Concurrency

"Concurrency" describes having simultaneous computational agents (e.g. processes).

The interaction between computational agents raises many issues.

Suppose two different processes each want to modify a file in a certain way:

Read the file into memory (variables), add something to the data (e.g. a record in sorted order), write it back to the file.

Possible sequences:

- First process reads, modifies, writes; then second process reads, modifies, writes. (good)
- First reads, second reads, each modify in memory, first writes, second overwrites ("wins"!) (bad).
- Interleaved writing...

Interleaved writing:

- First process does fopen(..., "w") (truncates)
- First process writes 10 records
- Second process does fopen(..., "w") (truncates, erasing 1's data)
- First process writes more data — automatically extends file with zeroes
- Second process writes data at the beginning, etc.

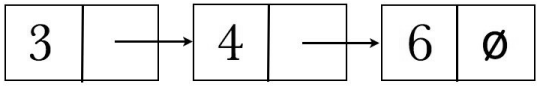
File may be pretty much destroyed.

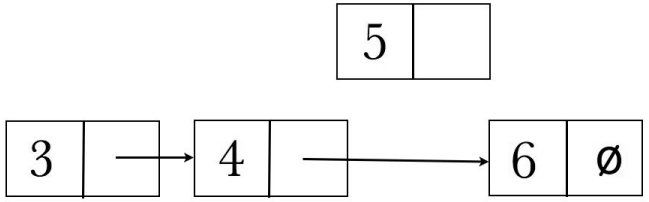
Concurrency problem demonstration:

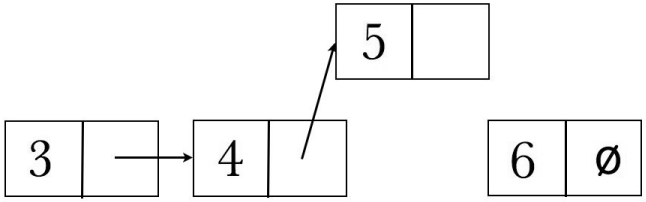
```
for i in 0 1 2 3 4 5 6 7 8 9
do
    x=`cat file`
    expr $x + 1 >file
done
```

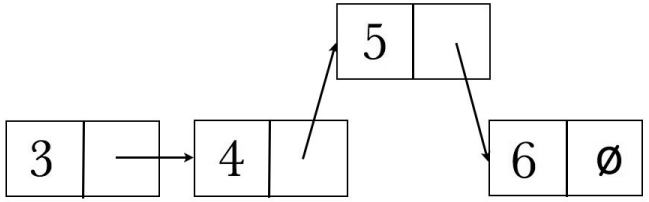
```
echo 0 >file
sh addten
cat file
sh addten & sh addten
cat file
sh addten & sh addten & sh addten & sh addten
```

With “threads”, we have the spectre of concurrency problems even within a single process!









Possible solution:

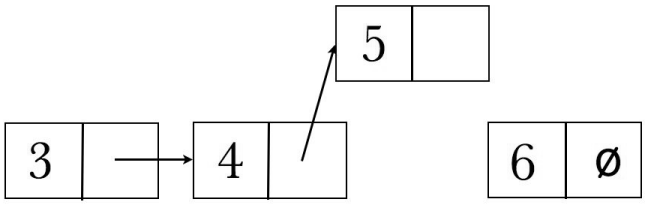
- When one process is editing the file, the other one doesn't even start.
  - "lock file": create lock file before starting; remove when it's ok for others to edit
  - Same problem, just a smaller window!
  - We need an "atomic test and set"...
  - Techniques for making an atomic test and set are highly architecture-specific.

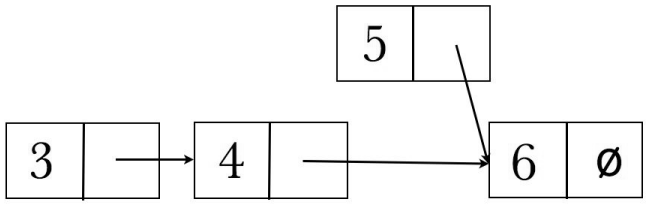
Dijkstra's P and V:

```
int lockvar = 1;
... P(&lockvar) ... V(&lockvar) ...

def P(int *lockp):
      wait until *lockp > 0, then (*lockp)--, atomically
      (no possibility of other processes or threads modifying
      *lockp between the test and the decrement)

def V(int *lockp):
      (*lockp)++;   (atomically)
```

```
whatever malloc(whatever)
{
    static int arenalock = 1;   (shared variable)

    ...
    P(&arenalock);
    do the linked-list insertion
    V(&arenalock);
    ...
}
```

