NTNU | Norwegian University of Science and Technology

Operating Systems

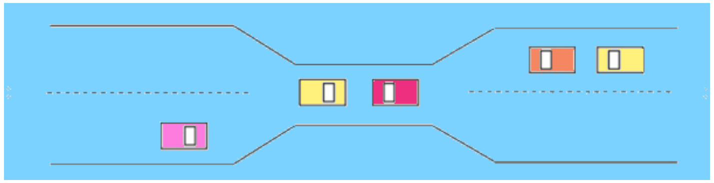
Example solutions for Theoretical Exercise 3

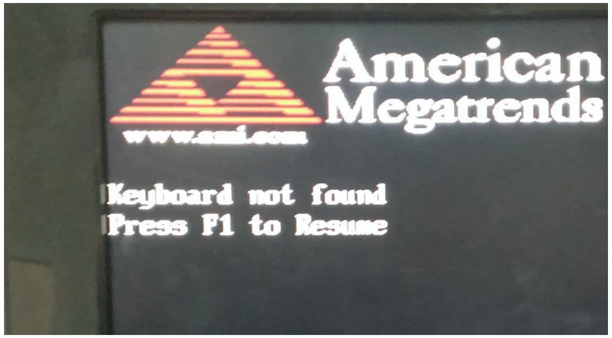
Michael Engel

3.1 Deadlocks in real life

We have seen the crossroads example to demonstrate the problem in the lecture. List three other examples of deadlocks that are not related to a computer system environment.

- Another traffic example: a one-lane bridge in which a car from both directions has entered. It can be resolved if one car backs up



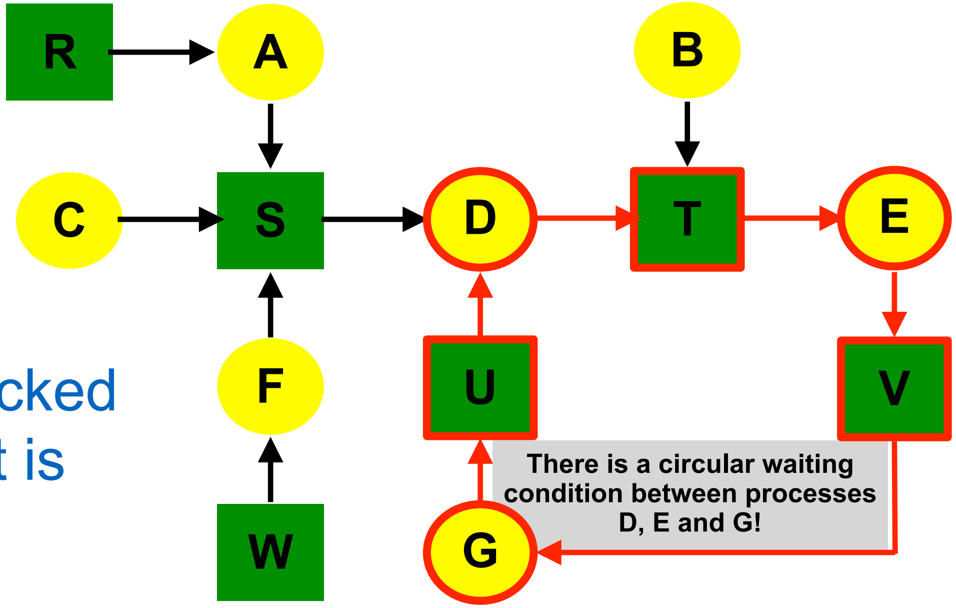


- A fun effect on some older PCs
- Problems of an introvert:
  "I don't really like to talk to someone before I get comfortable with them.
  I don't get comfortable with someone before I've been talking with them for a while."

3.2 Resource allocation graphs

Suppose that there is a resource deadlock in a system. Give an example to show that the set of processes deadlocked can include processes that are not in the circular chain in the corresponding resource allocation graph.

- This is shown in the example from lecture 7
  - Here we have a chain of processes D, E, G forming a chain in the resource graph
  - However, process **B** is also waiting for one of the deadlocked resources (T), even though it is not in a circular dependency



3.3 Deadlock conditions

Two processes, A and B, each need three records, 1, 2, and 3, in a database.
If both A and B request the records in the order 1, 2, 3, deadlock is not possible. However, if B asks for the records in the order 3, 2, 1, then a deadlock can occur.
With three resources, there are 3! = 6 possible combinations each process can request resources.
What fraction of all combinations is guaranteed to be deadlock free?

