that could have been made at the expense of how the API should be used.
- *Designing High-Powered Os/2 Warp Applications; The Anatomy of Multithreaded Programs*, David E. Reich; John Wiley & Sons, April '95 (??) (336 pages).
    - I have not seen this book yet.  (Looked for it, not in my bookstore.)

- ***Multiprocessor System Architectures*, Ben Catanzaro. SunSoft Press (1994)**
  - **Survey of papers and documentation on SPARC systems. Good hardware descriptions for experienced hardware types.**
- ***Solaris Multithreaded Programming Guide*, SunSoft Press (1995)**
  - **The SunSoft documentation for Solaris. The 1995 version (Solaris 2.4) covers the API for UI threads and has a good explanation of MT in general. The 1996 version (Solaris 2.5) covers POSIX threads, but is a step backwards in quality.**
- ***Sun Performance and Tuning*, Adrian Crockcroft. SunSoft Press (1995, 1998)**
  - **Describes system performance issues for Solaris systems (primarily SPARC systems).**
- ***Computer Architecture: A Quantitative Approach*, John L Hennessy & David A Patterson. Morgan Kaufmann (1996)**
  - **A *superb* book for the advanced student. Covers all aspects of CPU and memory system design, plus some broader system design issues (storage devices, external busses, network interfaces. Great section on multiprocessors (nothing on MT programming).**
- ***Scalable Shared-Memory Multiprocessing*, Daniel E Lenoski & Wolf-Dietrich Weber; Morgan Kaufman, Inc., `95 (340 pages).**
  - **This takes up in great detail what Hennessy & Patterson describe in mere passing detail. It describes the state of SMP research as it led to the Stanford DASH machine, and now the SGI Origin series and HAL Mercury. Superb research and exposition!**

# MT Products Available

- Dakota Scientific Software
    - (800) 641-8851 (605) 394-885 sales@scisoft.com
    - Math libraries
- ~~Pure~~ ~~PureAtria~~ Rational Software
    - (408) 863-9900 www.rational.com
    - MT tools: leak detector, performance monitor
- Rogue Wave
    - (800) 487-3217 www.roguewave.com
    - MT-safe C++ class libraries, thread.h++
- ObjectSpace
    - (214) 934-2496 www.objectspace.com
    - MT-safe C++ class libraries
- SMP Malloc Implementations and Papers
    - Hoard http://www.cs.utexas.edu/users/emery
    - SmartHeap http://www.microquill.com/smp/default.htm
    - List of mallocshttp://www.cs.colorado.edu/~zorn/Malloc.html
    - List of papers http://www.cs.utexas.edu/users/oops

- **Centerline**
  - **MT tools: debugger, else?**
  - **(415) 943-2114    www.centerline.com**
- **SunSoft**
  - **MT tools: debugger, performance monitor, leak detector, static analyzer**
  - **(800) SUN-SOFT (512) 434-1511    www.sun.com**
- **Transarc**
  - **Pthreads draft 4 (via DCE) for SunOS 4.1.3**
  - **(412) 338-4400    www.transarc.com**
- **Geodesic Systems**
  - **MT garbage collector**
  - **(800) 360-8388 (312) 728-7196    www.geodesic.com**
- **GNU (Cygnus)**
  - **MT debugger by ‘98?**
  - **(800) 294-6871 (415) 903-1400    www.cygnus.com**
- **THE ADAPTIVE COMMUNICATION ENVIRONMENT (ACE)**
  - **C++ toolkit for communications and threads**
  - **www.cs.wustl.edu/~schmidt/ACE.html**

# Summary

- Threads provide a more natural programming paradigm for many tasks.
- Threads allow the programmer to take advantage of overlapping I/O on uniprocessor machines.
- Threads allow the programmer to take full advantage of MP machines.
- Threads are very nice for running both realtime tasks, and doing complex signal handling.
- Threads will not solve every problem, but if they solve *your* problem, that’s all that counts!

## Threads are cool!

# Final Exam

1. What kinds of programs (a) shouldn’t use threads, (b) should use multiple processes, (c) should use threads?
2. Explain why multiple threads are cheaper and faster than multiple processes.
3. Prove that using a mutex hierarchy makes deadlocks impossible.
4. Why is it impossible to have a 100% correct FIFO mutex using `ldstub`?
5. What are the circumstances under which you would use a spin lock, and explain why.
6. Under what circumstances can you not lock the use of shared data? Explain why.
7. Why do you need to wait for cancelled threads? How long do you have to wait for them?
8. Explain how to do deadlock recovery.
9. Build a multithreaded web server.

# Evaluation

1.0 How was the seminar?
(Overhead slides, Room, Answering questions, Depth of Material, Quality of presentation, Sense of humor, etc.)

2.0 What one thing could I have done that would have made the seminar better for you?

3.0 Other Comments...

# Your MT Plans

If you have projects in mind that you may be writing as MT programs, and you don’t object to discussing them, I would like to hear about them. I will not identify your project to anyone else, but I will use the general information to describe the kinds of things folks are doing, and the kinds of issues they are facing.

So, if you can...

- What kinds of MT projects are you looking at?
- Is MP performance a significant aspect?
- Have you begun them?
- Have you faced any “interesting” MT problems?
- Email or Phone if you wouldn’t mind me following up some day in the future.

# [Intentionally Left Blank]

# Di Allra Sista Detaljerna

## [Intentionally Right Blank]

*Individual Advice*

**Collect Key Cards, Badges, Evaluations**

**Kvällsmackor**

**Consulting Contracts**

**Autograph Books**

**(What else did I forget?)**