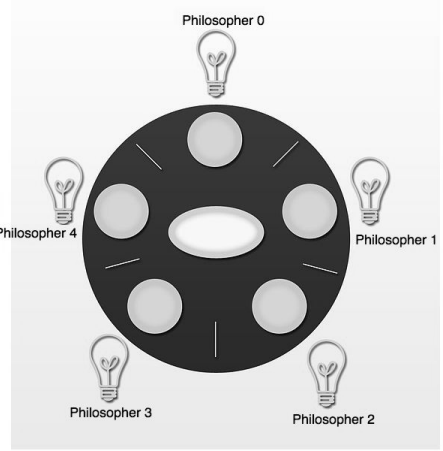
"Race condition": a situation where small details of timing create very different results.





Also, P needs to **block** rather than to "busy-wait".

The "Dining Philosophers" problem




```
void diningphilosopher(int i)  /* 0<=i<5 */
{
    while (1) {
        /* think */
        thinking occurs here

        /* eat */
        P(&chopstick[i]);
        P(&chopstick[(i+1)%5]);
        eating occurs here
        V(&chopstick[i]);
        V(&chopstick[(i+1)%5]);
    }
}
```

Deadlock:

∃ a chain of processes $p_0, p_1, p_2, \ldots, p_{n-1}$ such that ∀i, p_i has some resource which $p_{(i+1)\%n}$ is blocked waiting on.

(n is often 2)

Attempted solution:

Have all of the even philosophers pick up their left chopstick first; have all of the odd philosophers pick up their right chopstick first.

Would only work with an even number of philosophers.

Attempted solution #2:

Pick up both chopsticks simultaneously.

(Ok, but what if one of them is unavailable? Degenerates to either the previous case or the next case:)

Attempted solution #3:

First pick up the left chopstick (in a blocking way);

Then: if the right chopstick is available, pick it up (atomic test and set); else put down the left chopstick and loop.

But this constitutes "busy-waiting"; and involves lots of picking up and putting down the left chopstick to no avail.

Can we do better?

Dijkstra's solution:

All philosophers pick up their left chopstick first *except* the highest-number philosopher, who picks up their right chopstick first.

Avoids deadlock with no busy-waiting and no waste!

Rough proof that it avoids deadlock: For the deadlock definition, we need a cycle of blocked processes which each have something the previous one needs, and need something the subsequent one has. But if you're blocked holding one chopstick, the chopstick you have is always a lower-numbered chopstick than the one you need.

Generalization of Dijkstra's solution:

- Number all of the contentious resources.
- You must always acquire all needed resources (for a particular operation) in order from lowest-numbered to highest-numbered.
- Note how the dining philosophers solution follows the numbering of the chopsticks.
- Can result in having to release and re-acquire some needed resources.
- Obviously, you choose your numbering scheme to minimize this.

Another issue:

Consider the assignment four server:

```
while (1) {
    ... select ...
    for (p = clientlist; p && !isdataready(p); p = p->next)
        ;
    if (p)
        clientactivity(p);
}
```

(the "clientactivity" call is outside the loop because it might delete from the linked list)

"**starvation**": the case where some clients later in the list *never* get dealt with because an earlier one always has something

Solution:

Deal with clients in "round-robin" fashion even though it's trickier:

```
p = NULL;
while (1) {
    ... select ...
    if (p == NULL)
        p = clientlist;
    for (; p && !isdataready(p); p = p->next)
        ;
    if (p) {
        q = p;
        p = p->next;
        clientactivity(q);
    }
}
```

- "concurrent programming language"
- or library functions

"semaphore": a variable upon which you perform Dijkstra's P and V.

- When you call V on the semaphore, in addition to releasing the semaphore, if someone else is blocked waiting on it it wakes them up — called a "signal".

```
critical {
   ...
}
```

```
class whatever {
   ...
   int x, y;
   ...

   synchronized void foo() {
      int t = x;
      x = y;
      y = t;
   }

}
```