Thread (2)

Sunate Hwang
Kookmin University

1) Solaris Multithreaded Kernel

OPERATING SYSTEM ISSUES

Solaris Multithreaded Kernel

- The Solaris kernel itself is implemented using threads.

- Kernel 의 모든 schedulable한 entity들은 모두 thread로 구현되었기 때문에 SMP support, real time scheduling 과 kernel을 preemptable하게 만드는 것이 보다 쉽게 구현되었다.
  - LWPs are built on top of kernel threads.
  - Interrupts are built with kernel threads.
  - Preemption of kernel threads works much the same way as preemption of user-level threads.

Concurrency vs. Parallelism

- Concurrency means that two or more threads can be in the middle of executing code at the same time.

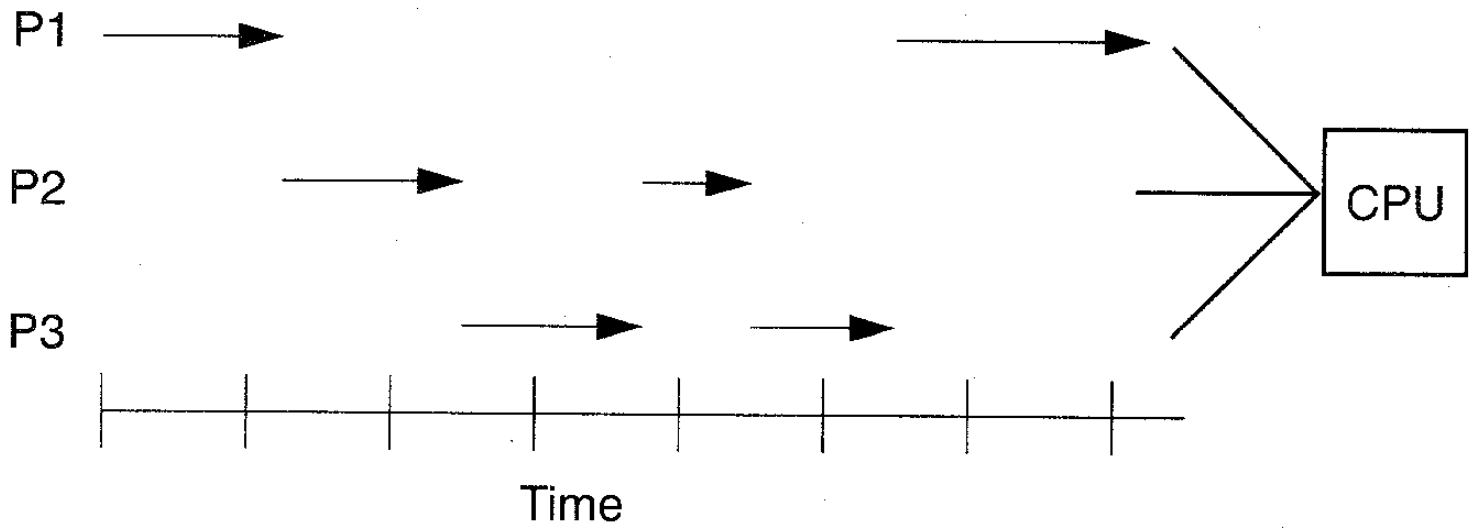


Figure 6-1 Three Processes Running Concurrently on One CPU

- Parallelism means that two or more threads actually run at the same time on different CPUs.

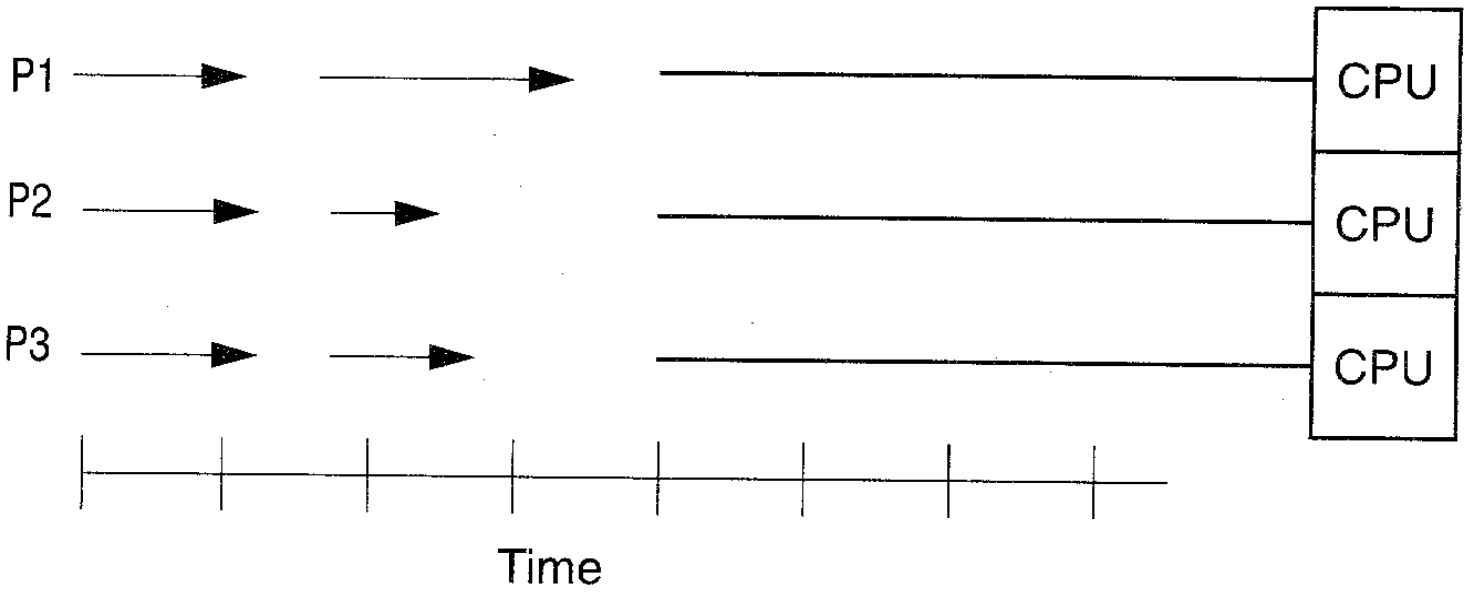


Figure 6-2 Three Processes Running in Parallel on Three CPUs

1. **In the first case: only one process can be in the midst of executing a system call at any one time. (like SunOS 4.1.3, the critical section is very large)**

2. **In the second case: locks are put around each major section of code in the kernel, so several processes can be in the midst of executing system calls, as long as the calls are to different portions of the kernel.**

3. **In the third case: the granularity of the locks has been reduced to the point that many threads can be in the midst of executing the same system calls. (like Solaris 2, there are lots of little critical sections)**

