Chapter 12: Multithreaded Programming and Multiprocessors

12.1 Why Multithreading?

Difference Between Process + Thread

- A thread is part of a process
- A thread is similar to a process in that it represents an active unit of processing
- Now, we redefine **process = program + state of all threads executing in program**

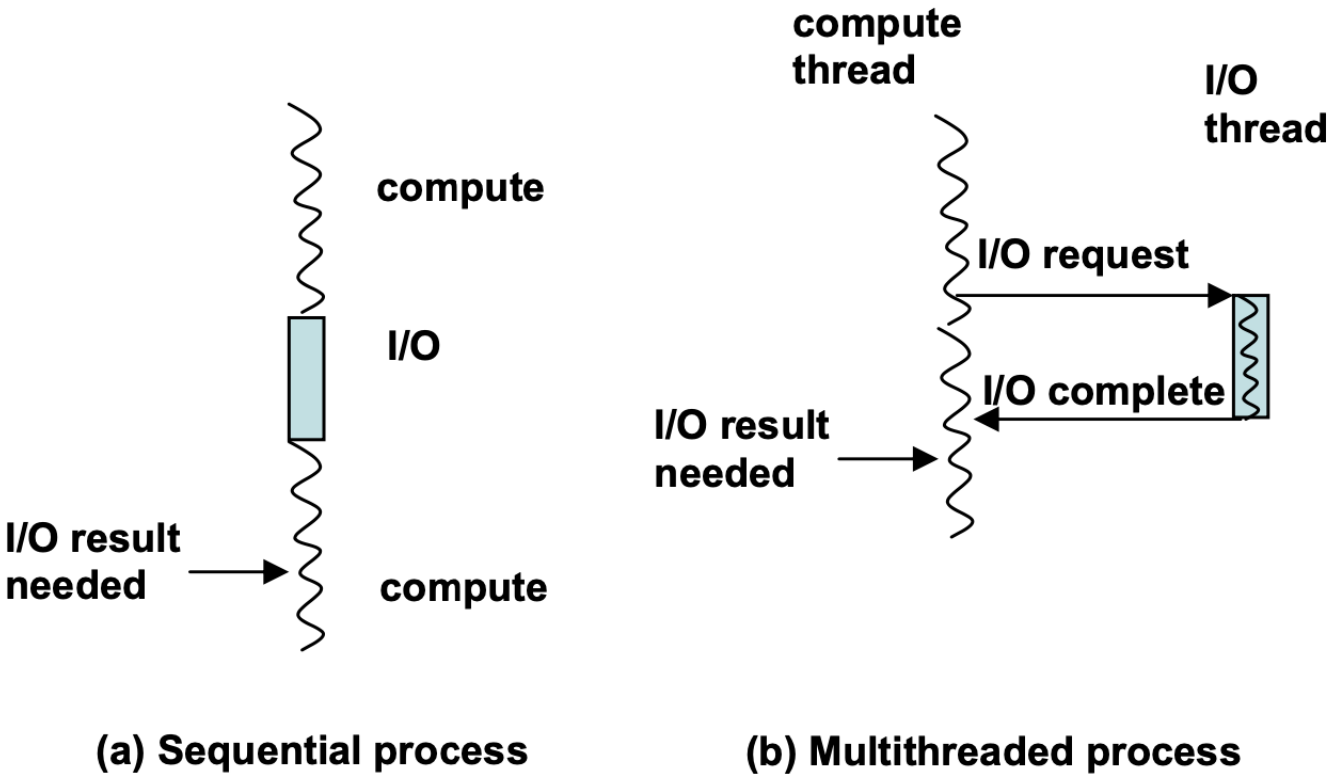


*Figure 12.1: Overlapping Computation with I/O using threads***Example**

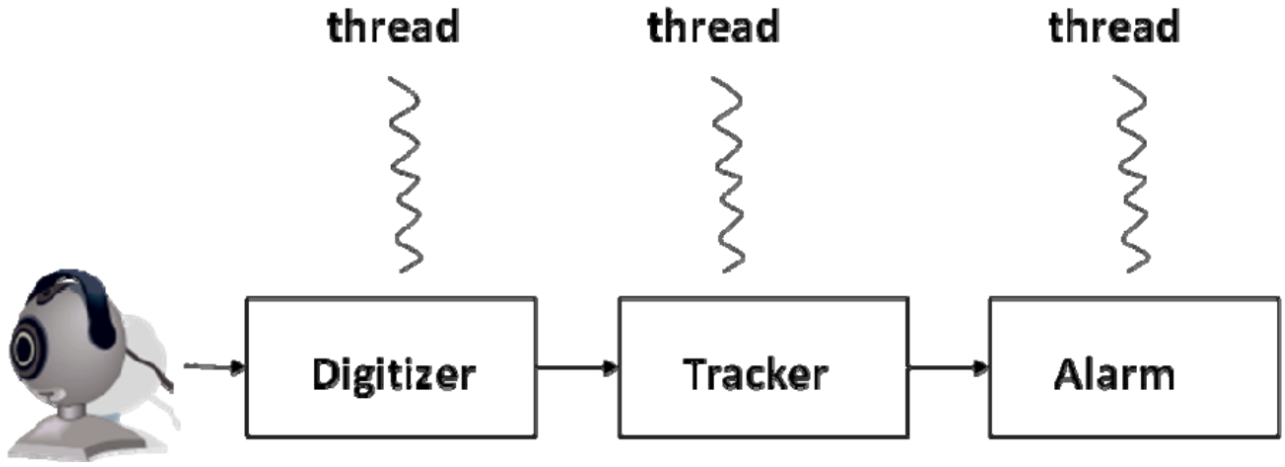


Figure 12.2: Video Processing Pipeline

12.2 Programming Support For Threads

- creation
  - pthread_create(top-level procedure, args)
- termination
  - return from top-level procedure
  - explicit kill
- rendezvous
  - creator can wait for children
  - pthread_join(child_tid)
- synchronization
  - mutex
  - condition variables

Thread Creation

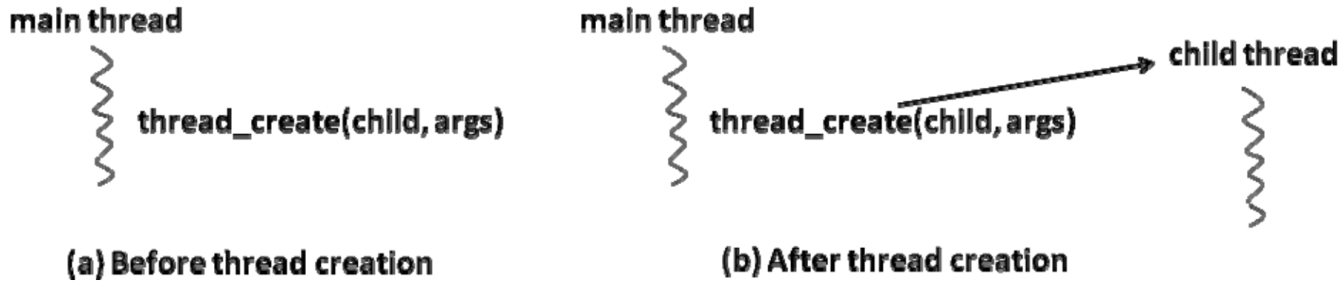


Figure 12.3: Thread Creation

```c
int foo(int n) {
    return 0;
}

int main() {
    int f;
    thread_type child_tid;

    child_tid = thread_create(foo, &f);

    thread_join(child_tid);
}
```

`thread_create` instantiates a new and independent entity called a *thread*

- Unlike processes, threads execute within a single address space

A thread automatically terminates when it exits the top-level procedure it started in *or* when `thread_terminate(tid)` is invoked

