4 Threads vs. Processes

A process includes

- an address space (defining all the code and data pages)
- a resource container (OS resource and accounting information)
- a "thread of control", which defines where the process is currently executing (basically, the PC, registers, and stack)

Communication between cooperating processes is relatively costly, because they don't share memory. Must communicate through

- shared files
- communication channels

That's fine for processes that interact at a "coarse grain" (e.g., "ps -aux | fgrep emacs | more"), but may be too costly/awkward for processes that have more complex interactions.

Idea: allow (mutually consenting) processes to share part or all of their memory. Processes can now interact efficiently (through shared memory), but each still has its own address space, set of OS resources, and accounting information. All modern operating systems support this in some way.

Sometimes, closely cooperating processes (e.g., processes inside the OS kernel) share *all* of their memory, and use the same OS resources. In this case, there would be a lot of duplication and unnecessary overhead if conventional processes were used.

- creating a process is relatively costly, because of all the structures that must be allocated and initialized.
- context switching is more complex than necessary, since there is no real need for separate address spaces.

What's similar in these processes?

- they share the same code and data (address space)
- they use the same resources (files, communication channels)

What don't they share?

- each has its own PC, registers, stack pointer

Idea: separate the concepts of a "thread of control" (PC, SP, registers) from the rest of the process (address space, resources, accounting, etc.) Many modern operating systems therefore support two entities:

- the *task* (process, actor, heavyweight process), which defines an address space, a resource container, accounting information
- the *thread* (lightweight process) which defines a single sequential execution stream within a process

There may be several threads executing in a single address space. Threads are the unit of scheduling; tasks are containers in which threads execute.

In this refined model, a conventional process consists of a task with a single thread of control.

Programming with threads is very flexible, but error-prone. There is no protection between threads. In C/C++

- automatic variables are private to each thread
- global variables and dynamically allocated memory (malloc) are shared among threads.

5 Synchronization: The Too-Much-Milk Problem

Synchronization: using atomic operations to ensure correct operation of cooperating processes.

The "too much milk" problem:

Person A	**Person B**
3:00 Look in fridge. Out of milk.	
3:05 Leave for store.	
3:10 Arrive at store.	Look in fridge. Out of milk.
3:15 Leave store.	Leave for store.
3:20 Arrive home, put milk away.	Arrive at store.
3:25	Leave store.
3:30	Arrive home. *OH, NO!*

What does correct mean? Somebody gets milk, but we don't get too much milk. One of the most important things in synchronization is to figure out what you want to achieve.

Mutual exclusion: Mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded). E.g. only one person goes shopping at a time.

Critical section: A section of code, or collection of operations, in which only one process may be executing at a given time. E.g. shopping. It's a large operation that we want to make "sort of" atomic.

There are many ways to achieve mutual exclusion. Most involve some sort of *locking* mechanism: prevent someone from doing something. Primitives: `lock(l)` and `unlock(l)` delimit critical section. For the fridge analogy: before shopping, leave a note on the refrigerator.

Three elements of locking:

1. Must lock before using. *leave note*
2. Must unlock when done. *remove note*
3. Must wait if locked. *don't shop if note*

1st attempt at computerized milk buying:

```
     Processes A & B
1    if (NoMilk) {
2         if (NoNote) {
3              Leave Note;
4              Buy Milk;
5              Remove Note;
6         }
7    }
```

- This doesn't always work: A1 A2 B1 B2 B3 B4 B5 B6 ...
- This solution works for people because lines 1-3 are performed atomically: you'll see the other person at the refrigerator and make arrangements. Typically, computers don't both test (look for other person) and set (leave note) at the same time.
- In this case, we haven't eliminated the problem; we've just moved it and made it a little less likely (i.e. a little more insidious). This is typical of first attempts at solutions to synchronization problems.

What happens if we leave the note at the very beginning: does this make everything work?

