

Multiprocessors and Multithreading

What is multithreading?

- technique allowing program to do multiple tasks
- is it a new technique?
 - has existed since the 70's (concurrent Pascal, Ada tasks, etc.)
- why now?
 - emergence of SMPs in particular
 - “time has come for this technology”

- What is an SMP?
 - multiple CPUs in a single box sharing all the resources such as memory and I/O
- Is an SMP more cost effective than two uniprocessor boxes?
 - yes (roughly 20% more for a dual processor SMP compared to a uni)
 - modest speedup for a program on a dual-processor SMP over a uni will make it worthwhile

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

Programming with Threads

- synchronization
 - for coordination of the threads
- communication
 - for inter-thread sharing of data
 - threads can be in different processors
 - how to achieve sharing in SMP?
 - software: accomplished by keeping all threads in the same address space by the OS
 - hardware: accomplished by hardware shared memory and coherent caches

Synchronization Primitives

- lock and unlock
 - mutual exclusion among threads
 - busy-waiting Vs. blocking
 - `pthread_mutex_trylock`: no blocking
 - `pthread_mutex_lock`: blocking
 - `pthread_mutex_unlock`
- condition variables
 - `pthread_cond_wait`: block for a signal
 - `pthread_cond_signal`: signal one waiting thread
 - `pthread_cond_broadcast`: signal all waiting threads

example

```
T1=> lock(cs-mutex);  
      while (res-state = busy)  
          do nothing;  
      res-state = busy;  
      unlock(cs-mutex);
```

```
      use resource;
```

```
T2=> lock(cs-mutex);  
      res-state = free;  
      unlock(cs-mutex);
```

example with cond-var

```
lock(cs-mutex)
while(res-state = busy)
T1=>   wait(cond-var); /* implicitly give up mutex */
        /* implicitly re-acquire mutex */
        res-state = busy
unlock(cs-mutex)
use resource
lock(cs-mutex)
    res-state = free;
unlock(cs-mutex)
T2=>   wakeup(cond-var); /* will wake up T1 */
```

Example Threads Program

- Producer

repeat

produce item;

pthread_mutex_lock(m);

while (full) pthread_cond_wait(nonfull, m);

....

add item to buffer;

if (no more bufferspace) full = true;

empty = false;

pthread_cond_signal(non_empty);

pthread_mutex_unlock(m);

until false;

- Consumer

repeat

pthread_mutex_lock(m);

while (empty) pthread_cond_wait (non_empty, m);

remove item from buffer;

full = false;

if (no more items) empty = true;

pthread_cond_signal(nonfull);

pthread_mutex_unlock(m);

...

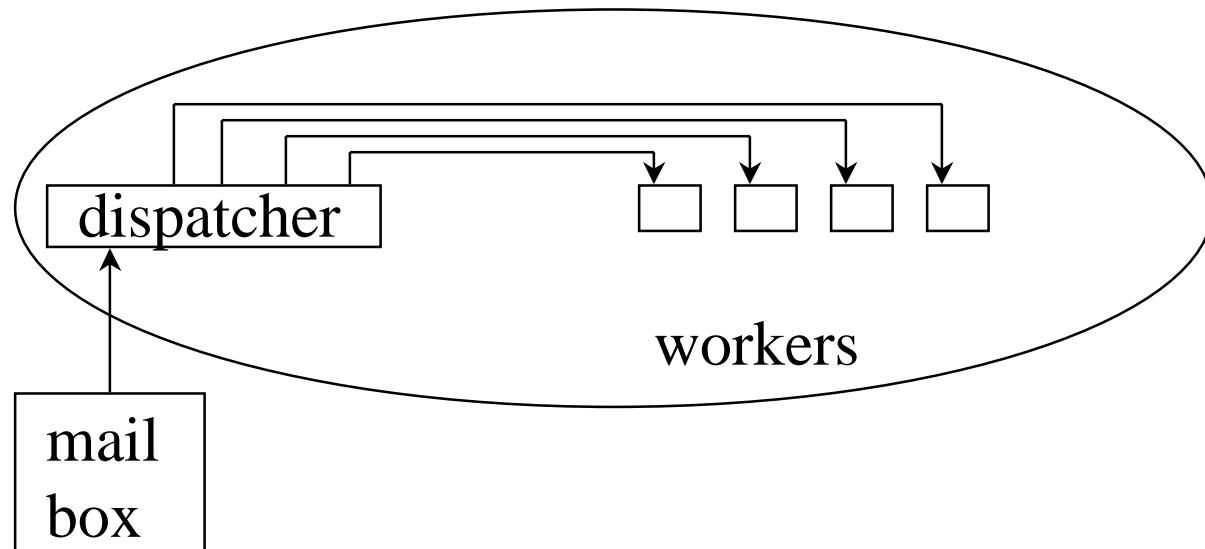
consume_item;