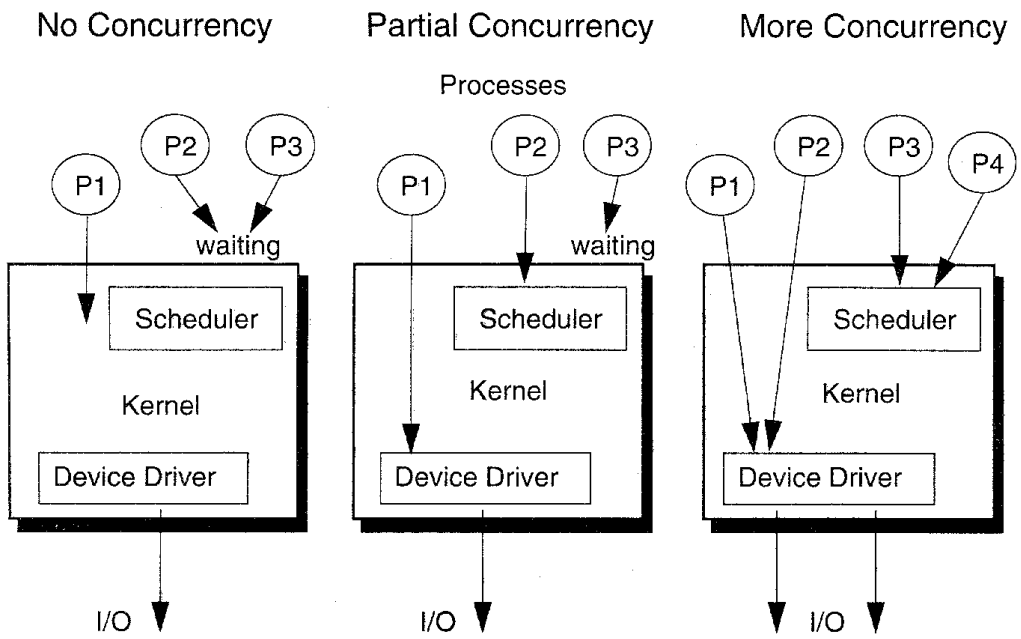


Figure 6-3 Concurrency Within the Kernel

Solaris Symmetric Multiprocessing

- SMP는 단지 모든 프로세서가 동일하고 설계자에 의해 특정 기능이 부여 되었다는 것을 의미한다. 이 기능 중에는 shared memory, kernel code 실행 능력, interrupt 처리 등이 있다.

- Figure 6-3의 첫번째 경우는 kernel 전체에 lock이 걸리므로 한 순간에 한 CPU만 kernel을 실행할 수 있다.

- Solaris 2는 uniprocessor system이나 tightly coupled, shared memory multiprocessor system을 위해 설계되었고 kernel은 모든 processor가 동일하다고 가정한다. (실제로는 load가 동일하지 않을지라도)

Kernel Scheduling

- Fixed priority real-time processes
  - system의 heuristic priority adjustment algorithm에 영향을 입지않는다.
- User LWP priority manipulation
  - priocntl()
- High-resolution timers
  - nanosecond granularity
- Completely preemptive scheduling
  - Any thread can be interrupted at almost any point.
- Process priority inheritance
  - To avoid priority inversion ...
- Deterministic and guaranteed dispatch latency
  - provides deterministic scheduling response and guarantees various dispatch l atencies on different hardware platforms.

2) Are Libraries Safe?

OPERATING SYSTEM ISSUES

MT Safety

- A function must lock any shared data it uses, it must use the correct definition of errno, and it must only call MT-safe functions.
  - Make it fast
  - Retain UNIX semantics
  - make it MT safe

- 예 1: getc()
  - Macro이므로 매우 빠름
  - 그러나 MT-safe를 위해서 mutex lock이 필요 - 매우 느려짐
  - fast, MT-unsafe version인 get_unlock()을 따로 제공.

- 예 2: getctime()
  - puts its data into a predefined location
  - semantic을 변경하지 않고 MT-safe하게 할 방법이 없다. (like errno)
  - getctime_r()이라는 새로운 함수 제공

Table 6-1 *Categories of MT Library Calls*

Category	Meaning
MT safe	A function may be called concurrently from different threads.
MT hot	An MT safe function that is also "fast" (perhaps it spawns threads, perhaps it uses no global data, perhaps it just uses a tiny bit of global data, reducing locking contention).
MT unsafe	A function that is legal in an MT program but cannot be called concurrently.
Alternative	A function that is MT unsafe, but there is an MT safe equivalent function that you can use.
MT illegal	A function that wasn't even compiled with `-D_REENTRANT` and should not be used from any thread other than `thr_main()`.

Async Safety

□만약에 malloc()중에 signal이 들어와서 signal handler 가 실행 되는데 그 안에서 또 malloc()을 호출하면

- **Deadlock!**

□왜냐하면 malloc()은 global variable을 lock하기 때문에

- **Aync unsafe**

□그러므로 signal handler 대신에 sigwait()를 사용하라.

3) New Semantics for System Calls

OPERATING SYSTEM ISSUES

Forking New Processes

- fork()는 모든 thread, LWP를 포함하여 copy된 child process 생성
- fork1() - calling thread와 LWP만 duplicate
  - 하지만 바로 exec()하는 것이 좋다.
  - 그렇지 않을 경우 매우 조심해야 되는데 존재하지 않는 thread (duplicate 되지 않았으므로)가 가지고 있는 lock을 건드릴 수도 있기 때문이다.
  - 이와 같은 상황은 단지 printf를 호출하는 것도 포함한다.

- POSIX의 fork()는 Solaris의 fork1()과 같고
- Solaris의 fork()에 해당되는 “fork all”은 POSIX에는 없다.

Executing a New Program

- exec()는 이전의 것과 똑같다.
  - Multithreaded program에서 exec()를 호출하면 모든 thread 와 LWP가 모두 terminated된다.

The New System Call sigwait(2)

- Both Solaris and POSIX define a new system call that is designed to handle signals in a new fashion.

- It will block until one of those signals is sent to the pro cess, then it will return with that signal number.

- Unlike signal handler, the code you write need not be async safe!