Read-write conflict, Race condition, and Non-determinism

Read-write conflict is a condition in which multiple concurrent threads are simultaneously trying to access a shared variable with at least one of the threads trying to write to the shared variable***Race condition*** is defined as the situation wherein a read-write conflict exists in a program without an intervening synchronization operation separating the conflictA ***data race*** is defined as a read-write conflict without an intervening synchronization operation wherein the variable being accessed by the threads is not a synchronization variableExample of data race

```
int count = 0; /* shared variable initialized to zero */

Thread 1 (T1)          Thread 2 (T2)          Thread 3 (T3)
.                      .                      .
.                      .                      .
count = count+1;       count = count+1;       count = count+1;
.                      .                      .
.                      .                      .
Thread 4 (T4)
.
.
printf("count = %d\n", count);
.
.
```

*Figure 12.4: Example of data race***Program order** is the combination of the textual order in which the instructions of the program appear to the programmer***Logical order** is the order in which these instructions will be executed in every run of the programA program is ***deterministic*** if for a given input the output of the program will remain unchanged in every execution of the program, ***non-deterministic*** otherwise

- Sequential programs are always deterministic

Synchronization Among Threads

Mutual Exclusion: if there are two threads, one a producer and the other a consumer then it is essential that the *producer not modify the shared buffer* while the *consumer is reading it*

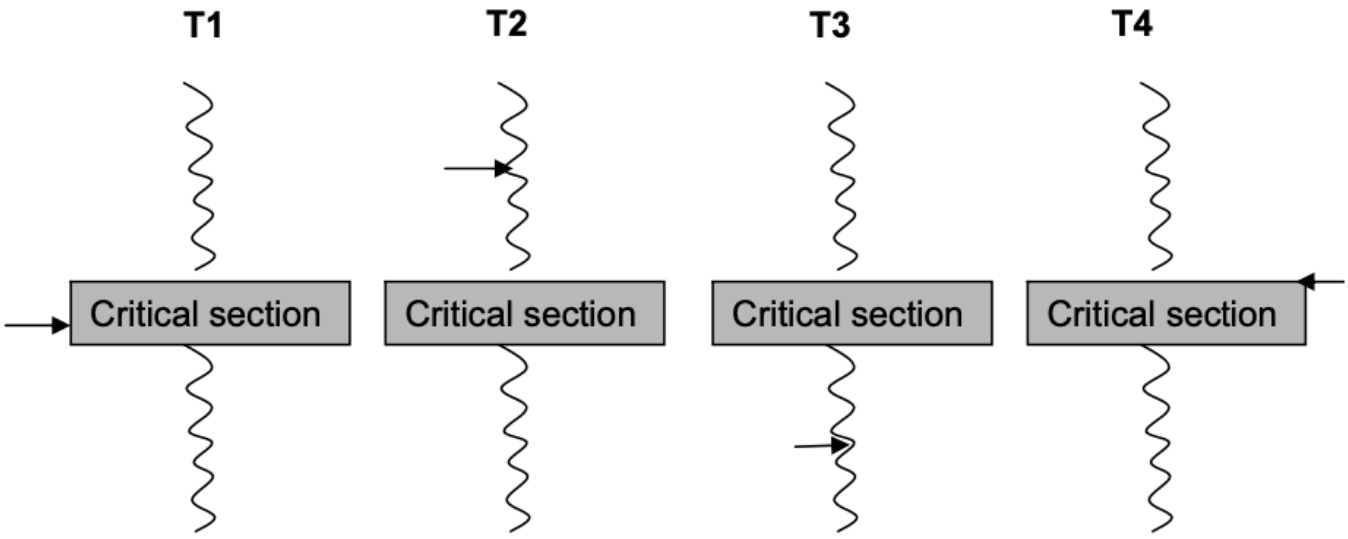
1. Mutual Exclusion Lock and Critical Selection

Only one thread can hold a particular lock at a time. Once a thread acquires a lock, other threads cannot get that same lock until the first thread releases the lock `mutex_lock_type mylock`

- The *blocked state* of a thread as one in which the thread cannot proceed in its execution until some condition is satisfied

Sometimes, a thread may not want to block but go on to do other things if the lock is unavailable `{success, failure} <- thread_mutex_trylock (mylock);`***Critical section*** is a

region of the program in which the execution of the threads is serializedExample



T1 is active and executing code inside its critical section. T2 is active and executing code outside its critical section. T3 is active and executing code outside its critical section. T4 is blocked and waiting to get into its critical section. (It will get in once the lock is released by T1).

2. Rendezvous

- Ex: parent wait for child to spawn

Rendezvous is a meeting point between threads of the same program

- All the threads of a given application that would like to participate in the rendezvous execute the barrier synchronization call
- Once all the threads have arrived at the barrier, the threads are allowed to proceed with their respective executions

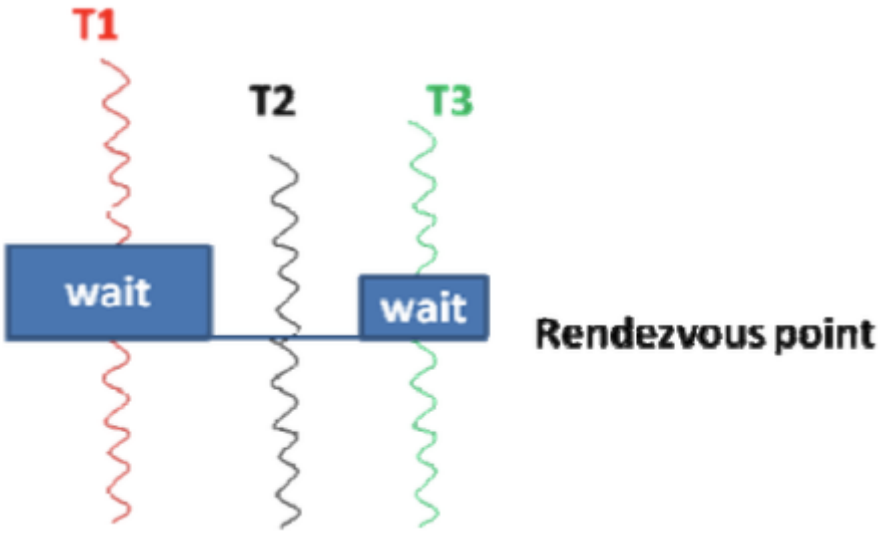


Figure 12.5: Rendezvous among threads

Condition Variables

- Leads to condition variables

```
thread_cond_wait(buf_not_empty, buflock);
```

- de-scheduling the thread that made that call
- implicit unlock on the lock variable

```
thread_cond_signal(buf_not_empty);
```

- signals any thread that may be waiting on the named condition variable
- library knows the specific lock variable associated with the wait call
- implicit lock on the lock variable

