Threads and Critical Sections

Otto J. Anshus, Tore Larsen
University of Tromsø, Univ. of Oslo
(including slides from Kai Li, Princeton University)

Process

- Provides illusion that program runs protected on its own computer
- Address space
  - Separate, protected, user address space
    - All the state needed to run a program
    - Program text & data, open files, child PIDs, pending alarms, signal handlers, accounting information, etc. etc.
- Thread of execution, -thread of control, -thread
  - Historically only *one* thread per process (implicit, hence no term)
  - Program counter
  - Registers
  - Stack w/ linkage information (frames)
  - *How about more than one thread per process?*

Threads, Overview

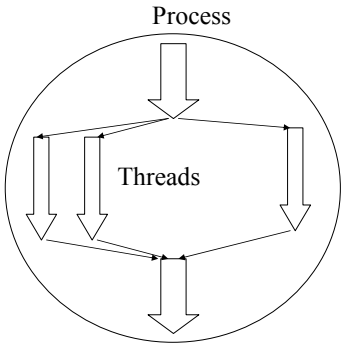
- Separate control flows (execution flows/threads) within a the confinement of one process
  - *Multithreading*
- Thread, entity scheduled for execution
- Threads in same process share address space, open files etc.
- Thread switching within process doesn't require new address mappings to be set up
  - *Lightweight processes*

Concurrency and Threads

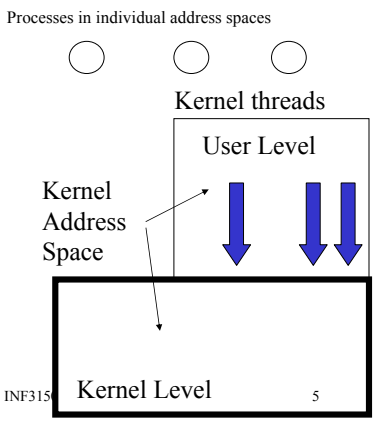
- I/O devices
  - Overlap I/Os with I/Os and computation (modern OS approach)
- Human users
  - Doing multiple things to the machine: Web browser
- Distributed systems
  - Client/server computing: NFS file server
- Multiprocessors
  - Multiple CPUs sharing the same memory: parallel program

Threads and Processes

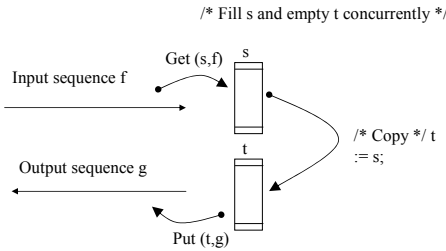
Trad. Threads



Project OpSys



Concurrency: Double buffering

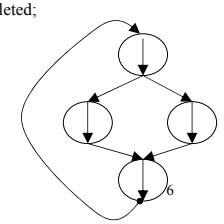



```
Get(s,f);
Repeat
    Copy;
    cobegin
        Put(t,g);
        Get(s,f);
    coend;
until completed;
```

Specifies concurrent execution

(Threads)

- **Put and Get** are disjunct
- … but not with regards to **Copy**!



Concurrency: Time Dependent Errors

```
Repeat
    Copy;
    cobegin
        Put(t,g);
        Get(s,f);
    coend;
until completed;
```

Oops!

```
Repeat
    cobegin
        Copy;
        Put(t,g);
        Get(s,f);
    coend;
until completed;
```

The rightmost (incorrect) solution can be executed in 6 ways:

- C-P-G
- C-G-P
- P-C-G
- P-G-C
- G-C-P
- G-P-C

Interleaving!