- Suppose that process A requests the records in the order 1, 2, 3. If process B also asks for 1 first, one of them will get it and the other will block.
- This situation is always deadlock free since the winner can now run to completion without interference.

3.3 Deadlock conditions

…What fraction of all combinations is guaranteed to be deadlock free?

- The other four combinations can be similarly reasoned about and shown to lead to possible deadlock:

  (1) 1 2 3: deadlock free
  (2) 1 3 2: deadlock free
  (3) 2 1 3: possible deadlock
  (4) 2 3 1: possible deadlock
  (5) 3 1 2: possible deadlock
  (6) 3 2 1: possible deadlock

  - So only one third of the cases are guaranteed to be deadlock free

3.4 Banker's algorithm

Consider a system that uses the banker's algorithm to avoid deadlocks. At some time a process P requests a resource R, but is denied even though R is currently available. Does it mean that if the system allocated R to P, the system would deadlock?

- No.
- An available resource is denied to a requesting process in a system using the banker's algorithm if there is a possibility that the system may deadlock by granting that request.
- It is certainly possible that the system may not have deadlocked if that request was granted.

3.5 C preprocessor

You want to define a C preprocessor macro to calculate the square of a given number x as follows:

```
#define SQUARE(x) (x * x)
```

Explain what is problematic with this macro definition and give an example of the problematic behavior.

- The C preprocessor performs only ***syntactic text expansion*** of macros, it does not know/understand C syntax or semantics!
- The parameter x is thus replaced by whatever is given as parameter to the macro invocation, e.g.
  `SQUARE(1+2)`
  (which you would expect to be 3*3 = 9) is expanded to
  `(1+2 * 1+2) = 1+2+2 = 5` (due to arithmetic precedence rules in C)
- There are many more ***macro pitfalls***, see e.g. https://gcc.gnu.org/onlinedocs/gcc-3.4.6/cpp/Macro-Pitfalls.html for details

3.6 ELF Segments

You are trying to analyze a binary program using the command `readelf -S prog` and obtain the following output (shortened):

```
Section Headers:
  [Nr] Name       Type       Address            Offset
       Size                  EntSize            Flags     Link  Info  Align
  [25] .data      PROGBITS   0000000000004000   00003000
       0000000000000010      0000000000000000   WA        0     0     8
```

Assume that you know that there are only global `int` variables and each variable uses four bytes. Can you tell how many global int variables are declared in the program?

- The size of the segment is `0000000000000010` (hex) = 16 bytes
- Thus, it can hold a maximum of 16/4 = 4 `int` variables
- However, the ***alignment requirement*** is a multiple of 8
  - Thus, 3 `ints` would also take 16 bytes (12 + 4 bytes alignment)
- So we cannot say if there are 3 or 4 `int` variables declared
  - …without further investigation (e.g., using `nm`)

3.7 ELF Symbols

Consider the following (very simple and useless) C program:

```
int foo;
int bar;
int main(int argc, char **argv) {
  int a, b;
}
```

Which ELF segment will the variables `foo` and `bar` be located in?

- Both are global uninitialized (in the source code) variables, which are automatically initialized to 0 by the C runtime. They are thus ***not*** stored in the data segment, but in the bss segment
- This saves space in the executable, since variables with initial value 0 do not need to be saved in the executable
- A global variable `int baz=42;` would be stored in data, since its value has to be set when the program starts

3.7 ELF Symbols

Consider the following (very simple and useless) C program:

```
int foo;
int bar;
int main(int argc, char **argv) {
  int a, b;
}
```

When running the `nm` command on the binary compiled from the program, variables `a` and `b` are not shown in the command's output. Explain why.

- `a` and `b` are local variables of the function `main`. Local variables cannot have a fixed address in memory (like global variables in data and bss), since we need a *separate copy* of the variable in case of a recursive call to the function the variables are declared in
- Thus, these variables are stored on the *stack*, which grows (and shrinks) when entering (leaving) a function
- Yes, you can also call `main` recursively…