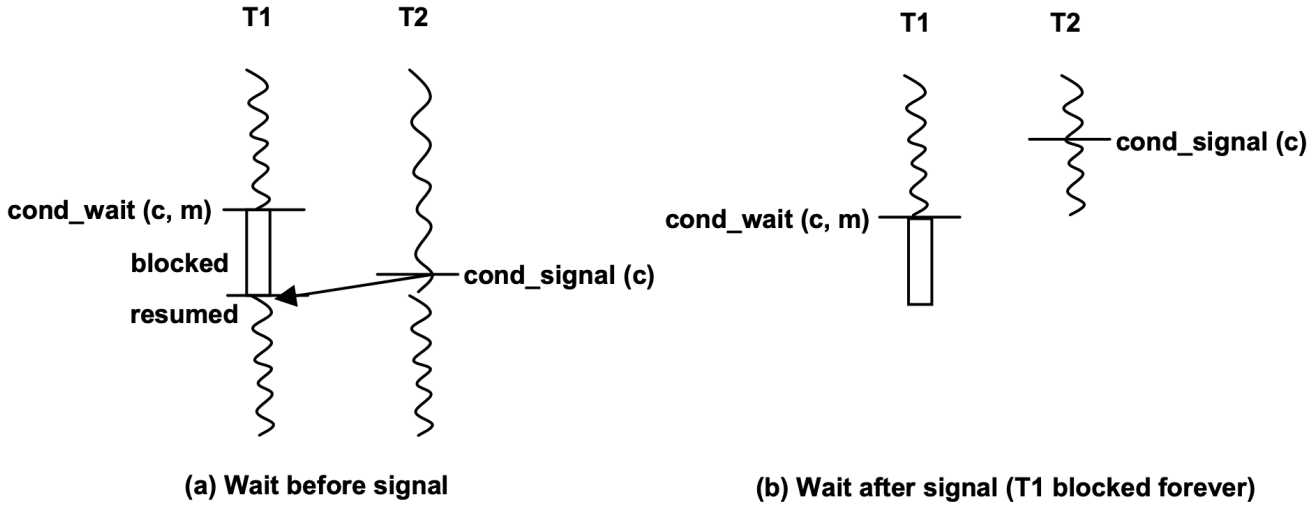


*Figure 12.6: Wait and Signal with condition variable***Turningpoint** Condition variable = enables a thread to wait for a condition without consuming processor cycles**Example (Rendezvous)** Write a function to be used by exactly 2 threads to rendezvous with each other

```c
wait_for_buddy() {
    /* both buddies execute the lock statement */
    thread_mutex_lock(mtx);

    if (buddy_waiting == FALSE) {
        /* first arriving thread executes this code block */
        buddy_waiting = TRUE;

        /* the following order is important
         * the first arriving thread will execute a wait statement */
        thread_cond_wait (cond, mtx);

        /* the first thread wakes up due to the signal from the second
         * thread, and immediately signals the second arriving thread
         */
        thread_cond_signal(cond);
    }
    else {
        /* second arriving thread executes this code block */
        buddy_waiting = FALSE;

        /* the following order is important */
        /* signal the first arriving thread and then execute a wait
         * statement awaiting a corresponding signal from the first thread
         */
        thread_cond_signal (cond);
        thread_cond_wait (cond, mtx);
    }
    /* both buddies execute the unlock statement */
  thread_mutex_unlock (mtx);
}
```

c

12.5 Threads As Software Structuring Abstraction

Dispatcher Model

- Single ready queue gets put into dispatcher

- Dispatcher thread dispatches requests as they come in to one of a pool of worker threads
- Once worker thread is completed, it returns to the free pool
- Request queue serves to smooth the traffic when the burst of requests exceeds server capacity
- Dispatcher serves as a workload manager as well, shrinking and growing the number of worker threads to meet the demand
- Example: hey I have a bunch of candies, I'll hand them out to you

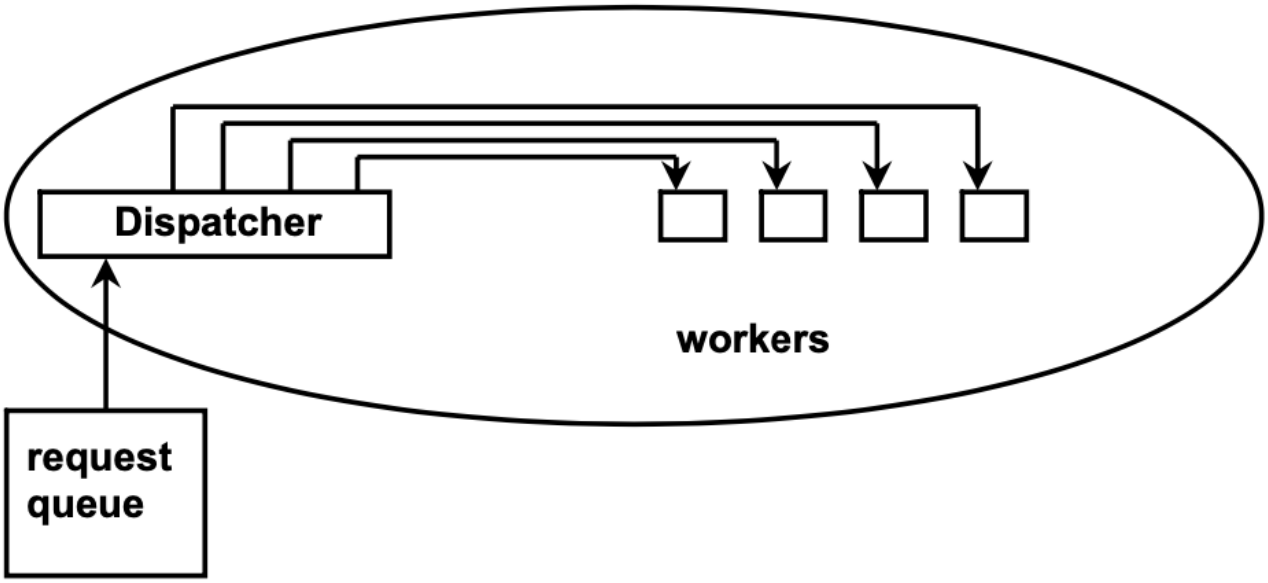
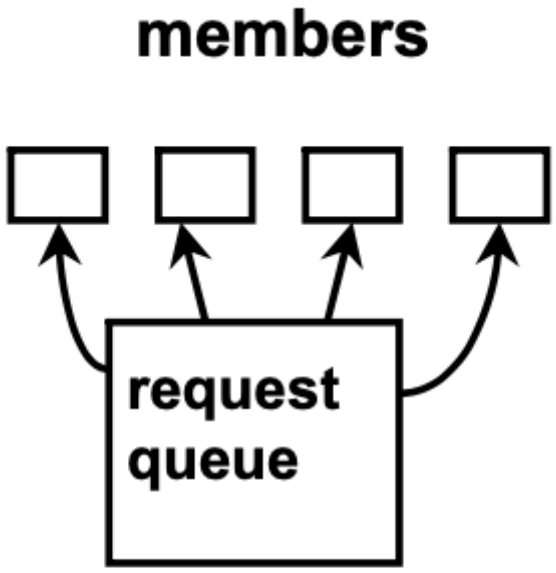


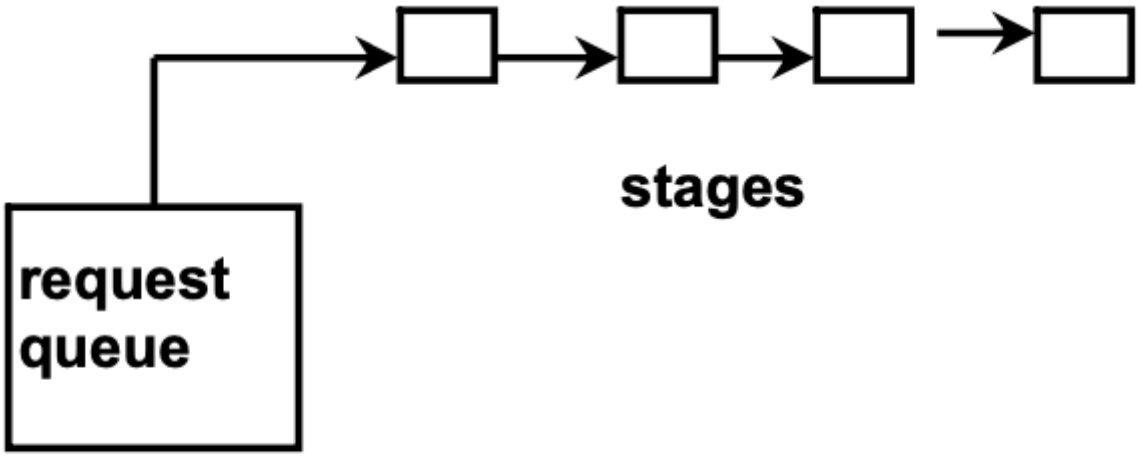
*Figure 12.14.1: Structuring Servers Using Dispatcher Model*Team Model

- Shared ready queue
- All the workers look in the mail box/ready queue, and if there is something available, it takes one
- Example: hey guys, here are a bunch of candies – come and help yourself



(b) Team model

*Figure 12.14.2: Structuring Servers Using Team Model*Pipelined Model



(c) Pipelined model

Figure 12.14.3: Structuring Servers Using PIpelined Model

12.7 OS Support For Threads

- Equivalent to procedure call to go between kernel and user
- No memory protection, no corruption
- Assumed users were only going to run a single program at a time, don't need all the bells and whistles