2nd attempt: change meaning of note. A buys if there's no note, B buys if there is a note. This gets rid of confusion.

```
     Process A
1    if (NoNote) {
2         if (NoMilk) {
3              Buy Milk;
4         }
5         Leave Note;
6    }
```

Process B

```
1   if (Note) {
2       if (NoMilk) {
3           Buy Milk;
4       }
5       Remove Note;
6   }
```

- Does this work?
- How can you tell?
- Ideally, we shouldn't rely on intuitions or informal reasoning when dealing with complex parallel programs: we should be able to *prove* that they behave correctly. Unfortunately, formal verification has only been successful on very small programs (too hard to do). For example, in the above example:
  - A note will be left only by A, and only if there isn't already a note.
  - A note will be removed only by B, and only if there is a note.
  - Thus, there is either one note, or no note.
  - If there is a note, only B will buy milk.
  - If there is not a note, only A will buy milk.
  - Thus, only one process will buy milk.
- Suppose B goes on vacation. A will buy milk once and won't buy any more until B returns. Thus this really doesn't really do what we want; it's unfair, and leads to starvation.
- In other words, can't distinguish "You're buying" from "I'm not buying." Not enough information.

3rd attempt: use 2 notes.

Process A

```
1   Leave NoteA;
```

```
2    if (NoNoteB) {
3         if (NoMilk) {
4              Buy Milk;
5         }
6    }
7    Remove NoteA;
```

Process B is the same except interchange NoteA and NoteB.

What can you say about this solution? (Look at the boundary between statements 1 and 2)

- At most one process will buy milk: each process leaves note before it checks.
- If one process goes on vacation after step 7, the other process will still buy milk.
- Suppose both A and B leave notes at exactly the same time: nobody will buy milk (there is still starvation).

Solution is almost correct. We just need a way to decide who will buy milk when both leave notes (somebody has to hang around to make sure that the job gets done).

4th attempt: in case of tie, B will buy milk. Process A stays the same as before.

```
     Process B
1    Leave NoteB;
2    while (NoteA) DoNothing;
3    if (NoMilk) {
4         Buy Milk;
5    }
6    Remove NoteB;
```

This solution works. But it still has two disadvantages:

- Asymmetric (and complex) code.
- While B is waiting it is consuming resources (busy-waiting).

6 Synchronization with Semaphores

The too-much-milk solution is much too complicated. The problem is that the mutual exclusion mechanism was too simple-minded: it used only atomic reads and writes. This is sufficient, but unpleasant. It would be unbearable to extend that mechanism to many processes. Let's look at more powerful, higher-level mechanisms.

Requirements for a mutual exclusion mechanism:

- Must allow only one process into a critical section at a time.
- If several requests at once, must allow one process to proceed.
- Processes must be able to go on vacation outside critical section.

Desirable properties for a mutual exclusion mechanism:

- Fair: if several processes waiting, let each in eventually.
- Efficient: don't use up substantial amounts of resources when waiting. E.g. no busy waiting.
- Simple: should be easy to use (e.g. just bracket the critical sections).

Desirable properties of processes using the mechanism:

- Always lock before manipulating shared data.
- Always unlock after manipulating shared data.
- Do not lock again if already locked.
- Do not unlock if not locked by you (usually: there are a few exceptions to this).
- Do not spend large amounts of time in critical section.

Semaphore: A synchronization variable that takes on positive integer values. Invented by Edsger Dijkstra in the mid 60's.

- P(semaphore): an atomic operation that waits for semaphore to become greater than zero, then decrements it by 1 ("proberen" in Dutch).
- V(semaphore): an atomic operation that increments semaphore by 1 ("verhogen" in Dutch).

Semaphores are simple and elegant and allow the solution of many interesting problems. They do a lot more than just mutual exclusion.

Too much milk problem with semaphores:

```
     Processes A & B
1    P(OKToBuyMilk);
2    if (NoMilk) {
3         Buy Milk;
4    }
7    V(OKToBuyMilk);
```

Note: OKToBuyMilk must initially be set to 1. What happens if it isn't?

Show why there can never be more than one process buying milk at once.

Binary semaphores are those that can only take on two values, 0 and 1.

Semaphores aren't provided by hardware. (I'll describe implementation later.) But they have several attractive properties:

- Machine independent.
- Simple.
- Work with many processes.
- Can have many different critical sections with different semaphores.
- Can acquire many resources simultaneously (multiple P's).

- Can permit multiple processes into the critical section at once, if that is desirable.

Desirability of layering: picking powerful and flexible intermediate solutions to problems. A synchronization kernel is appropriate for one layer.

Semaphores are used in two different ways:

- Mutual exclusion : to ensure that only one process is accessing shared information at a time. If there are separate groups of data that can be accessed independently, there may be separate semaphores, one for each group of data. These semaphores are always binary semaphores.
- Condition synchronization : to permit processes to wait for certain things to happen. If there are different groups of processes waiting for different things to happen, there will usually be a different semaphore for each group of processes. These semaphores aren't necessarily binary semaphores.