until false;

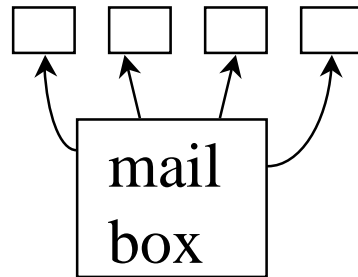
Using Threads

Servers

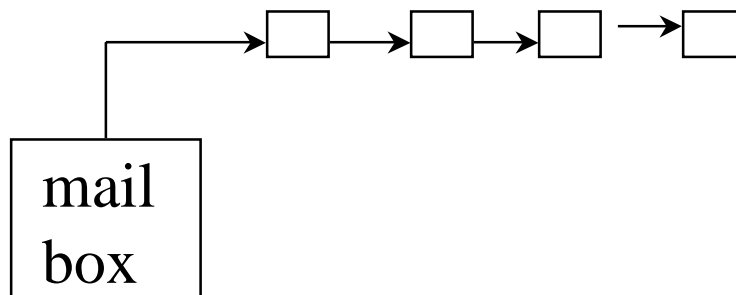
- dispatcher model



- team model



- pipelined model



Clients

- software simplicity
- exception handling
- terminal I/O
- signal handling

```

FILE *fp;

start_routine()
{ string fname;
  fname = self();
  fp = open(fname);
  <write file>
  print_file();
}

print_file()
{ char ch;
  ch = read(fp);
  print(ch);
}

T =>

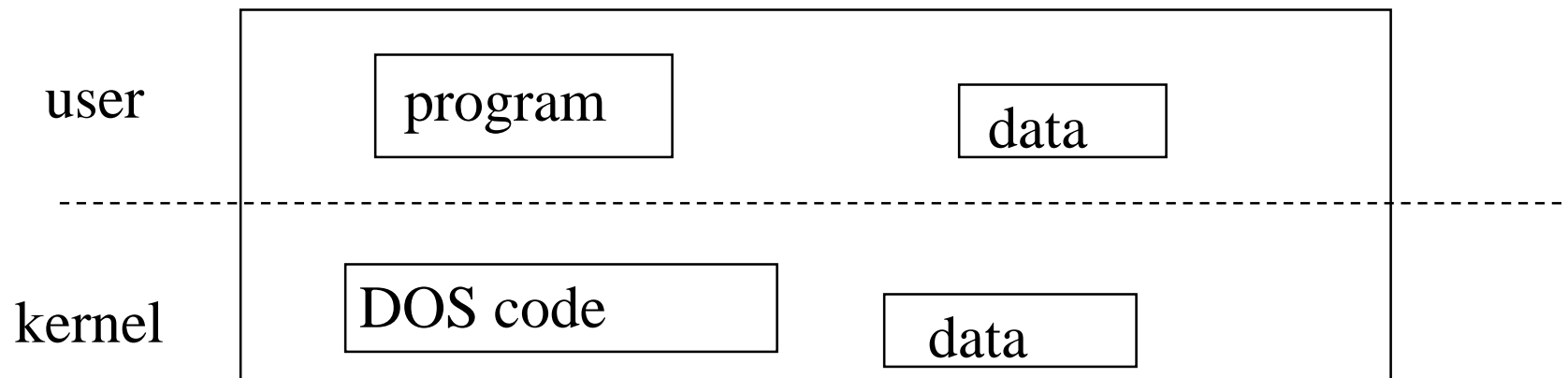
main()
{
  t1 = create_thread
      (start_routine);
  t2 = create_thread
      (start_routine);
  join(t1, t2);
}