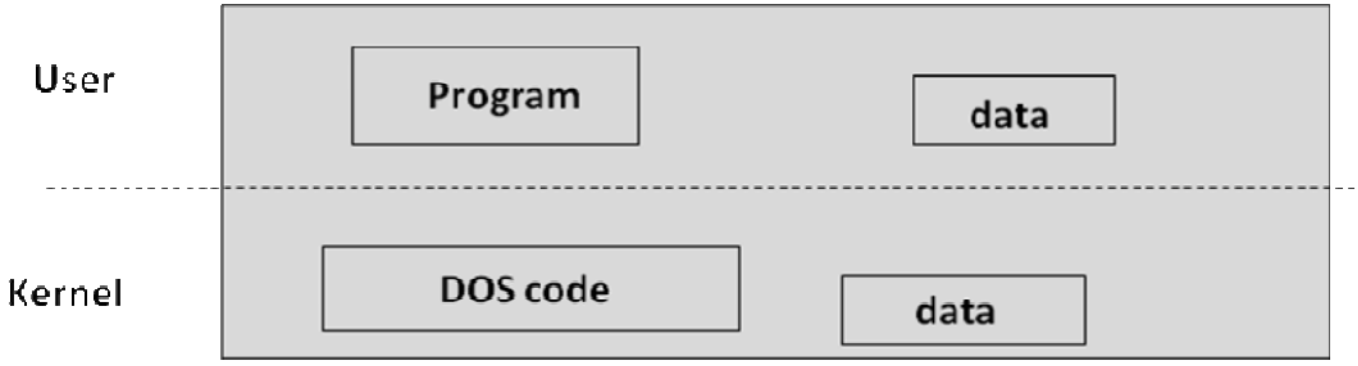


*Figure 12.15: MS-DOS User-Kernel Separation*Unix Memory Layout

- Protects the user from kernel section
- Each process has its own address space

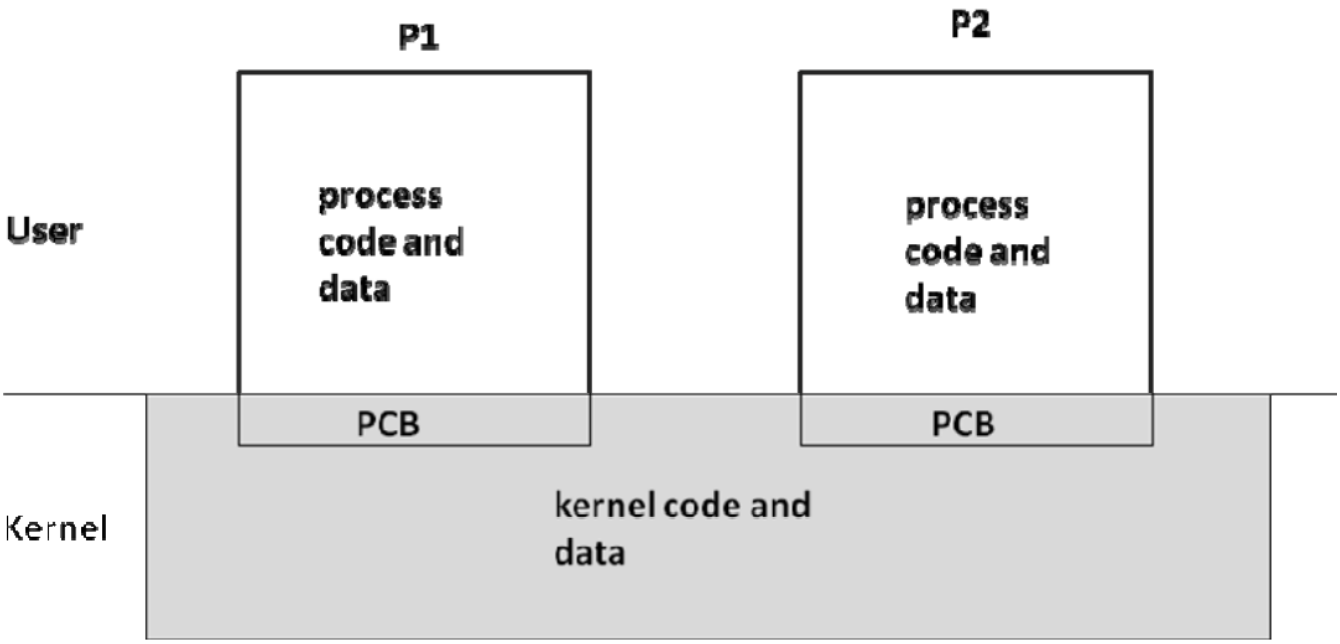


Figure 12.16: Memory Protection in Traditional OS

- PCB contains information for fully specifying the activity of this single thread on the processor (current PC value, stack pointer value, general-purpose register values, etc.)

- If a process makes a system call that blocks the process (e.g. read a file from the disk), then the program as a whole does not make any progress