Semaphore Example: Producer & Consumer. Suppose one process is creating information that is going to be used by another process, e.g. suppose one process is reading information from the disk, and another process will compile that information from soure code to binary. Processes shouldn't have to operate in perfect lock-step: producer should be able to get ahead of consumer.

- Producer: creates copies of a resource.
- Consumer: uses up (destroys) copies of a resource. (may produce something else)
- Buffers: used to hold information after producer has created it but before consumer has used it.
- Synchronization: keeping producer and consumer in step.
- Define constraints (definition of what is "correct"). Note importance of doing this before coding.
  - Consumer must wait for producer to fill buffers. (condition synchronization)

  - Producer must wait for consumer to empty buffers, if all buffer space is in use. (condition synchronization)
  - Only one process must manipulate buffer pool at once. (mutual exclusion)

- A separate semaphore is used for each constraint. Explain the three semaphores, what they mean, who P's and who V's.

- Initialization:
  - Put all buffers in pool of empties.
  - Initialize semaphores: empties = numBuffers, fulls= 0, mutex = 1;

- Producer process:

```
P(empties);
P(mutex);
get empty buffer from pool of empties;
V(mutex);
produce data in buffer;
P(mutex);
add full buffer to pool of fulls;
V(mutex);
V(fulls);
```

- Consumer process:

```
P(fulls);
P(mutex);
get full buffer from pool of fulls;
V(mutex);
consume data in buffer;
P(mutex);
add empty buffer to pool of empties;
V(mutex);
V(empties);
```

- Important questions:
  - Why does producer P(empties) but V(fulls)? Explain in terms of creating and destroying resources.
  - Why is order of P's important? Deadlock (deadly embrace).
  - Is order of V's important?
  - Could we have separate mutex semaphores for each pool?
  - How would this be extended to have 2 consumers?

Producers and consumers produces something much like Unix pipes.

THIS IS AN IMPORTANT EXAMPLE! Go over the two classes of semaphore usage again: mutual exclusion and scheduling.

Another example of semaphore usage: a shared database with readers and writers. It is safe for any number of readers to access the database simultaneously, but each writer must have exclusive access. Must use semaphores to enforce these policies. Example: checking account (statement-generators are readers, tellers are writers).

- Writers are actually readers too.
- Constraints:
  - Writers can only proceed if there are no active readers or writers (use semaphore OKToWrite).
  - Readers can only proceed if there are no active or waiting writers (use semaphore OKToRead).
  - To keep track of who's reading and writing, need some shared variables. These are called *state variables*. However, must make sure that only one process manipulates state variables at once (use semaphore Mutex).
- State variables:
  - AR = number of active readers.
  - WR = number of waiting readers.

  - AW = number of active writers.
  - WW = number of waiting writers.

  AW is always 0 or 1.

  AR and AW may not both be non-zero.

- Initialization:
  - OKToRead = 0; OKToWrite = 0; Mutex = 1;
  - AR = WR = AW = WW = 0;
- Scheduling: writers get preference.

Reader Process:

```
P(Mutex);
if ((AW+WW) == 0) {
      V(OKToRead);
      AR = AR+1;
} else {
      WR = WR+1;
}
V(Mutex);
P(OKToRead);
– read the necessary data;
P(Mutex);
AR = AR-1;
if (AR==0 && WW>0) {
      V(OKToWrite);
      AW = AW+1;
      WW = WW-1;
}
V(Mutex);
```

Writer Process:

```
P(Mutex);
if ((AW+AR+WW) == 0) {
      V(OKToWrite);
      AW = AW+1;
} else {
      WW = WW+1;
}
V(Mutex);
P(OKToWrite);
– write the necessary data;
P(Mutex);
AW = AW-1;
if (WW>0) {
      V(OKToWrite);
      AW = AW+1;
      WW = WW-1;
} else while (WR>0) {
      V(OKToRead);
      AR = AR+1;
      WR = WR-1;
}
V(Mutex);
```

Go through several examples: (tell what happens)

- Reader enters and leaves system.
- Writer enters and leaves system.
- Two readers enter system.
- Writer enters system and waits.
- Reader enters system and waits.
- Readers leave system, writer continues.