SYNCHRONIZATION ISSUES

Memory Model

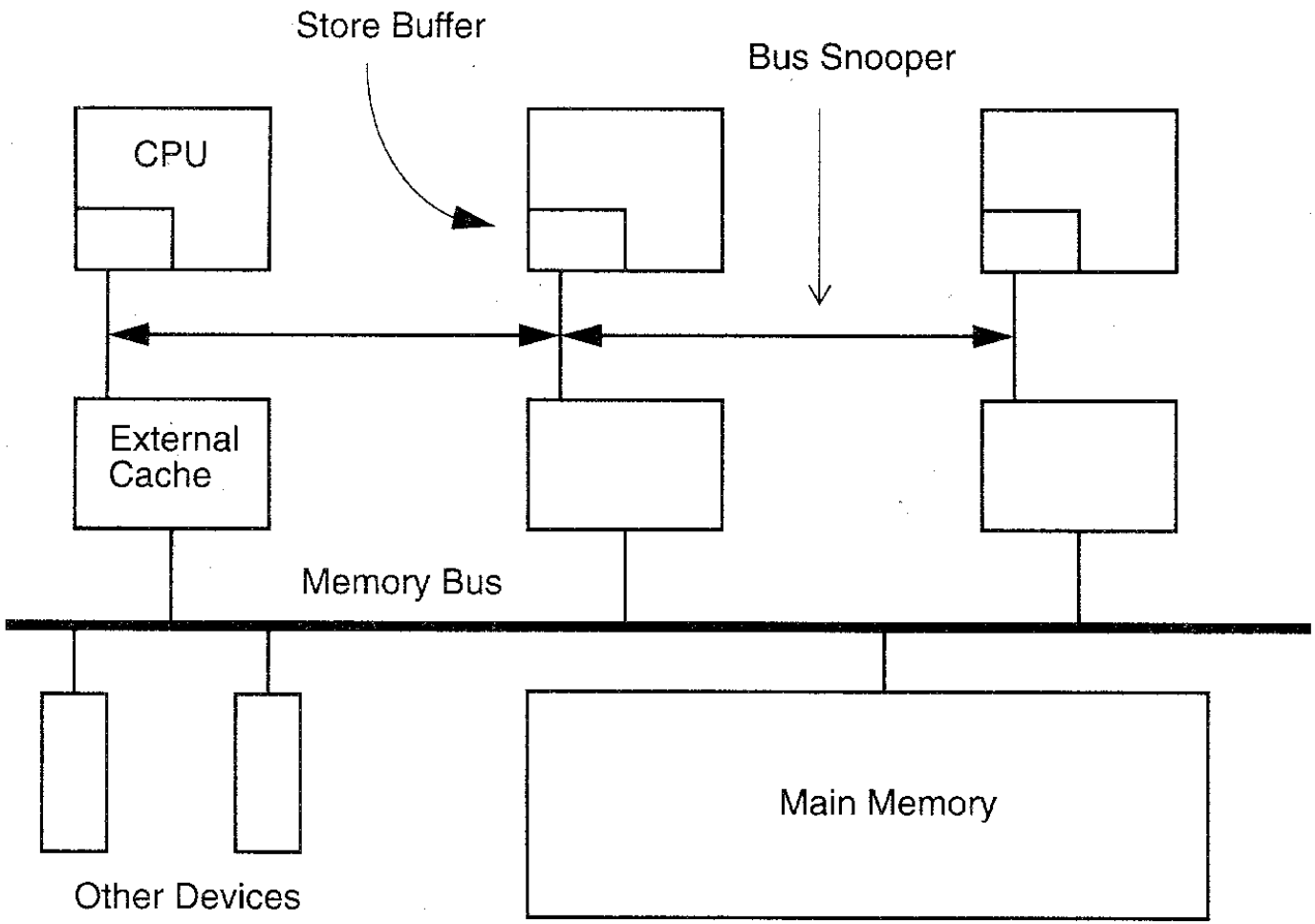


Figure 5-1 A Typical Design for an SMP Machine

- write X;
- write Y;
- 가 모든 다른 processor(CPU)에 그 순서대로 비친다는 보장이 없다.

→ all use of shared data must be locked.

- write한 것이 바로 cache에 저장되지 않고 CPU 내부의 store buffer에 머무를 수 있다. 즉 다른 CPU는 memory (cache)의 내용이 바뀌었음을 아직 모를 수 가 있다. 따라서 synchronization variable을 사용할 때는 “flush” 명령을 실행 해야 한다.

Critical Section

- A critical section is a section of code that must be allowed to complete atomically with no interruption that affects its successful completion or the data it is working on.

- Critical section을 처리 중인 thread가 processor를 놓을 수도 있지만 그사이에 다른 thread가 critical section에 영향을 줄 수 는 없다.

- **Lock Your Global Variables! (including Static variables)**

SYNCHRONIZATION VARIABLES

Mutexes

- Simplest and most primitive
- 처음으로 mutex_lock()을 호출한 thread가 lock을 얻고 그 뒤로 호출한 thread들은 sleep한다. Owner thread가 unlock하면 sleeper 중에 하나를 깨워서 runnable로 만든다. 그러나 이 thread가 반드시 lock을 얻으리라는 보장은 없다.

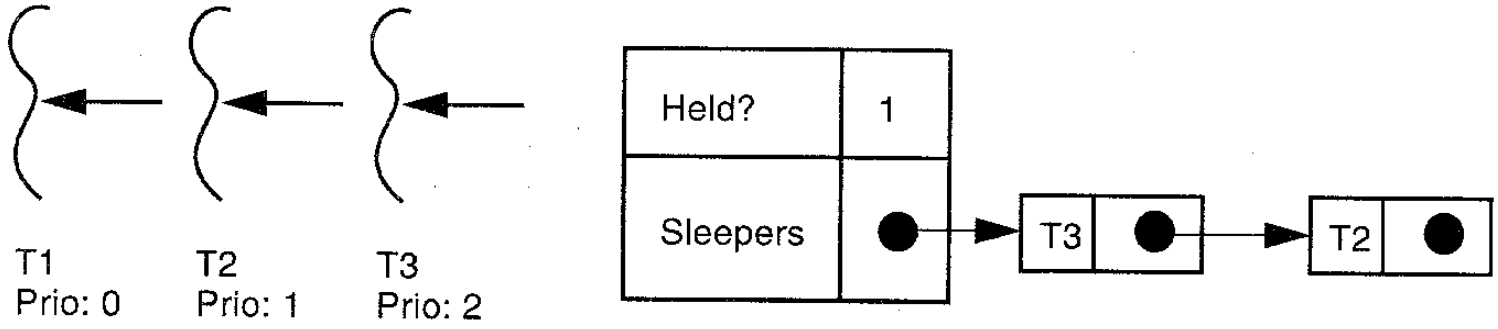


Figure 5-2 A Mutex with Several Threads Sleeping on It

전형적인 **mutex**의 이용

Thread 1	**Thread 2**
mutex_lock(&m) global++ mutex_inlock(&m)	mutex_lock(&m) local = global mutex_inlock(&m)

- mutex_trylock()
  - nonblocking library call, return 0 or EBUSY

Reader/Writer Locks

- Sometimes you will find yourself with a shared data structure that gets read often, but written only seldom.

- They are more expensive than mutexes.

- rw_rdlock()
- rw_wrlock()

- 첫 reader가 일고 있는 동안 후속 reader들도 lock을 얻고 읽을 수 있다. 이때 writer가 rw_wrlock()을 호출하면 이는 모든 read가 끝날 때 까지 sleep 한다. 후속 writer들도 모두 sleep 한다. write들이 앞에 존재하므로 후속 reader들도 sleep 한다.
- 즉 curretn action이 진행 중이면 이는 우선권이 있고 writer가 더 우선권이 있다.