Modern Multithreaded OS

- Stack has to individualized for every thread

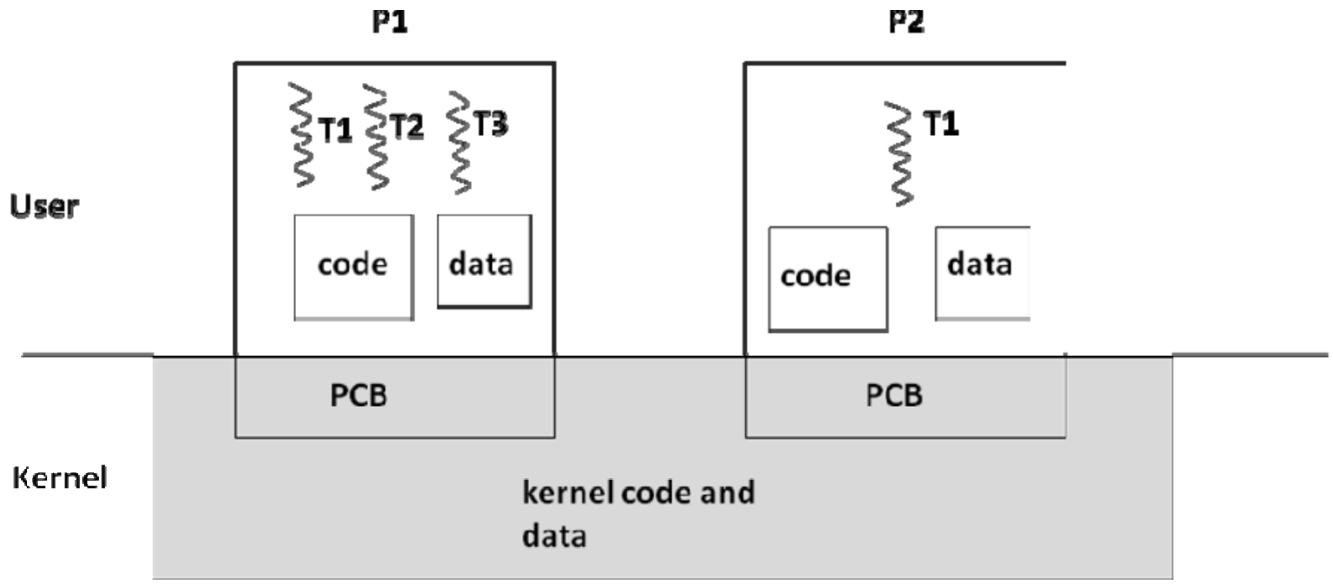


Figure 12.17: Memory Protection in Modern OS With Multithreading

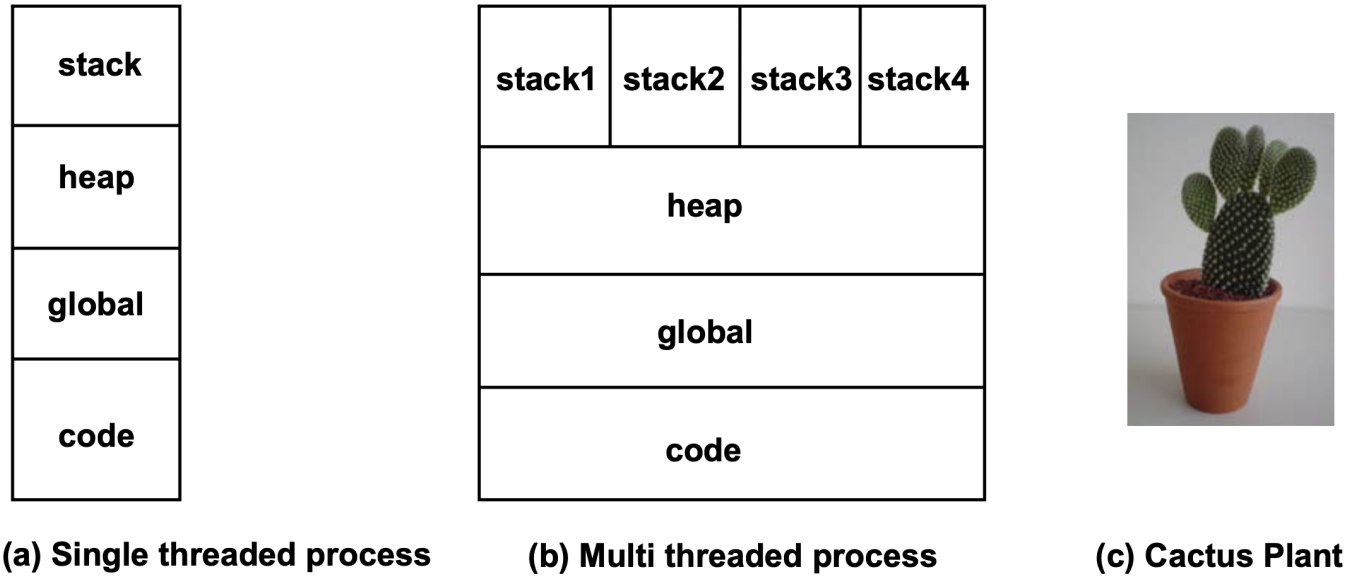


Figure 12.18: Memory Layout With Cactus Stack

- Threads share address space
- Thread control block (TCB) contains all the state information pertaining to a thread
    - PC value, stack pointer value, GPR contents

- Are the threads concurrent? Maybe if the CPU is a true multiprocessor (multiple cores); share the same CPU on a uniprocessor
- Threaded code need synchronization

User Level Threads

- OS independent
- Scheduler is part of the runtime system
- Thread switch is cheap (save PC, SP, regs)
- Scheduling is ***customizable***, i.e. more app control
- ***Downside:*** Blocking call by thread blocks process
    - OS doesn't know about the multiple threads

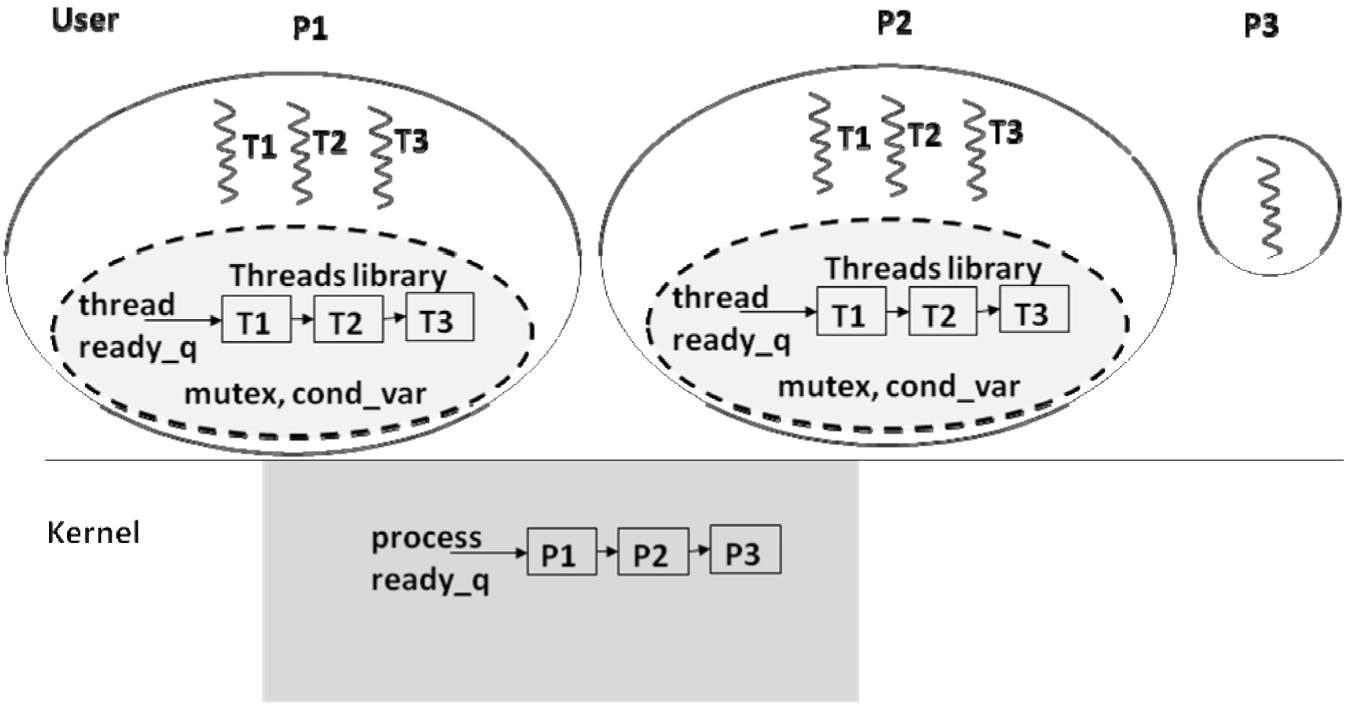


*Figure 12.19: User Level Threads****Turningpoint*****Question**: User level threads with process level scheduling**Answer:** Useful for structuring**Question:** User level threads with process level scheduling**Answer:** will have no performance advantage on a multiprocessor compared to a uniprocessor (OS doesn't know about the threads)**Upcall** What happens when one of the threads makes a blocking system call?

- OS has to block the whole process because it has no knowledge of other threads

1. Wrap all OS calls with an envelope that forces all the calls to go through the thread library

2. Upcall mechanism

- Allows the thread scheduler (in the library) to perform a thread switch and/or defer the blocking call by the thread to a later more opportune time
- Allows extension to OS for supporting upcall