```

Threads and OS

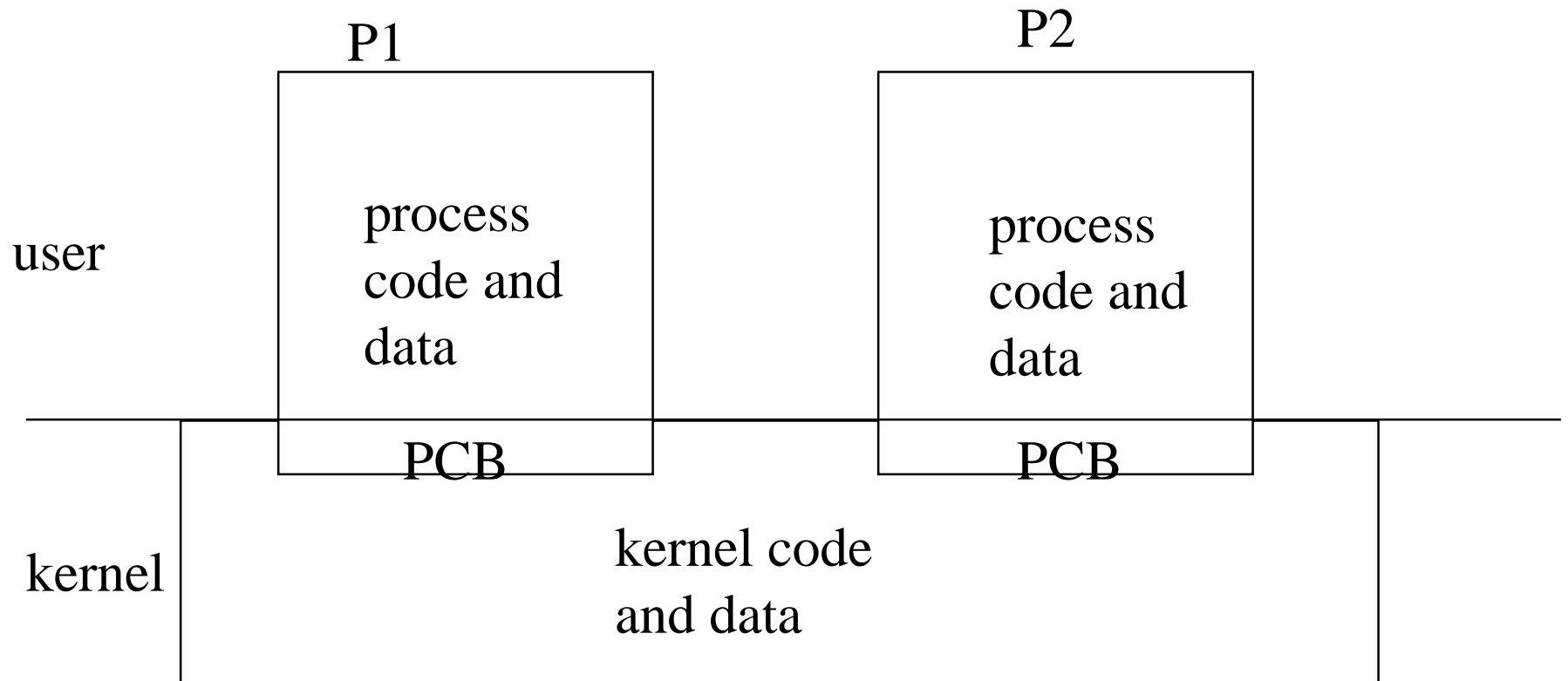
Traditional OS

- DOS
 - memory layout



- protection between user and kernel?

- Unix
 - memory layout



- protection between user and kernel?
- PCB?