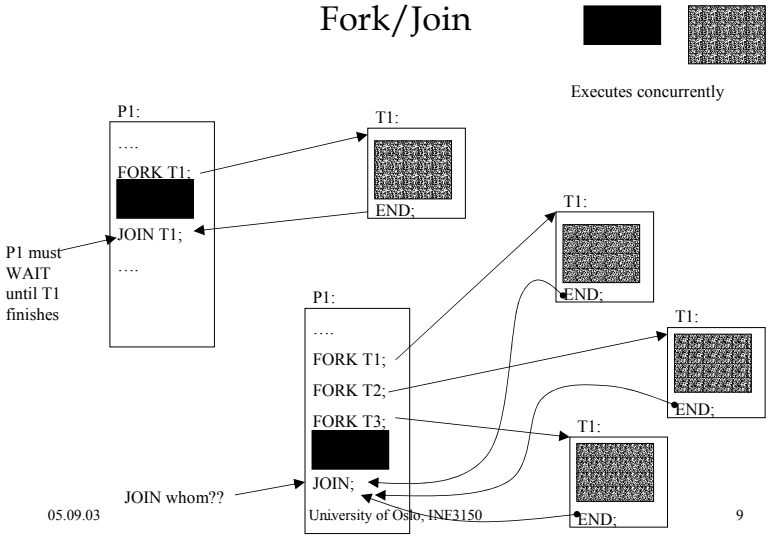
In the correct solution we solved the problem of sharing of the buffers between Copy and Put/Get by designing an algorithm avoiding problems

Typical Thread API

- Creation
  - Fork, Join
- Mutual exclusion
  - Acquire (lock), Release (unlock)
- Condition variables
  - Wait, Signal, Broadcast
- Alert
  - Alert, AlertWait, TestAlert

- Difficult to use
- Not good: Combines **specification** of concurrency (Fork) with **synchronization** (Join)

Fork/Join



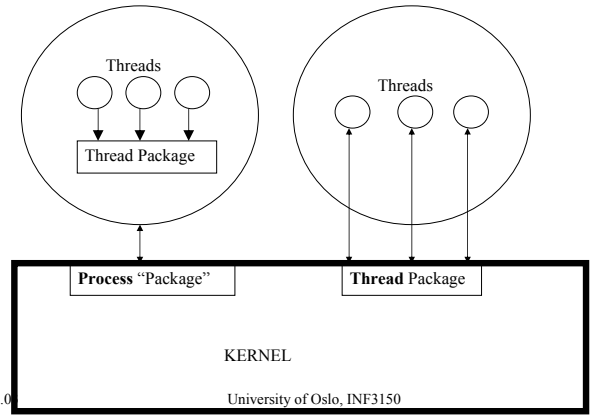
Concurrent programming w/ threads

- "Lighter" than processes
  - Easy and efficient sharing of data (why?)
  - Thread switches cheaper than context switches
- Easier, more general, more scalable structure than I/O multiplexing

User vs. Kernel-Level Threads

- Question
  - What is the difference between user-level and kernel-level threads?
- Discussions
  - When a user-level thread is blocked on an I/O event, the whole process is blocked
  - A context switch of kernel-threads is expensive
  - A smart scheduler (two-level) can avoid both drawbacks

User vs. Kernel Threads



Thread Control Block

- Shared information
  - Processor info: parent process, time, etc
  - Memory: segments, page table, and stats, etc
  - I/O and file: comm ports, directories and file descriptors, etc
- Private state
  - State (ready, running and blocked)
  - Registers
  - Program counter
  - Execution stack

System Stack for Kernel Threads

- Each kernel thread has
  - a user stack
  - a private kernel stack
- Pros
  - concurrent accesses to system services
  - works on a multiprocessor
- Cons
  - More memory

- Each kernel thread has
  - a user stack
  - a shared kernel stack with other threads in the same address space
- Pros
  - less memory
- Cons
  - serial access to system services

Typical for all shared resources

"Too Much Milk" Problem

Person A	Person B
Look in fridge: out of milk	
Leave for Shop	
Arrive at Shop	Look in fridge: out of milk
Buy milk	Leave for Shop
Arrive home	Arrive at Shop
	Buy milk
	Arrive home

- Don't buy too much milk
- Any person can be distracted at any point

A Possible Solution?