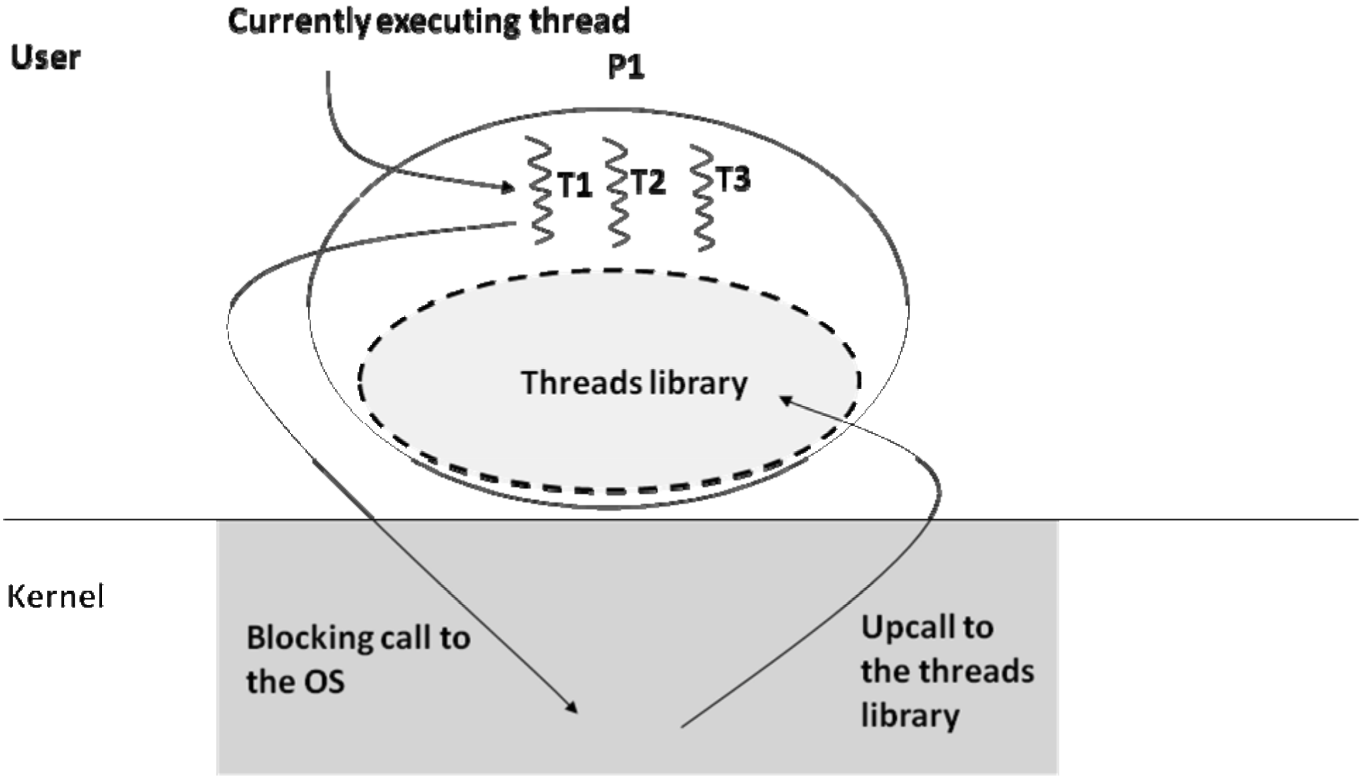


Figure 12.20: Upcall mechanism

- Thread library registers a handler with the kernel
- Thread makes a blocking call to the OS
- OS makes an upcall to the handler, thus alerting the threads library of the blocking system call made by a thread in that process

Kernel Level Threads

- Expensive thread switch
- Makes sense for blocking calls by threads
- Kernel becomes complicated: process v.s. threads scheduling
- Thread packages become non-portable
- Supports true multi-processors
- OS provides the thread library
- Threading in app visible to OS

- Each thread needs its own stack
- All threads of a process live in a single address space

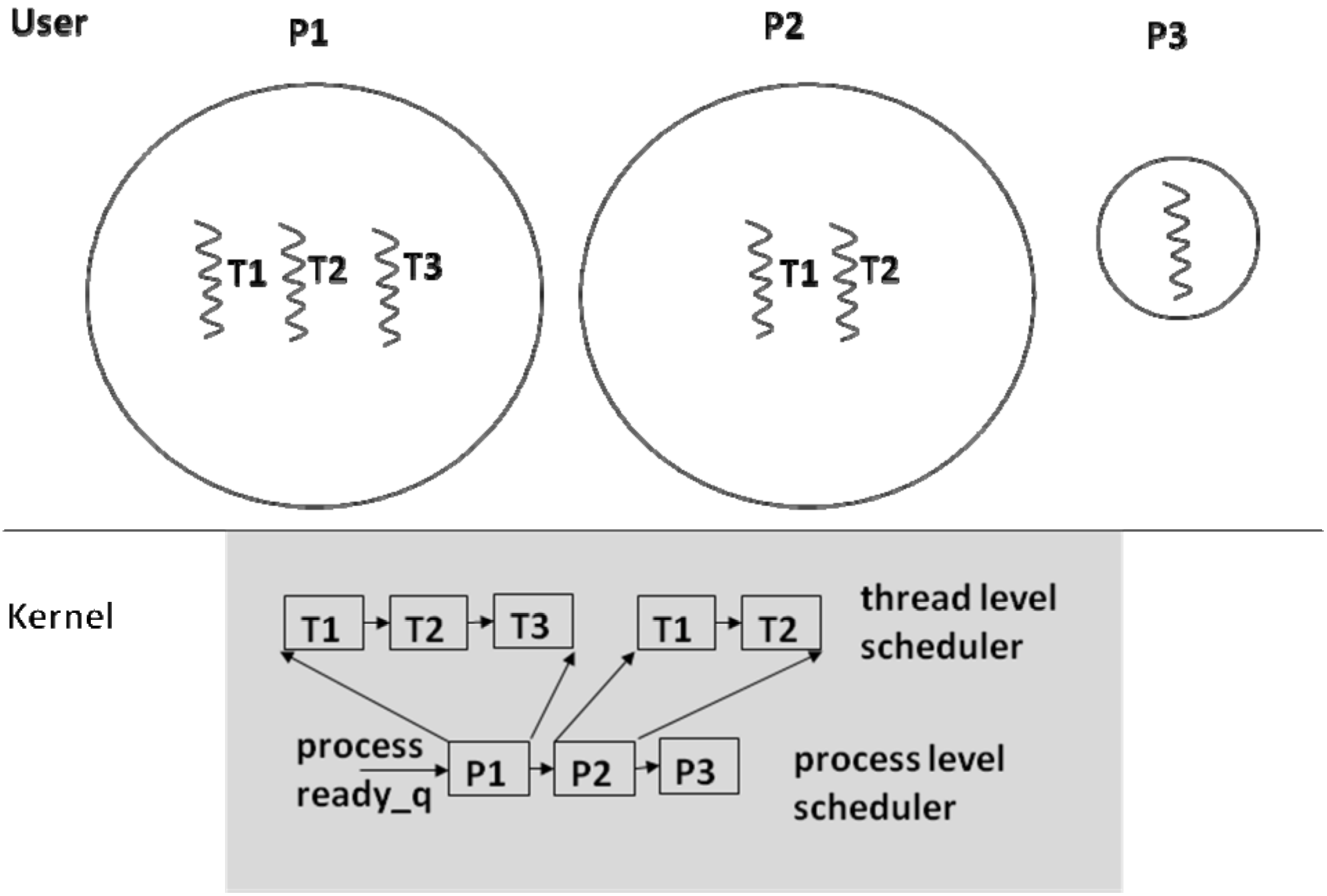


Figure 12.21: Kernel Level Threads

12.8 Hardware Support For Multithreading

1. Thread creation and termination
2. Communication among threads
3. Synchronization among threads

Thread creation, termination, and communication among threads

- Does not require any special hardware support

Inter-thread synchronization

```
Lock:
     if (mem_lock == 0) mem_lock = 1;
```

```
        else block the thread;


Unlock:
     mem_lock = 0;
```

The datapath actions necessary to implement the lock algorithm are as follows: • Read a memory location • Test if the value read is 0 • Set the memory location to 1We need to do all of these **atomically**. Therefore, to make the lock algorithm atomic, we introduce a new instruction: `Test-And-Set memory-location`**Example (Binary Semaphore)**

```c
static int shared-lock = 0; /* global variable to both T1 and T2 */

/* shared procedure for T1 and T2 */
int binary-semaphore(int L) {
    int X;
    X = test-and-set (L);

    /* X = 0 for successful return */
    return(X);
}
```

12.9 Multiprocessors

Symmetric Multiprocessor (SMP) - all processors have an identical view of the system resources

- Cost effective approach to increasing the system performance at a nominal increase in total system cost