- programs in these traditional OS are single threaded
 - one PC per program (process), one stack, one set of CPU registers
 - if a process blocks (say disk I/O, network communication, etc.) then no progress for the program as a whole

MT Operating Systems

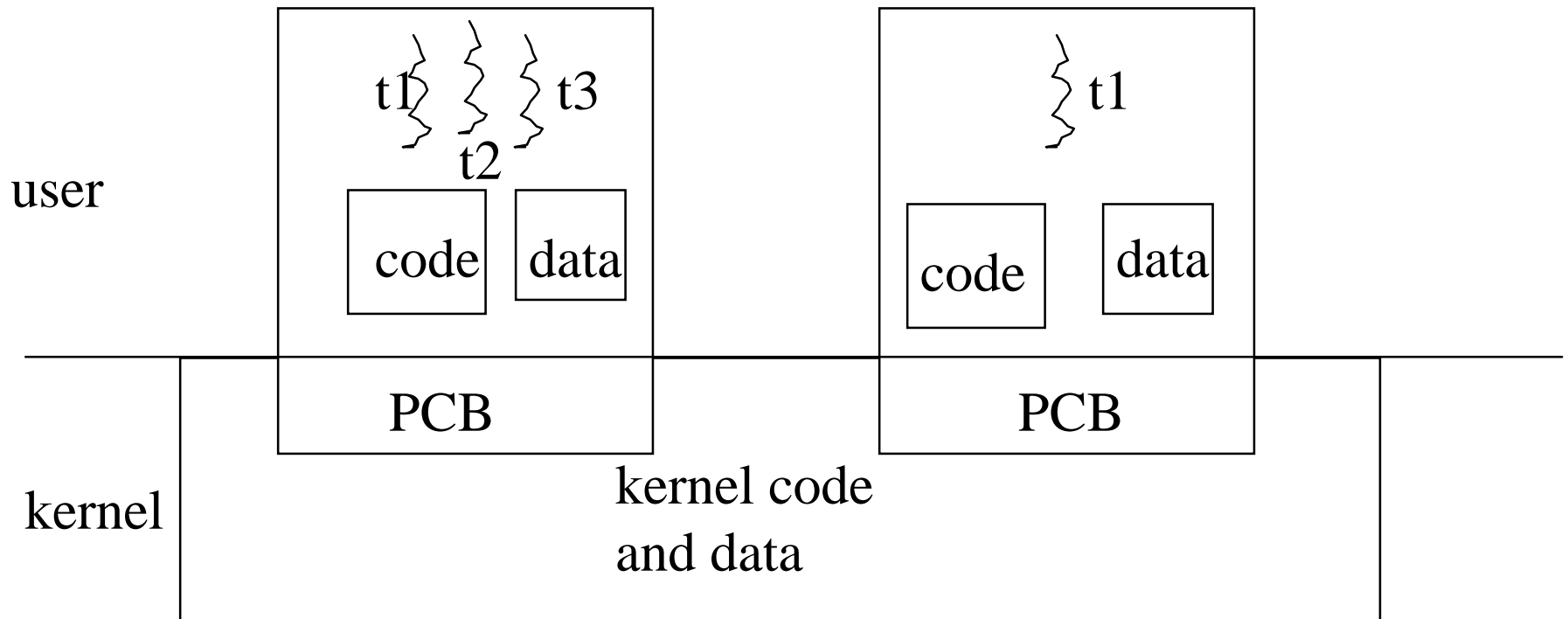
How widespread is support for threads in OS?

- Digital Unix, Sun Solaris, Win95, Win NT

Process Vs. Thread?

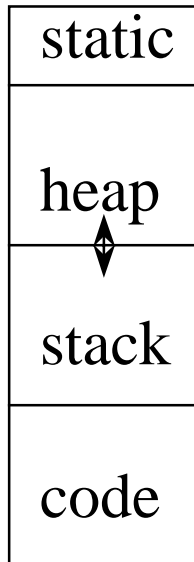
- in a single threaded program, the state of the executing program is contained in a process
- in a MT program, the state of the executing program is contained in several ‘concurrent’ threads

Process Vs. Thread

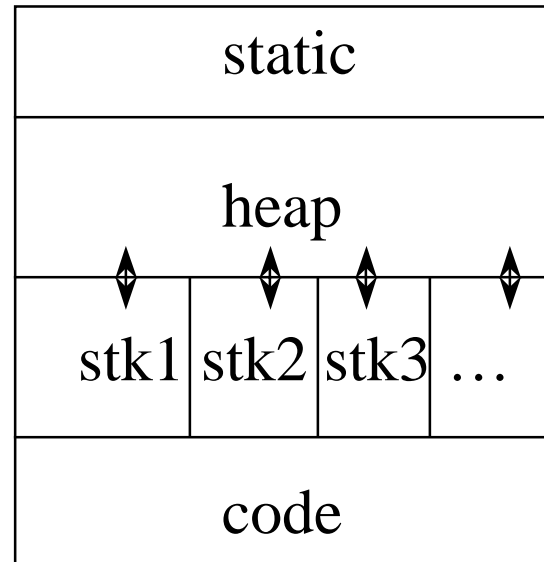


- computational state (PC, regs, ...) for each thread
- how different from process state?