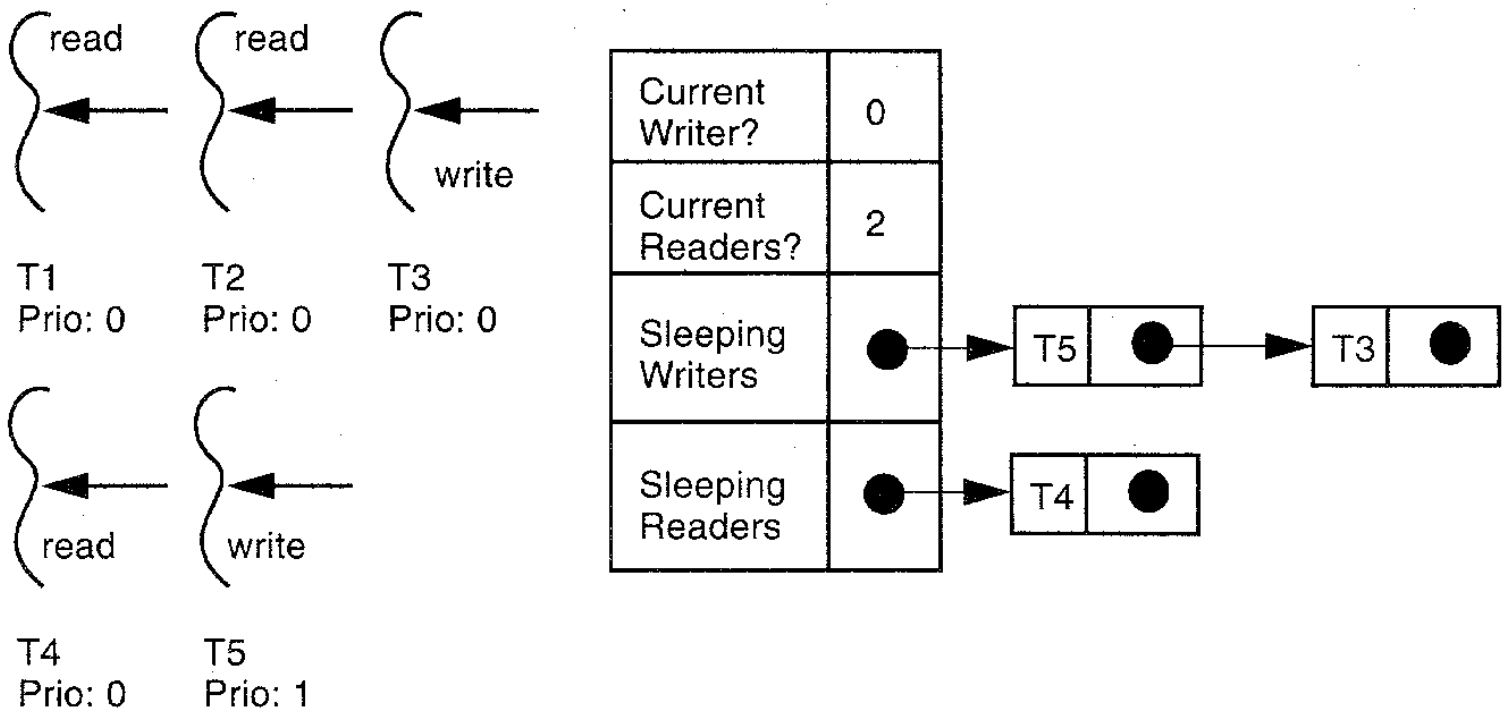


Figure 5-3 How Reader/Writer Locks Work

Condition Variable

- complex situations - 즉 어떤 특정 조건에서만 한 thread를 실행 시키고 싶을 때.

- Condition variable creates a safe environment for you to test your condition, sleep on it when false, and be awakened when it might have become true.

T1	T2
Acquire Mutex Test Condition FALSE! *Release Mutex* Sleep on CV	
	Acquire Mutex Change Condition Signal T1 Release Mutex
Reacquire Mutex Retest Condition SUCCESS! Do Thing, Release Mutex	

Figure 5-4 Classic Operation of Condition Variables

Semaphores

- **sema_post**(): increments the semaphore
- **sema_wait**(): attempts to decrement it. If semaphore > 0, the operation succeeds; if not, then the calling thr ead must go to sleep until a different thread increase it

- **EINTR**: the semaphore was interrupted by a signal or a call to fork() and it did not successfully decrement.

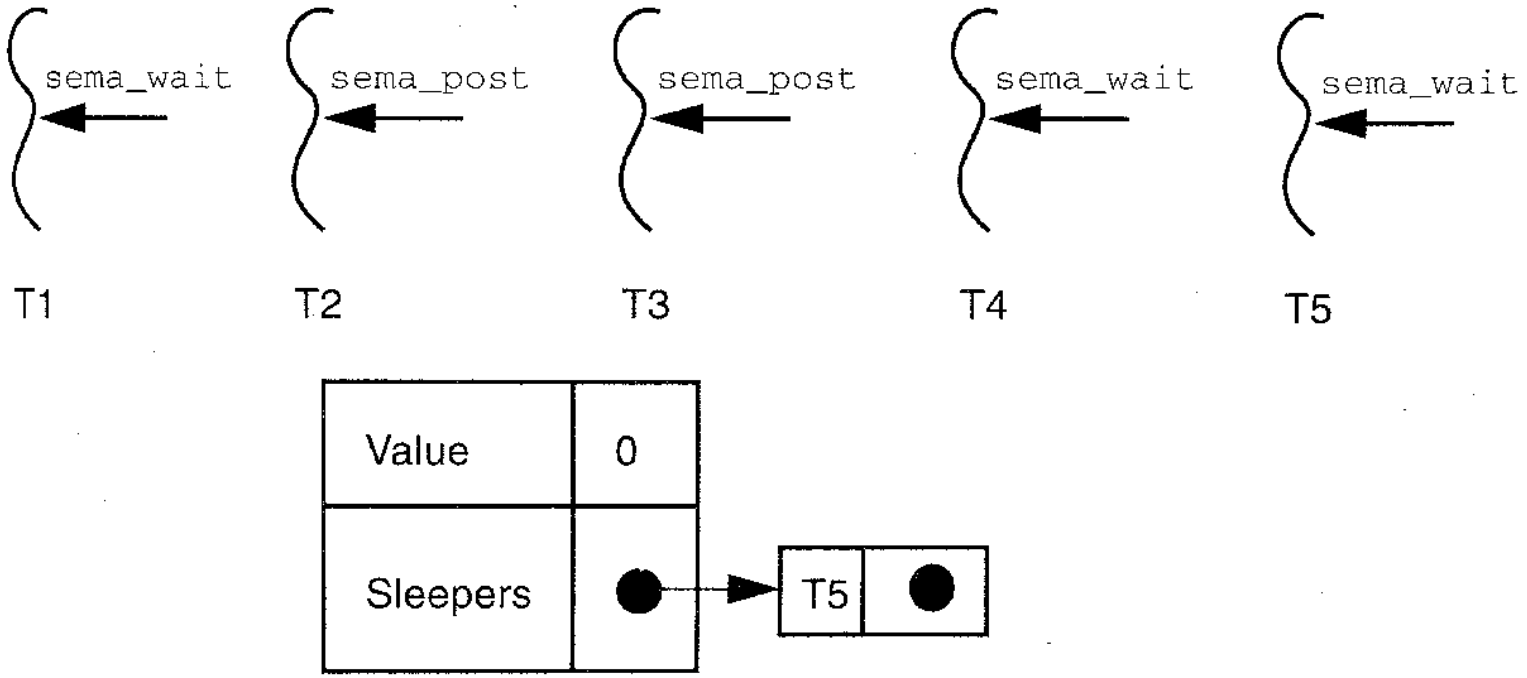


Figure 5-6 How a Semaphore Operates

```
while (sema_wait(&s) == EINTR) { <probably do nothing> }
 do_thing();                    /* NOW the semaphore has been decremented! */
```