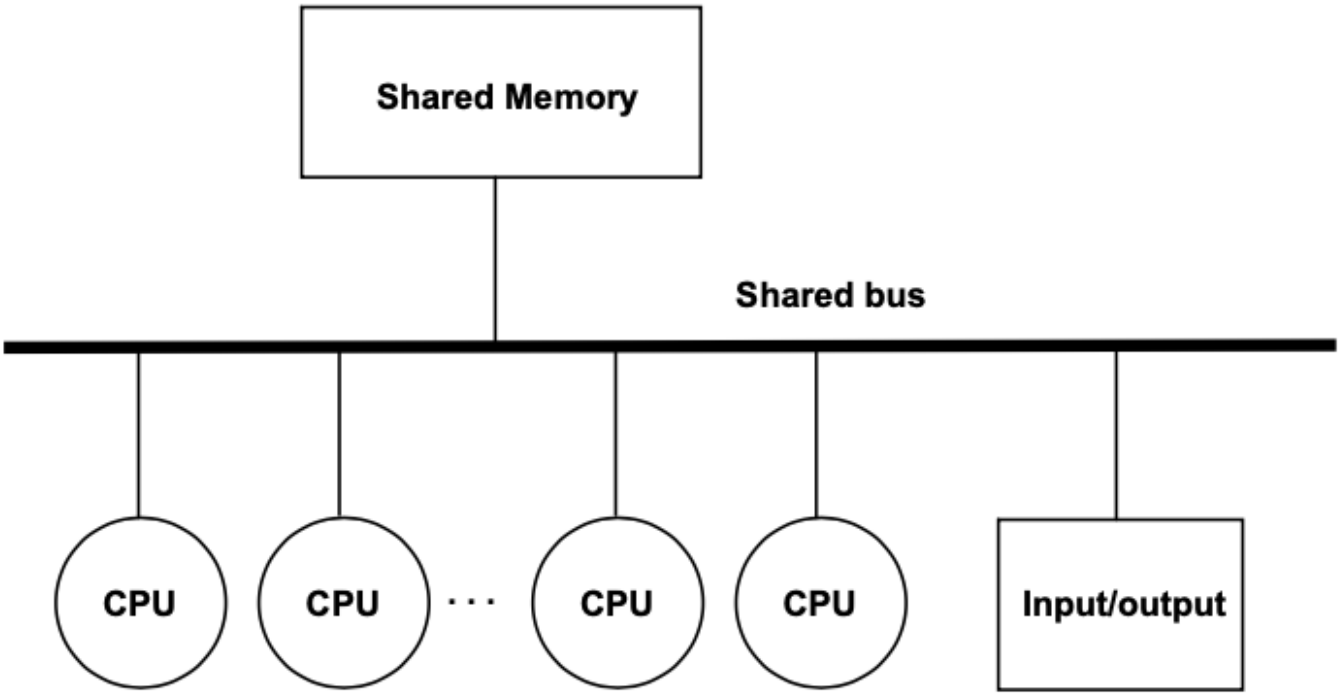


Figure 12.25: Symmetric Multiprocessor

1. Threads of the same process share the same page table
2. Threads of the same process have identical views of the memory hierarchy despite being on different physical processors
3. Threads are guaranteed atomicity for synchronization operations while executing concurrently

***Turningpoint*Question:** Synchronization atomicity in an SMP**Answer:** Automatic since T&S instruction of the ISA is atomic (instructions are by definition atomic)

SMP Page Tables

- Processors share the same memory as in Figure 12.25
- First requirement already met (same page table)
- *beyond scope of this course: each processor independently executes the same OS, many problems with TLB consistency, scheduling, ... etc.*

Memory Hierarchy

- Each CPU has its own TLB and cache
- Each per-processor cache may be currently encacheing the same memory location

- *Hardware* is responsible for maintaining a consistent view of shared memory that may be encached in the per-processor caches
    - This is called ***multiprocessor cache coherence***

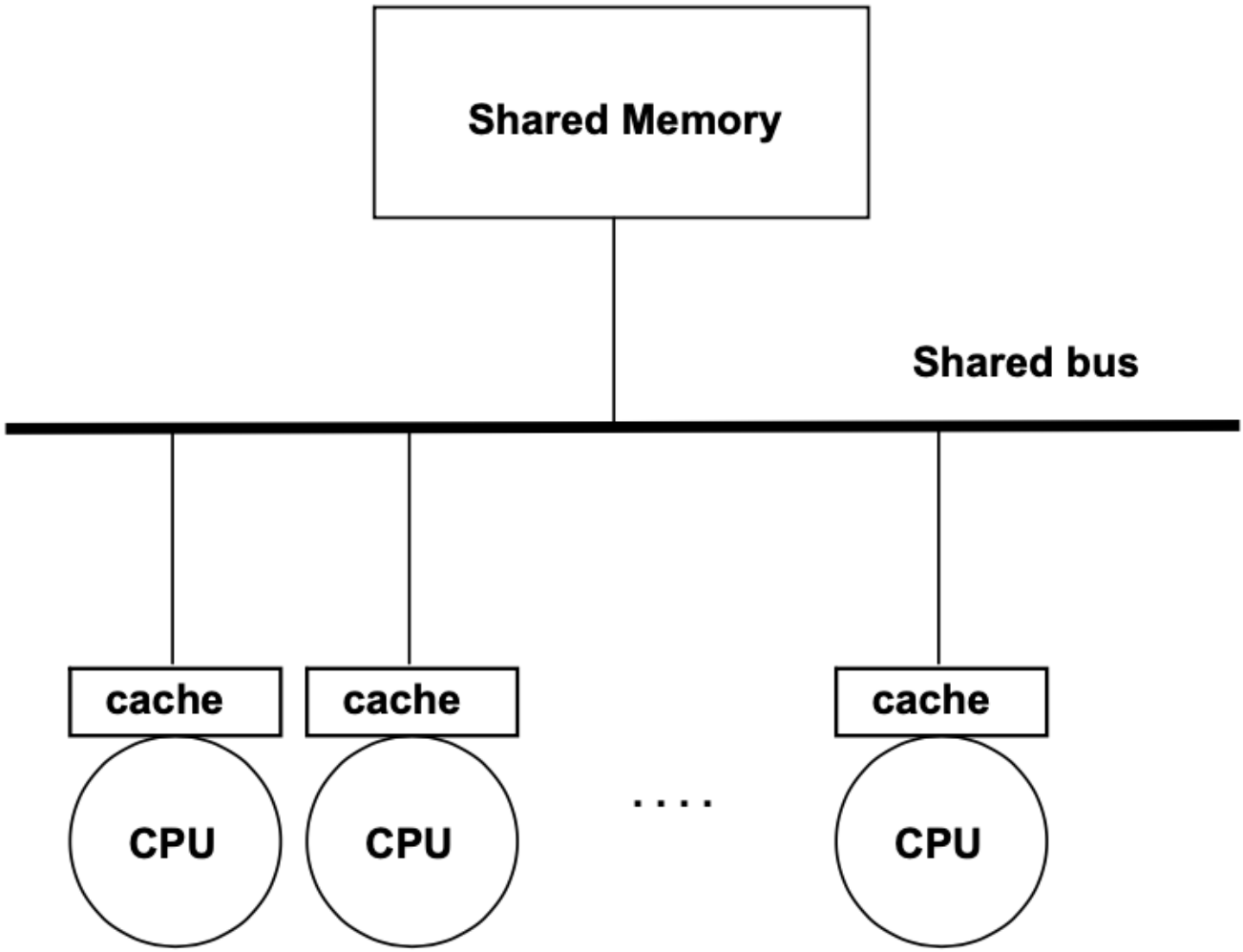


Figure 12.26: SMP With Per-Processor Caches

- Threads T1, T2, T3 execute on Processors P1, P2, P3 respectively
- All three of them currently have location X cached in their respective caches
- T1 writes to X
    - **Write-Invalidate Protocol**: *Invalidate* copies of X in the peer caches
        - Requires enhancement to the shared bus in the form of an invalidation line
        - Caches *monitor* the bus by snooping on it to watch out for invalidation requests
        - Upon request, every cache checks if this location is cached locally and invalidates it if it is
    - **Write-Update Protocol:** *Update* copies of X in the peer caches

- Manifests as a memory write on the bus
- Peer caches observe this bus request update their copies of X