Memory Layout



ST program

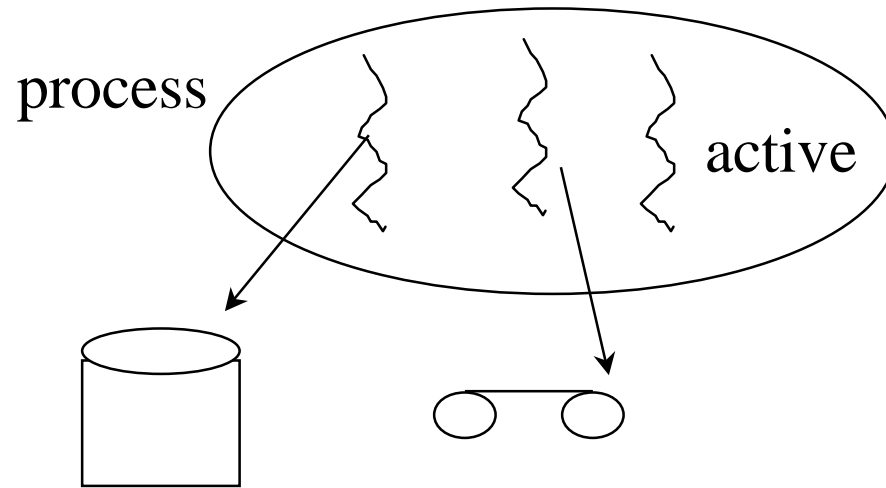


MT program

- MT program has a per-thread stack
- heap, static, and code are common to all threads

- threads
 - share address space of process
 - cooperate to get job done
- threads concurrent?
 - may be if the box is a true multiprocessor
 - share the same CPU on a uniprocessor
- threaded code different from non-threaded?
 - protection for data shared among threads
 - synchronization among threads

- threads in a uniprocessor?



- allows concurrency between I/O and user processing even in a uniprocessor box

Threads Implementation

- user level threads
 - OS independent
 - scheduler is part of the runtime system
 - thread switch is cheap (save PC, SP, regs)
 - scheduling customizable, i.e., more app control
 - blocking call by thread blocks process

- solution to blocking problem in user level threads
 - non-blocking version of all system calls
 - polling wrapper in scheduler for such calls
- switching among user level threads
 - yield voluntarily
 - how to make preemptive?
 - timer interrupt from kernel to switch

- Kernel level
 - expensive thread switch
 - makes sense for blocking calls by threads
 - kernel becomes complicated: process vs. threads scheduling
 - thread packages become non-portable
- problems common to user and kernel level threads
 - libraries
 - solution is to have thread-safe wrappers to such library calls