A:

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

B:

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

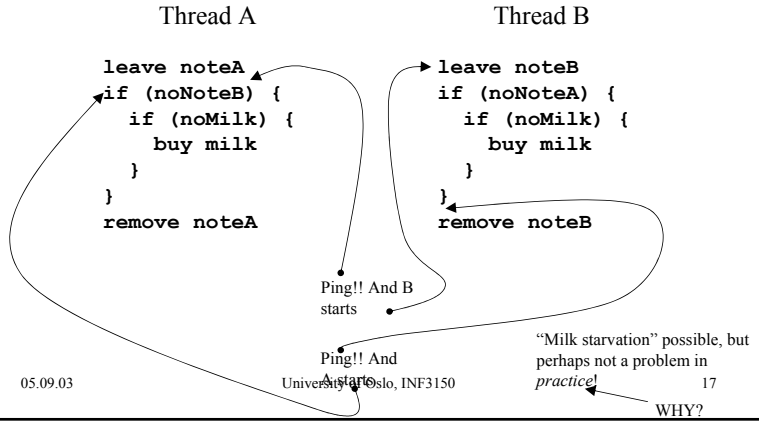
Ping!!!: and B starts executing until finished, and then A starts again

The ENTRY is flawed

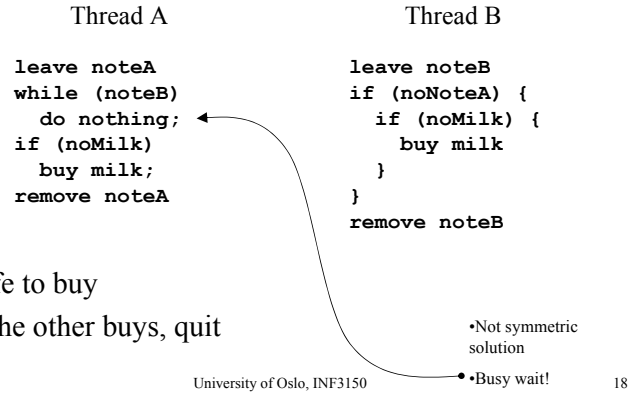
And both A and B buys milk.

(But B will "see" A by the fridge?: That is what we are trying to achieve.)

Another Possible Solution?



Yet Another Possible Solution?



- Safe to buy
- If the other buys, quit

•Not symmetric solution

•Busy wait!

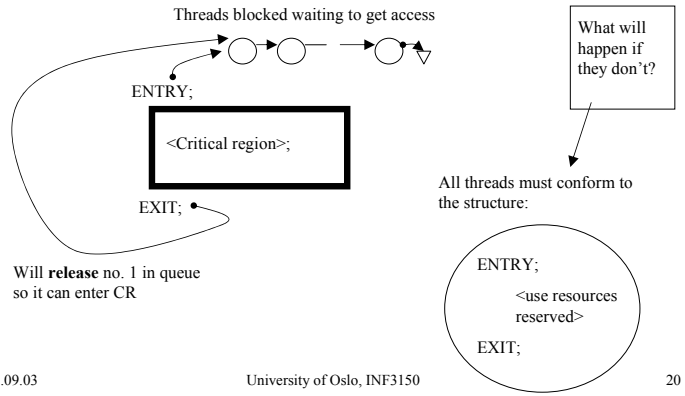
Remarks

- The last solution works, but
  - Life is too complicated
  - A's code is different from B's
  - Busy waiting is a waste
- Peterson's solution is also complex
- What we want is:

```
Acquire(lock);
if (noMilk)
  buy milk;
Release(lock);
```

Critical section a.k.a.
Critical region a.k.a.
Mutual Exclusion
(Mutex)

Entry and Exit Protocols



Critical Regions
Four Requirements for Good Solution

1. **Mutual Exclusion**: No two processes may be simultaneously inside their critical regions
2. No assumptions may be made about speeds or the number of CPUs
3. **Progress**: No process running outside its critical region may block other processes from entering their critical region
4. **Bounded Waiting**: No process should have to wait forever to enter its critical region

Solutions w/ Busy Waiting

- Busy waiting
  - Historically considered bad *(Why?)*
  - Is it always?
- Disabling interrupts
  - Cannot allow user processes to disable interrupts
  - Affects only single CPU
  - Useful for OS
- Spin locks
- Peterson
- TSL Instruction

Thread Safety

- Function is thread safe if and only if it will always produce correct results when called repeatedly from multiple concurrent threads
- Classes of thread-unsafe function:
  1) Functions that do not protect shared variables
  2) Function that keep state across multiple invocations
  3) Functions that return a pointer to a static variable
  4) Functions that call thread-unsafe functions

Reentrancy

- Reentrant functions are functions that do not reference any shared data
- Proper subset of thread-safe functions