- semaphore는 synchronization variable 중 유일하게 Async Safety이다. 즉 signal handler에서 호출될 수 있다.

Spin Locks

- mutex blocking time보다 훨씬 짧은 시간 동안만 mutex lock이 필요하다면?
  - (예: mutex blocking on SS10/41 – 48us)

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

Adaptive Locks

- Solaris kernel에서는 사용되고 있으나 Solaris나 POSIX에서는 허용하지 않고 있다.
- 만약에 mutex owner가 running이면 spin을 하고 아니면 spin 대신에 blocking한다.
- 하지만 user-level threads library에서는 어떤 thread가 mutex를 가지고 있는지 알 수 있는 방법이 없고, 할 수 있더라도 system call이 필요할 것이다. 즉 그냥 blocking하는 것보다 더 비싸진다.

Cross-process Synchronization Variables

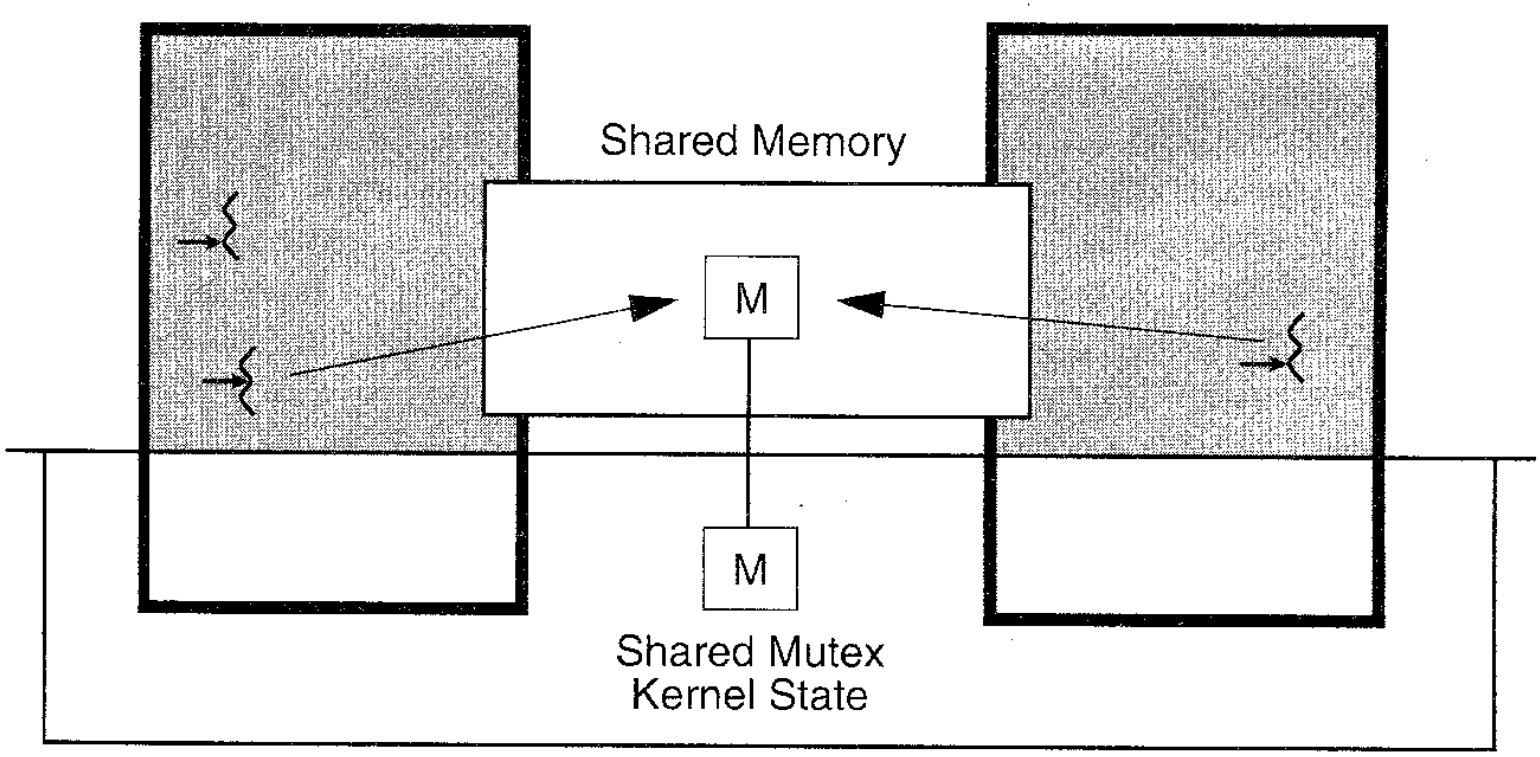


Figure 5-7 Synchronization Variables in Shared Memory

- Both processes must know about the synchronization variable, and (exactly) one of them must initialize it to be cross-process. Then, both processes (or possibly more) can use it as a normal synchronization variable.

- Synchronization variables can also be placed in files a nd have lifetimes beyond that of the creating process.

Synchronization Variable Initialization and Destruction

- initialization은 단 한번만 한다.

- Dynamically allocated된 synchronization variable은 free하기 전에 반드시 destroying해야 한다. 즉 어떤 thread도 destroy된 synchronization variable을 access하는 것을 확실히 해야 한다.