- Writer leaves system, last reader continues and leaves.

Questions:

- In case of conflict between readers and writers, who gets priority? Readers can get locked out.
- Is the WW necessary in the writer's first if? No: if there is a waiting writer, then there must be an active reader or an active writer.
- Can OKToRead ever get greater than 1? What about OKToWrite?
- Is the first writer to execute P(Mutex) guaranteed to be the first writer to access the data?

7 Semaphore Implementation

Need a simple way of doing mutual exclusion in order to implement P's and V's. We could use atomic reads and writes, as in the "too much milk" problem, but these are very clumsy.
Furthermore, the busy wait in the "too much milk" implementation (i.e., the while loop polling the note variable) is wasting CPU cycles. This can be avoided if we implement P and V as system calls that block the threads that need to wait for the semaphore to be greater than zero upon invoking P (by changing their state to "Waiting"), and wake up threads that were blocked in a semaphore upon another thread invoking V (by changing their state to "Ready").

Uniprocessor solution: disable interrupts. Recall that the only way the dispatcher regains control is through interrupts or through explicit requests.

```
typedef struct {
    int count;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;

void P(Semaphore s)
{
    Disable interrupts;
    if (s->count > 0) {
        s->count -= 1;
        Enable interrupts;
        return;
    }
    Add(s->q, current_thread);
    sleep(); /* re-dispatch */
    Enable interrupts;
}
```

```
void V(Semaphore s)
{
    Disable interrupts;
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread); /* put thread on the ready queue */
    }
    Enable interrupts;
}
```

What do we do in a multiprocessor to implement P's and V's? Can't just turn off interrupts to get low-level mutual exclusion.

- Turn off all other processors?
- Use atomic read and write, as in "too much milk"?

In a multiprocessor, there will have to be busy-waiting at some level: can't go to sleep if don't have mutual exclusion.

Most CISC machines provide some sort of atomic *read-modify-write* instruction. Read existing value, store back in one atomic operation. E.g. Test and set (oldval=TAS(addr, newval)).

Using TAS to build a spin lock: 1 means someone holds the lock, 0 means it's OK to proceed. Definition of TAS prevents two threads from getting a 0-to-1 transition (e.g., lock acquisition) simultaneously.

```
int spin_lock = 0;
..
while (TAS(&spin_lock, 1) != 0);
..
critical section
..
spin_lock = 0;
```

Modern RISC machines don't provide read-modify-write instructions. Instead, most of them provide a weaker mechanism that does not guarantee atomicity, but detects interference.

- *load-linked* instruction (ldl): Loads a word from memory and sets a per-processor flag associated with that word (usually stored in the cache).
- store operations to the same memory location (by any processor) reset all processor's flags associated with that word.
- *store-conditionally* instruction (stc): Stores a word iff the processor's flag for the word is still set; indicates success or failure.

Using ldl/stc for mutual exclusion:

```
int spin_lock=0;
..
while (ldl(&spin_lock) != 0 || !stc(&spin_lock, 1));
..
critical section
..
spin_lock = 0;
```

Using ldl/stc to implement semaphores in a multiprocessor: For each semaphore, keep an integer (lock) in addition to the semaphore integer and the queue of waiting threads.

```
typedef struct {
    int spin_lock; /* initially 0 */
    int count;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;

void P(Semaphore s)
{
    Disable interrupts;
```

```
    while (ldl(s->spin_lock) != 0 || !stc(s->spin_lock, 1));
    if (s->count > 0) {
        s->count -= 1;
        s->spin_lock = 0;
        Enable interrupts;
        return;
    }
    Add(s->q, current_thread);
    Remove(ReadyQueue, current_thread);
    SetState(current_thread, WAITING);
    s->spin_lock = 0;
    dispatch(); /* switch to first thread on the ready queue */
    Enable interrupts;
}
```

```
void V(Semaphore s)
{
    Disable interrupts;
    while (ldl(s->spin_lock) != 0 || !stc(s->spin_lock, 1));
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        Thread t = RemoveFirst(s->q);
        SetState(current_thread, READY);
        Add(ReadyQueue, current_thread);
    }
    s->spin_lock = 0;
    Enable interrupts;
}
```

Why do we still have to disable interrupts in addition to using the spin_lock with ldl/stc?

Important point: implement some mechanism once, very carefully. Then always write programs that use that mechanism. Layering is very important.