Threads and Critical Sections

Thomas Plagemann

Slides from Otto J. Anshus, Tore Larsen (University of Tromsø), Kai Li (Princeton University)

Thread and Address Space

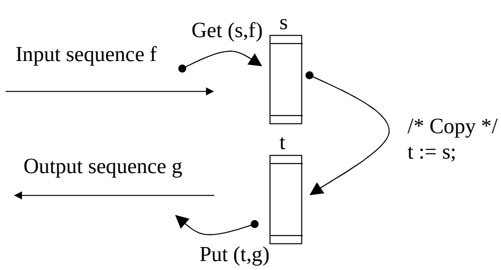
- Thread
  - A sequential execution stream within a process (also called lightweight process)
- Address space
  - All the state needed to run a program
  - Provide illusion that program is running on its own machine (protection)
  - There can be more than one thread per address space

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

Concurrency: Double buffering

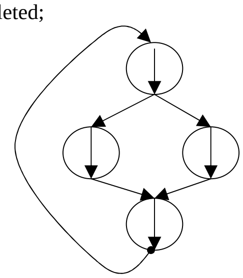
/* Fill s and empty t concurrently */




```
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;
```

```
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!

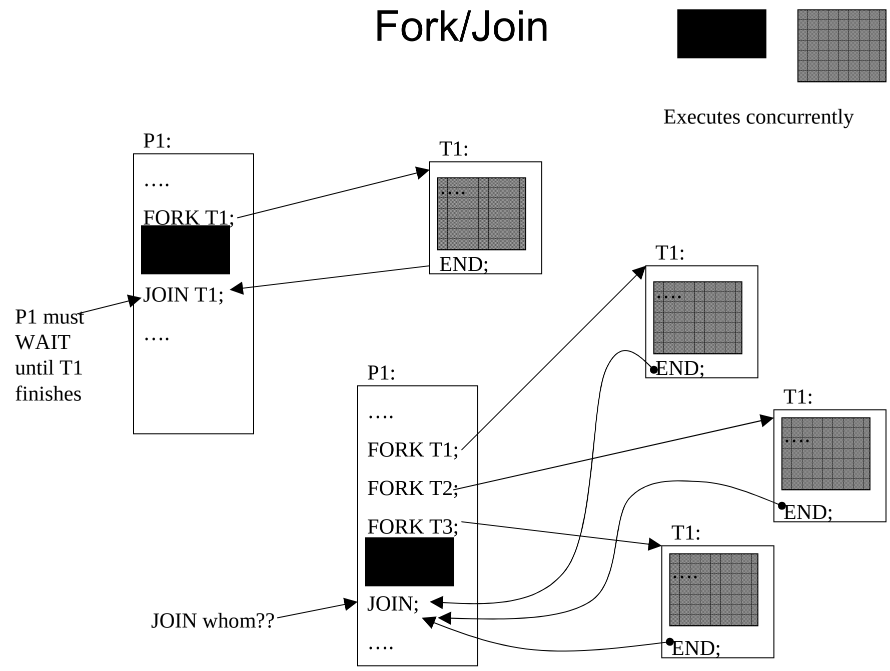
In the correct solution we solved the problem of sharing of the buffers between Copy and Put/Get by designing an algorithm avoiding problems

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)

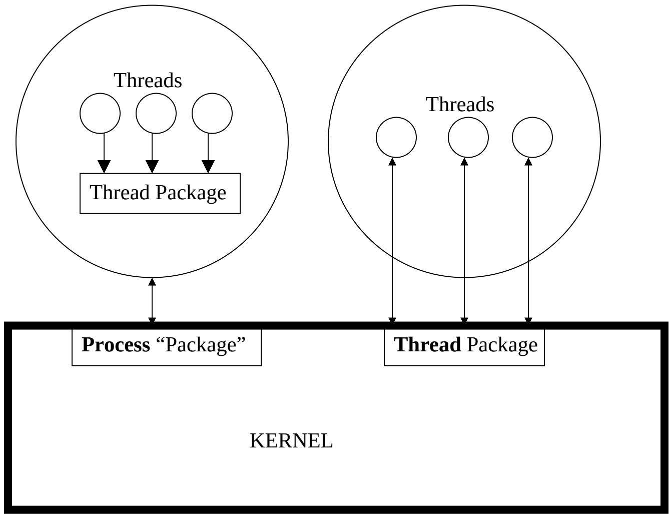
Fork/Join



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



Recall last week: PCB resp. PT

- Which information has to be stored/saved for a process?

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 Wawa	
Arrive at Wawa	Look in fridge: out of milk
Buy milk	Leave for Wawa
Arrive home	Arrive at Wawa
	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;
  }
}
```

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?

Thread A

```
leave noteA
if (noNoteB) {
  if (noMilk) {
    buy milk
  }
}
remove noteA
```

Thread B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

Another Possible Solution?

Thread A	Thread B

```
leave noteA                 leave noteB
if (noNoteB) {              if (noNoteA) {
  if (noMilk) {               if (noMilk) {
    buy milk                    buy milk
  }                           }
}                           }
remove noteA                remove noteB
```

Ping!! And B starts

Ping!! And A starts

"Milk starvation" possible, but perhaps not a problem in *practice*!

WHY?

Yet Another Possible Solution?

Thread A	Thread B

```
leave noteA                 leave noteB
while (noteB)               if (noNoteA) {
  do nothing;                 if (noMilk) {
if (noMilk)                     buy milk
  buy milk;                   }
remove noteA                }
                            remove noteB
```

Yet Another Possible Solution?

Thread A

```
leave noteA
while (noteB)
  do nothing;
if (noMilk)
  buy milk;
remove noteA
```

Thread B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

- 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)

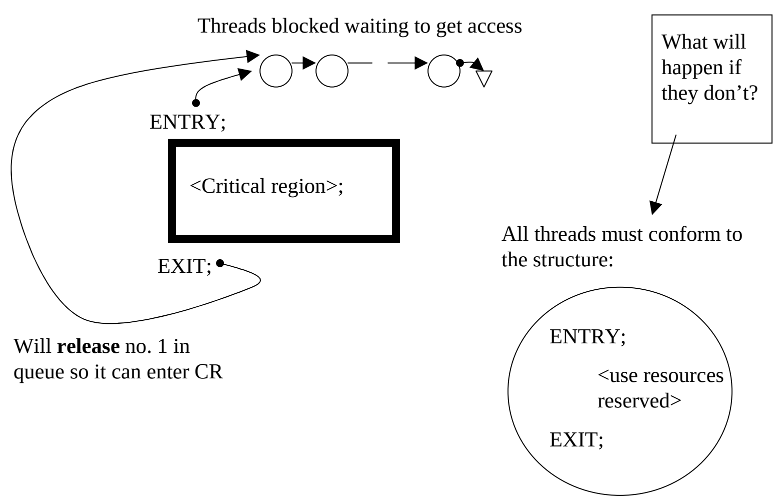
Entry and Exit Protocols

ENTRY;

<Critical region>;

EXIT;

Entry and Exit Protocols



Characteristics of a realistic solution for Mutual Exclusion

- Mutex: Only one process can be inside a critical region
- Non-preemptive scheduling of the resource: A thread having the resource must release it after a finite time
- No one waits forever: When the resource is requested by several threads concurrently, it must be given to one of them after a finite time
- No busy wait (?)
- Processes outside of critical section should not block other processes
- No assumption about relative speeds of each thread (time independence)
- Works for multiprocessors