SYNCHRONIZATION PROBLEMS

Deadlocks

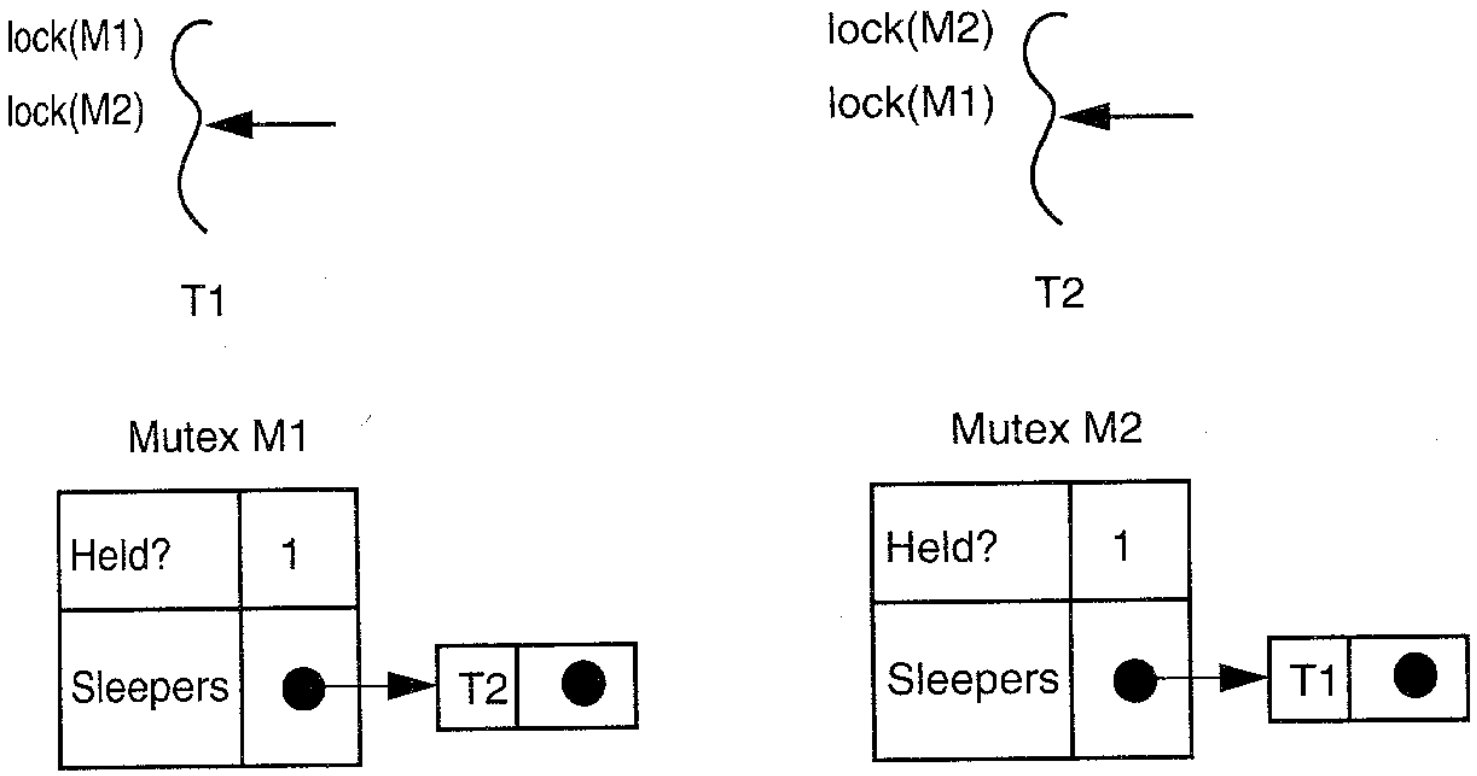


Figure 5-8 A Typical Deadlock

- 2개 또는 그 이상의 thread가 chain을 형성할 수 있다.
- recursive하게도 발생한다. 즉 Async Safety가 아닌 synchronization variable을 signal handler에서 사용할 경우 ...

- Deadlock avoid
  - 항상 같은 순서로 lock을 요구한다.
  - 항상 같은 순서를 유지할 수 없으면 trylock function을 이용하여 모든 lock을 얻었는지를 본다. 모든 lock을 얻지 못하면 모두 release하고 나중에 재 시도한다.

Race Condition

- 보통 shared data에 정상적인 locking protection을 잊었을 때 발생
- 하지만 ...

Thread 1

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

Thread 2

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

Priority Inversion

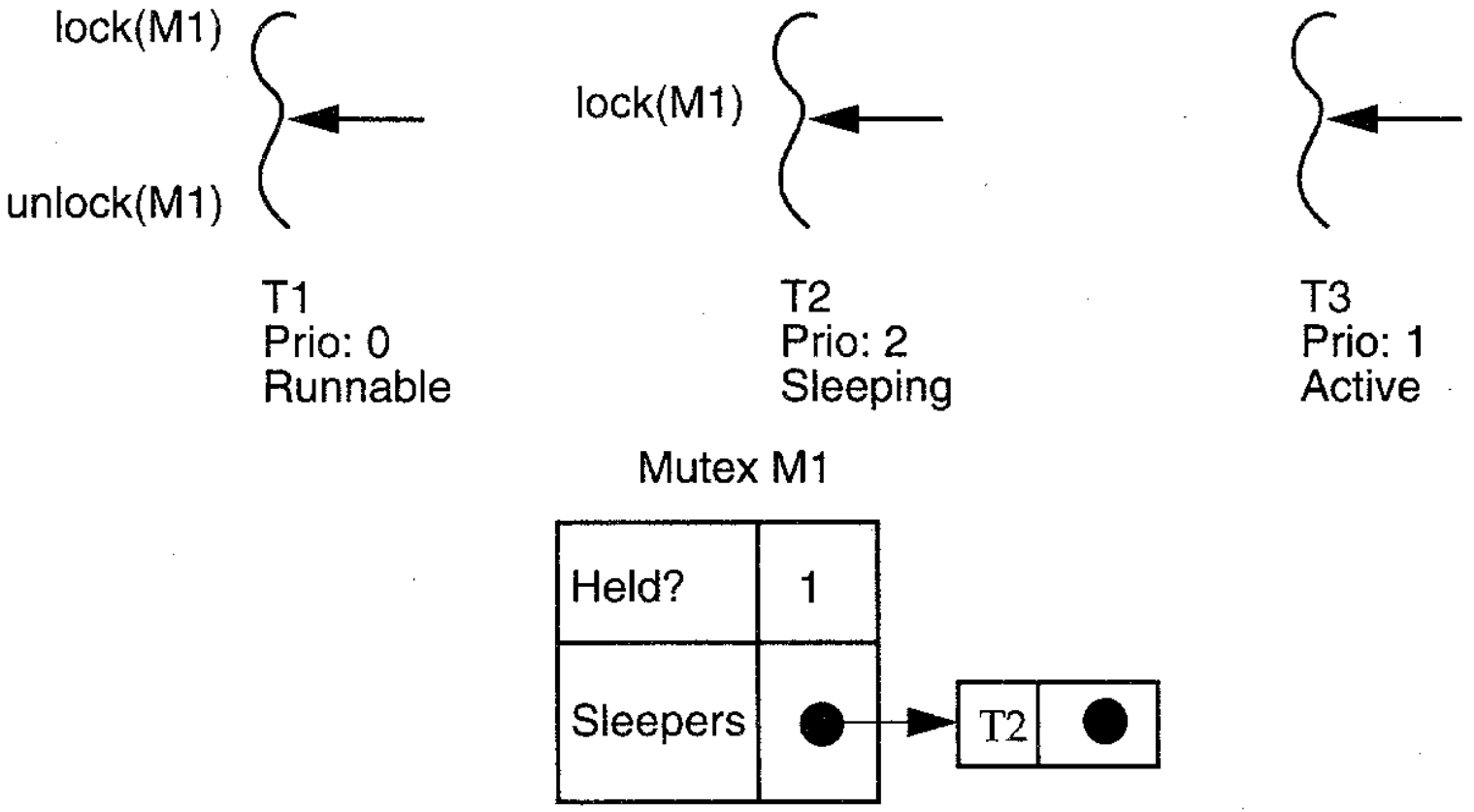
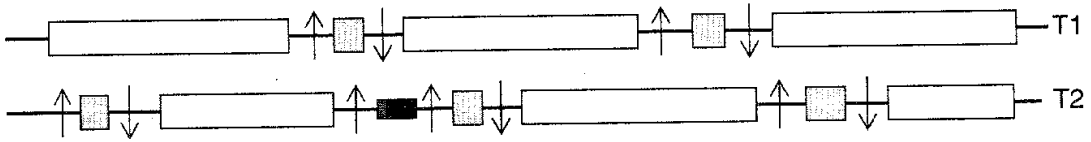