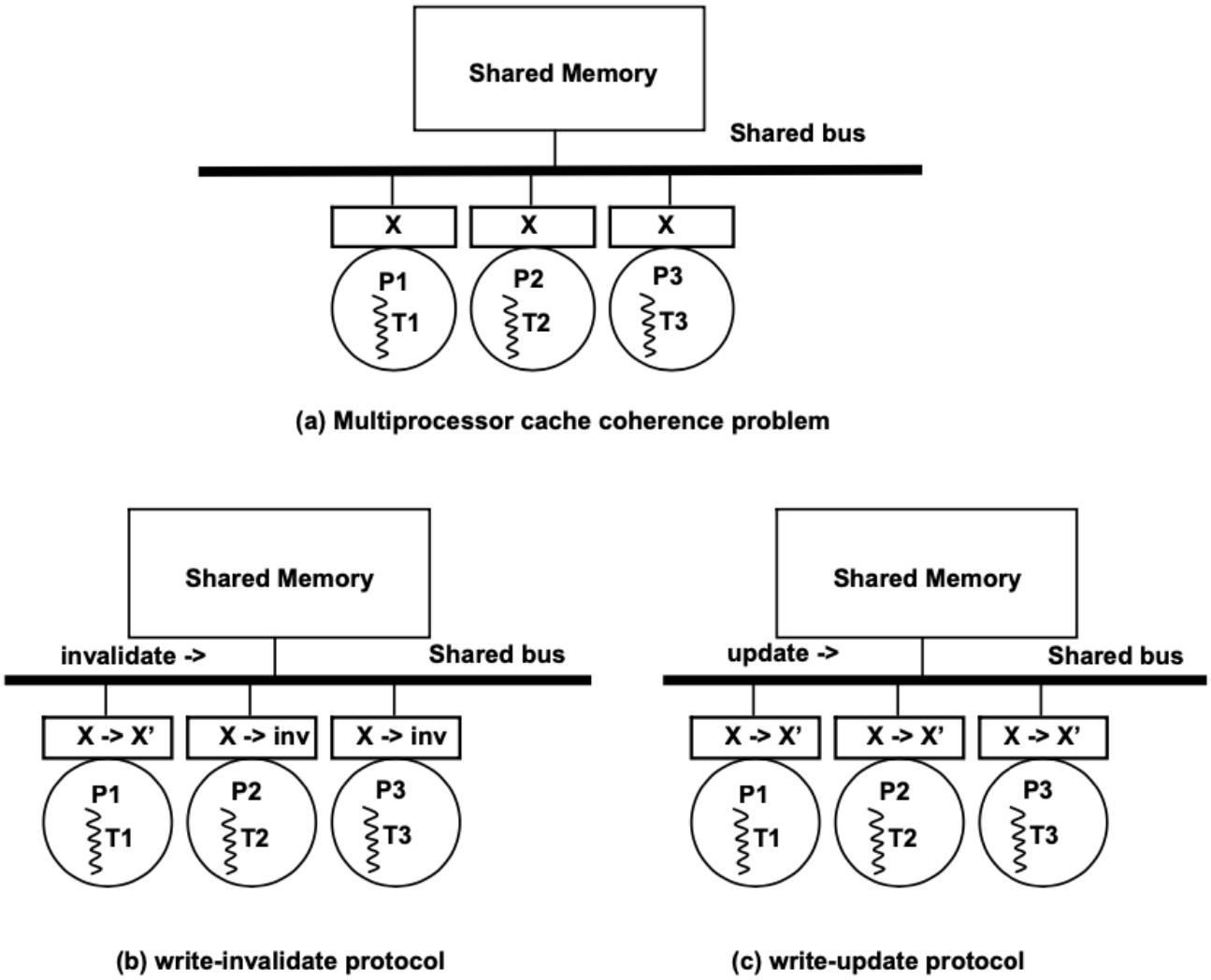


*Figure 12.27: Multiprocessor Cache Coherence Problem and Solutions*Snoopy caches is the term for bus-based cache coherence protocols

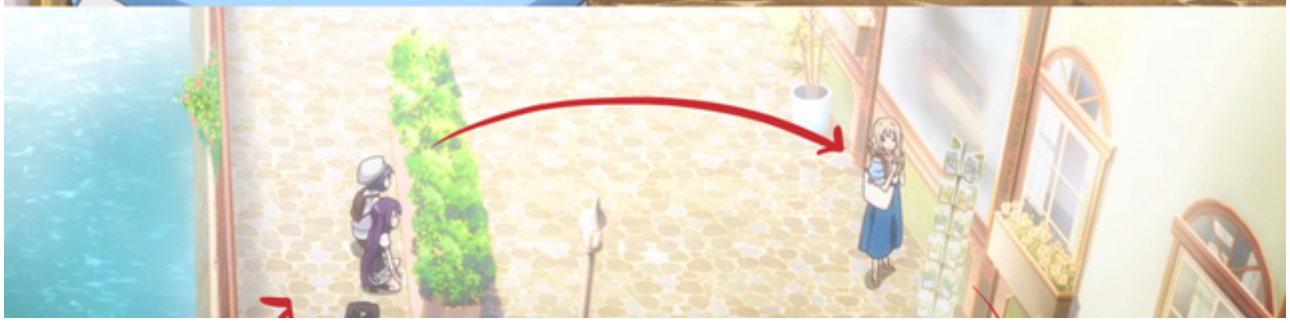
- Do not work if the processors do not have a shared bus

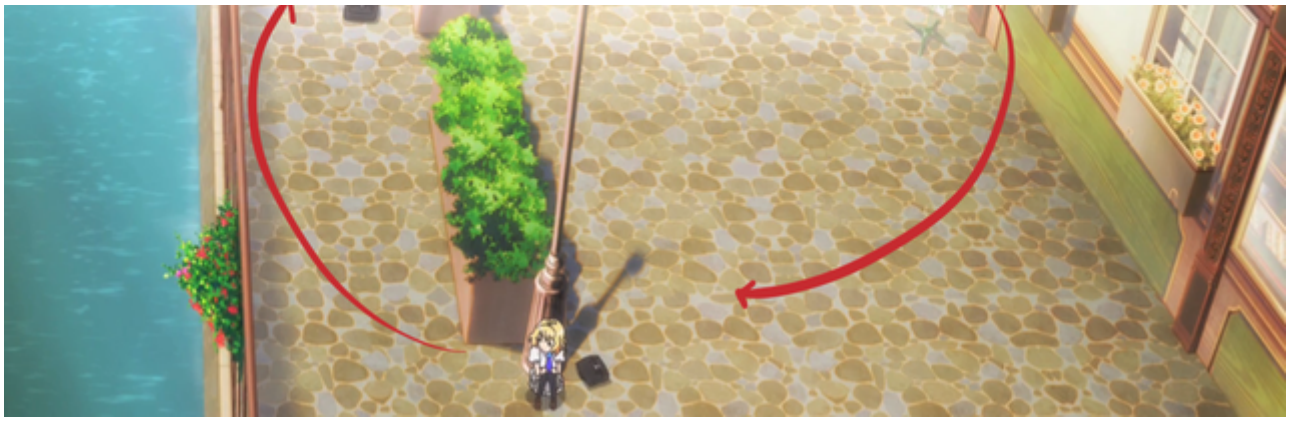
Ensuring Atomicity

12.10 Advanced Topics

Deadlocks

Deadlock illustrated





1. Need for mutual exclusion
2. Lack of preemption

- Only one court and 2 rackets
- Nick and his partner goes to claim rackets
- Alex and his partner goes to claim court
- Deadlocked

- **Mutual exclusion**: a resource can be used only in a mutually exclusive manner
- **No preemption**: the process holding a resource has to give it up voluntarily
- **Hold and wait**: a process is allowed to hold a resource while waiting for other resources
- **Circular wait**: there is a cyclic dependency among the processes waiting for resources (A is waiting for resource held by B; B is waiting for a resource held by C; C....X; X is waiting for a resource held by A)

Deadlock Example Consider a system with three resources:

```
1 display, 1 KBD, 1 printer
```

```
P1 needs all three resources
P2 needs KBD
P3 needs display
P4 needs KBD and display
```

Avoidance: Allocate all needed resources as a bundle at the start to a process. Essentially, this amounts to not starting P2, P3, or P4 if P1 is running; not starting P1 if any of the others are running.**Prevention:** Have an artificial ordering of the resources, say KBD, display, printer. Make the processes always request the three resources in the above order and release all the resources a process is holding at the same time upon completion. This ensures no circular wait (P4 cannot be holding display and ask for a KBD; P1 cannot ask for printer without already having the display, etc).**Detection**: Allow resources to be requested individually and in any order. We will assume all processes are re-startable. If a process (P2) requests a resource (say KBD), which is currently assigned to another process (P4) and if P4 is waiting for another resource, then force a release of KBD by aborting P4, assign the KBD to P2, and restart P4.

Starvation - some process is indefinitely blocked awaiting a resource