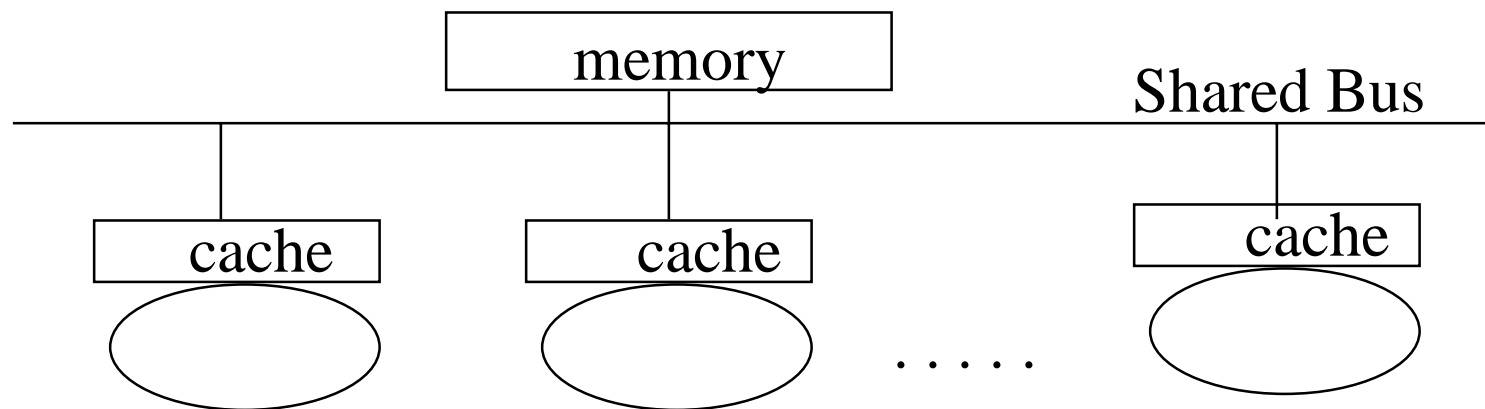
Solaris Threads

- Three kinds
 - user, lwp, kernel
- user: any number can be created and attached to lwp's
- one to one mapping between lwp and kernel threads
- kernel threads known to the OS scheduler
- if a kernel thread blocks, associated lwp, and user level threads block as well

Multiprocessor: First Principles

- processors, memories, interconnection network
- Classification: SISD, SIMD, MIMD, MISD
- message passing MPs: e.g. IBM SP2
- shared address space MPs
 - cache coherent (CC)
 - SMP: a bus-based CC MIMD machine
 - several vendors: Sun, Compaq, Intel, ...
 - CC-NUMA: SGI Origin 2000
 - non-cache coherent (NCC)
 - Cray T3D/T3E

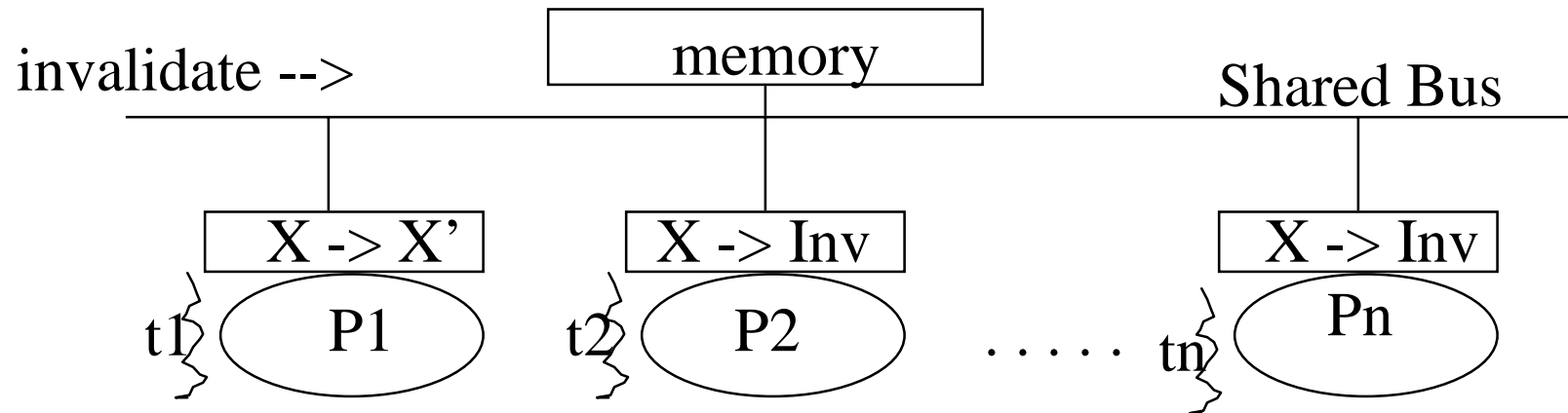
Caches and Consistency in SMP



- cache with each processor to hide latency for memory accesses
- miss in cache => go to memory to fetch data
- multiple copies of same address in caches
- caches need to be kept consistent

Cache Coherence

- write-invalidate



- write-update (also called distributed write)