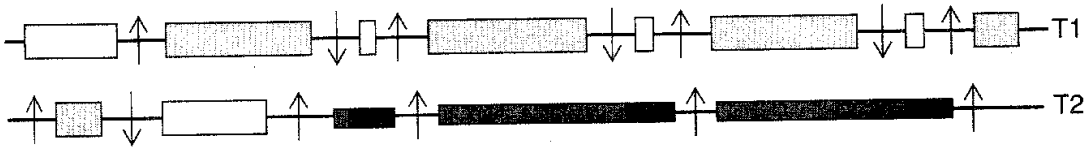
Figure 5-9 Priority Inversion

T2가 T3 보다 priority가 높지만 T1 때문에 먼저 실행될 수 없다.

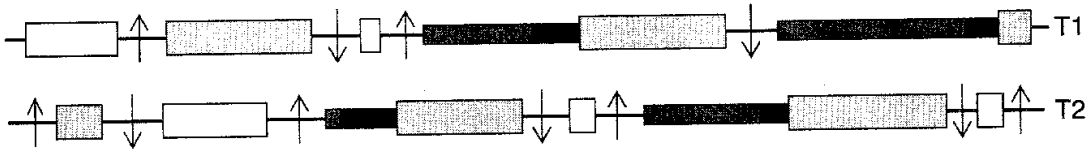
FIFO Mutexes



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.

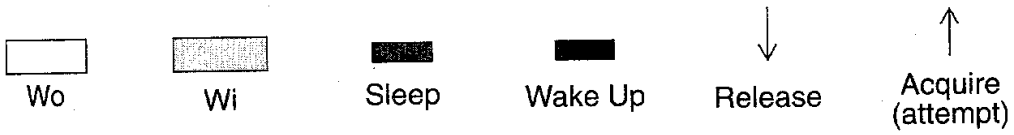


Figure 5-10 When FIFO Mutexes Are Valuable

THREAD-SPECIFIC DATA

- globally accessible to any function, yet still unique to the thread

- TSD provides this kind of global data by means of a set of function calls.
- Essentially, this is done by creating an array of “key” offsets to “value” cells, attached to each thread structure.

- 통상 TSD는 structure, array등과 같은 것의 pointer값을 저장한다.

  (malloc() 필요)

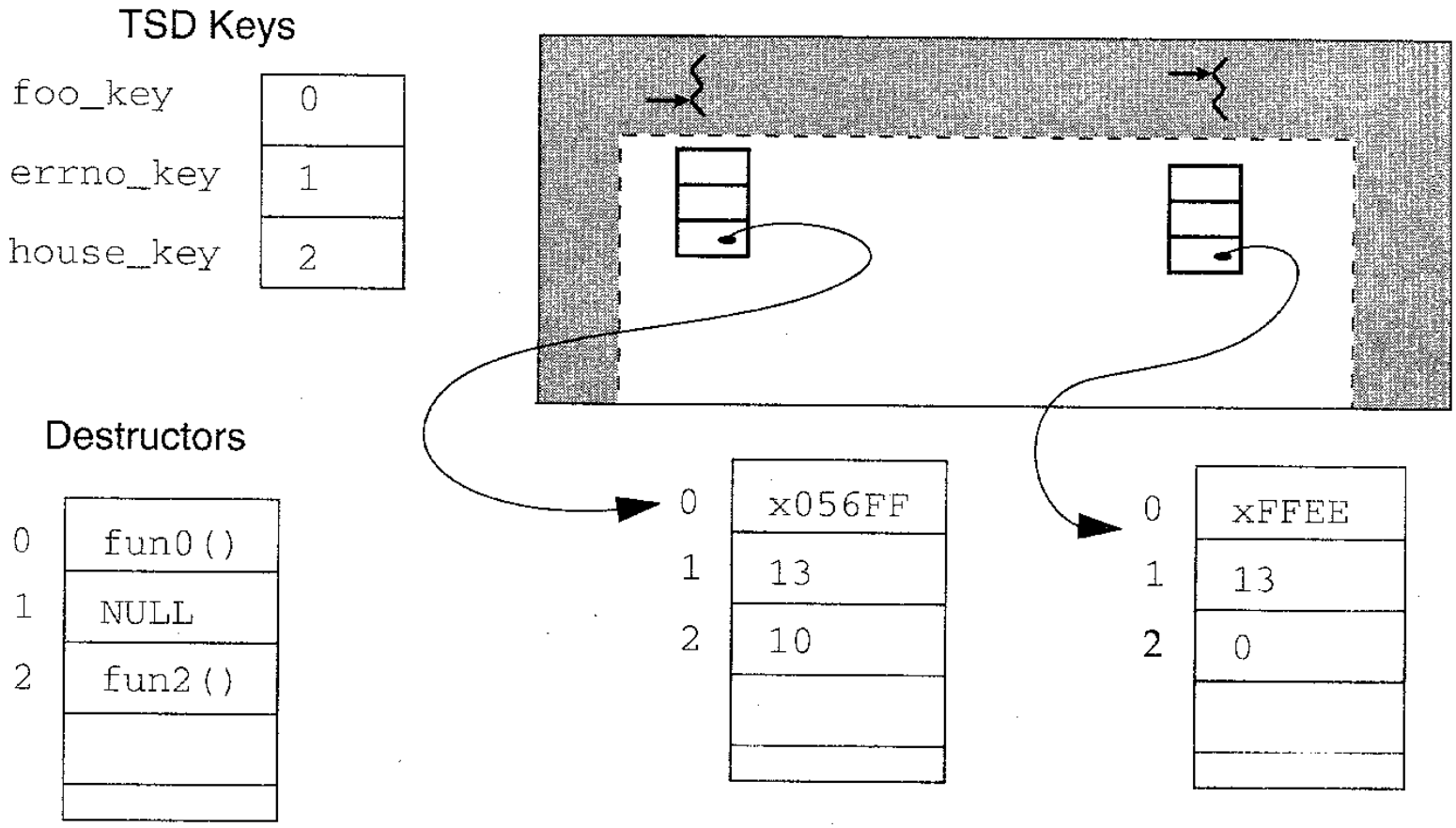


Figure 5-11 Thread-specific Data

```
extern thread_key_t house_key;

foo()
{ float n;
   the_getspecific(house_key, (void **) &n);

   . . . . .
}

main()
{ . . .

   thr_keycreate(&house_key, NULL);
   thr_setspecific(house_key, (void *) 3.14159265358979);
   thr_create(NULL, NULL, foo, NULL, NULL, NULL);
. . .
}
```

- TSD는 다른 global variabe의 access 보다 비싸다.
- TSD access - 약 40 instructions
- Global variable access - just one load instruction

1) Cache TSD

Normal Use of TSD

```
{ for (i . . . ) {
    thr_getspecific(v_key, &v);
    s+=f(v);
    }
}
```

Cached Use of TSD

```
{ thr_getspecific(v_key, &v);
  for (i . . . ) s+=f(v);
}
```

2) Passing Structure Instead of Using TSD

```
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;
    . . .
}
```

Thread Local Storage (TLS)

- TSD를 대치할 수 있는 방법.
- Global variable의 일부를 “thread local”로 선언할 수 있다.
- Accessing은 global variable과 똑같고 locking이 필요없다.

- 그러나,..
- compiler 나 memory map manipulation을 바꾸어야 한다.
- 새로운 keys를 dynamically allocate할 수 없다.
- 호환성이 없다.

- Solaris와 POSIX는 TLS를 사용하지 않는다. (TSD만 사용)
- NT에는 static TLS라는 것이 있다. compiler가 바뀌어야 한다.

Comparing the four libraries

Table 5-1 Comparing the Different Thread Specifications

Functionality	Solaris Threads	POSIX Threads	NT Threads	OS/2 Threads
Design Philosophy	Base Primitives	Near-Base Primitives	Complex Primitives	Complex Primitives
Scheduling Classes	Local/Global	Local/Global	Global	Global
Mutexes	Simple	Simple	Complex	Complex
Counting Semaphores	Simple	Simple	Complex	Complex
R/W Locks	Simple	Buildable	Buildable	Buildable
Condition Variables	Simple	Simple	Impossible	Impossible
Multiple-Object Synchronization	Buildable	Buildable	Complex	Complex
Other Complex Synchronization	Buildable	Buildable	Complex	Complex
Thread Suspension	Yes	Impossible	Yes	Impossible
Cancellation	Buildable	Yes	Difficult	Difficult
Thread-specific Data	Yes	Yes	Yes	Difficult
Signal-Handling Primitives	Yes	Yes	n/a	n/a
Compiler Changes Required	No	No	Yes	No

PROGRAMMING WITH THREADS

Programming with Threads

- implicit assumptions for single-threaded code:
  - 글로벌 변수에 쓰고 잠시 후에 읽으면 그 값은 이전에 쓴 값 바로 그 것이다.
  - Nonglobal, static variable도 마찬가지 이다.
  - synchronize할 것이 없으므로 synchronize할 필요가 없다.

Global Variables (errono)

- error code를 반환하는 대신에 errorno라는 global variable에 error code를 기입한다. System call이 실패했을 때, errorno를 읽어서 무엇이 잘못 되었는지 알 수 있다.

- Multi-threaded environment에서는 위와 같이 하는 것이 문제

- Solaris와 POSIX에서는 이를 thread-specific data라는 새로운 storage class를 통해서 해결하였다.
  - global to any procedure in which a thread might be running, but private to the thread only

```
/* error.h */
. . . . .
#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) */
```

이 함수 안에서는 thr_getspecific() 등을 호출 할 것이다.

Static Local Variables

- Solaris에서 getXbyY call 같은 것은 결과를 static local data structure에 넣고 그 pointer만을 반환 한다. 예: get hostbyname() 등

- 좋은 해결 방법은 caller가 결과를 넣기 위한 storage를 마련하는 것인데 이를 위해서는 새로운 interface의 정의가 필요하다. 즉 _r 이 원래의 이름에 붙은 새로운 함수를 정의한다.
  - 예: gethostbyname_r(3N) --- manual page 참조(man -s3n gethostbyname_r)

Synchronizing Threads

- 1. Single-Lock Strategy
  - a single, application-wide mutex lock - effectively a single-thre aded program
- 2. Reentrance
  - A reentrant function is one that behaves correctly if it is called simultaneously by several threads.
- Functions that access global state, like memory or file s, have reentrance problems.
- These functions need to protect their use of global stat e with the appropriate synchronization mechanisms pr ovided by threads library.

- Code Locking
  - at the function call level
  - The assumption is that all access to data is done through functions.

- Data Locking
  - locking code의 개념은 그대로 있음
  - Data locking guarantees that access to a collection of data is maintained consistently.
  - Data locking typically allows more concurrency than does code locking.
    - multiple reader, single writer protocol
  - The granularity of the lock depends on the amount of data it protects.
  - The common wisdom is to start with a coarse-grained approach, identify bottlenecks, and add finer-grained locking where necessary to alleviate t he bottlenecks.

□3. Avoiding Deadlock

- The most common error causing deadlock is *self-deadlock* or *recursive deadlock*.

- 순서대로 lock
- 순서대로 lock할 수 없을 때는 non-blocking library call을 이용
  - 예: mutex_trylock()

4. Scheduling Problems

- Because there is no guaranteed order in which locks are acquired, a common problem in threaded programs is that a particular thread never acquires a lock, even though it seems that it should.

- Thread library does not provide any time-slice mechanism for scheduling threads.

- Calling thr_yield(3T) just before the call to require the lock.

□5. Locking Guidelines

- Try not to hold locks across long operations.
- Don’t hold locks when calling a function that is outside the mo dule.
- Don’t try for excessive processor concurrency.
- Acquire the locks in the same order.

Following Some Basic Guidelines

- Know what you are importing and whether it is safe
- Threaded code can safely refer to unsafe code only from the initial thread.
- Sun-supplied libraries are defined to be *safe* unless explicitly documented as unsafe.
- Specify -D_REENTRANT when compiling
- When making a library safe for mutithreaded use, do not thread global proce ss operations.

- Creating Threads
  - The Solaris threads package caches the threads data structure, stacks, and LWPs so th at the repetitive creation of unbound threads can be inexpensive.
  - So, creating and destroying threads as they are required is usually better than attemptin g to manage a pool of threads that wait for independent work.
- Thread Concurrency
  - thr_setconcurrency(3T)
  - DB server(expected # of active user) / Window server(expected # of active clients) / File c opy program (2)
- Bound Threads
  - Use bound threads obly when a thread needs resources that are available only through t he underlying LWP. (kernel level real time scheduling)