Operating Systems 2016/17
Tutorial-Assignment 8

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 8.1: Kernel Synchronization

a. On single-processor systems, why can mutual exclusion in the kernel be achieved by masking interrupts?

Solution:
In a single-processor system no real parallelism exists (i.e., no two threads ever run at the same time). Nonetheless, synchronization is required because with preemptive thread dispatching, a thread switch can occur any time (even if the thread runs in the kernel). This switch is usually triggered by a timer interrupt. By masking all interrupts, invocation of the scheduler by the timer interrupt and thus involuntary thread switches are avoided. Furthermore, it also prevents the execution of other interrupt handlers that possibly access the same data.

b. Why does this approach not work on multi-processor systems?

Solution:
Masking interrupts only affects a single CPU. To mask interrupts on all CPUs, one would have to send a notification to all other CPUs to have them mask interrupts, too. These notifications would have to be sent on every entry and exit of a critical section, which would be expensive.

Furthermore, even if one would do this, it would not suffice: masking of interrupts does not prevent the thread currently running on a different CPU to access the same shared data structures. It would only prevent that this thread gets interrupted by another activity.

c. How can mutual exclusion within the kernel be achieved on multi-processor systems?

Solution:
As simply disabling interrupts is not sufficient for multi-processor systems, one has to rely on synchronization mechanisms such as spinlocks instead. A simple approach is to have one "big" lock that protects the entire kernel. This approach was taken by early versions of the Linux kernel. The latest occurrences of the so-called Big Kernel Lock *(BKL) have been removed (and replaced by more fine-granular locking) only in Linux version 2.6.39.*

While the approach of a single lock is easy to implement (and can thus for example be used to port a single-processor operating system to multi-processor systems), it does not allow for any parallelism within the kernel and thus greatly limits the scalability of the entire system. Fine-grained locking is desirable instead, where separate locks are used for each data structure (or for subsets of data structures).

Fine-grained locks allows for more parallelism within the kernel, but also make the code more complex and error-prone: Programmers might not be aware of all the locks that are held, might forget to acquire or to release locks, or might acquire locks in the wrong order, thus causing deadlocks.

For some spinlocks interrupts must still be disabled on the same CPU. Consider the scenario where a thread runs in a critical section and is interrupted. The interrupt handler *wants to access the same data and would need to acquire the spinlock. However, in that case the CPU would spin in the handler forever, because it has been interrupted from the critical section, which is responsible to release the spinlock. A problem known as lock-holder-preemption.*

Question 8.2: Prerequisites for Deadlocks

a. Define the term deadlock and give some examples.

Solution:
A deadlock is a situation in which several activities (e.g., processes or threads) cannot make any progress, because each activity is waiting for a resource that is assigned to another activity, and no activity releases any of its resources. An example is a 4-way junction where the car on the right-hand side has the right of way. If four cars arrive simultaneously, all would have to wait for their right-hand neighbor (unless one driver waives his right of way).

b. Enumerate and explain the necessary conditions for deadlocks.

Solution:

Mutual exclusion (aka exclusiveness) *Resources cannot be shared between processes.*

Hold and wait *A process already holding a resource can wait to acquire more resources.*

No preemption *Resources cannot be taken away forcibly from processes.*

Circular wait *There exists a set of processes $\{P_0, \dots P_n\}$ where P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_n is waiting for a resource held by P_0.*

Note that these four conditions are not truly independent, as circular wait implies hold and wait.

Question 8.3: Searching for Deadlocks

Consider the following code fragment:

```
Spinlock s1, s2, s3 = FREE;
int counter = 0;

void Thread1() {
    if (counter == 0) {
        lock(s1);
        counter++;
        unlock(s1);
    }
    lock(s2);
    lock(s3);
    // update some more data
    unlock(s3);
    unlock(s2);
}
```

```
void Thread2() {
    lock(s3);
    counter++;
    // update some data
    if (counter == 2) {
        lock(s2);
        // update some more data
        unlock(s2);
    }
    lock(s1);
    // update even more data
    unlock(s3);
    unlock(s1);
}
```

a. Is the code vulnerable to race conditions?

Solution:
Yes. `counter` *is accessed by both threads, and the critical sections are enclosed by different locks (*`s1` *in case of thread 1,* `s3` *in case of thread 2). As* `counter++` *is not atomic,* `counter` *could take the value 1 instead of 2 after both threads have performed their increment operation.*

b. Can a deadlock occur? Why, or why not?

Solution:
Yes. Assume thread 1 runs first and is preempted after having acquired lock `s2`*. Now thread 2 runs and increments* `counter` *to two. Thus, the condition of the* `if`*-statement is* `true`*, and thread 2 will try to acquire lock* `s2`*. Since lock* `s2` *is already held by thread 1, thread 2 blocks, waiting for thread 1 to release the lock. When thread 1 is scheduled, it will however try to acquire* `s3`*, which is already held by thread 2. Thus, each thread is waiting for a lock held by the other. A deadlock occurred.*

Question 8.4: Deadlock Prevention and Avoidance

a. What is deadlock prevention and how does it differ from deadlock avoidance?

Solution:

deadlock prevention *Deadlocks cannot occur, because (at least) one of the four conditions that are necessary for deadlocks is broken.*

deadlock avoidance *The resource allocator has knowledge about the way resources are used in processes. The resource allocator only fulfills a resource request if at least one schedule remains in which all processes can finish — that is, if the system remains in a safe state.*

b. For each condition necessary for deadlocks, give an example of how deadlocks can be prevented by breaking the condition.

Solution:

Mutual exclusion (aka exclusiveness) *In some cases, spooling can be used to avoid that multiple processes directly access a non-sharable resource. As an example, consider printer spooling. Only the spooler accesses the exclusive resource, all other threads send their requests to the spooler.*

Hold and wait *Prevent deadlocks by allocating all resources atomically—that is, en bloc. Once a process holds any resources, it must not allocate additional resources.*

No preemption *Prevent deadlocks by allowing resources to be revoked — that is, multiplex resources by saving and restoring the states of the preempted resource (e.g., CPU, RAM). Of course, this is not possible for all kinds of resources.*

Circular wait *Prevent deadlocks by ordering resources numerically and only allow multiple resources to be allocated in a specified order (e.g., ascendingly). That is a common strategy, when developing multi-threaded applications that require threads to hold multiple locks at once.*

c. What is a safe state?

Solution:
In a safe state it is guaranteed that there exists a scheduling order that allows all processes to run to completion, even if all processes request their maximum number of resources immediately.

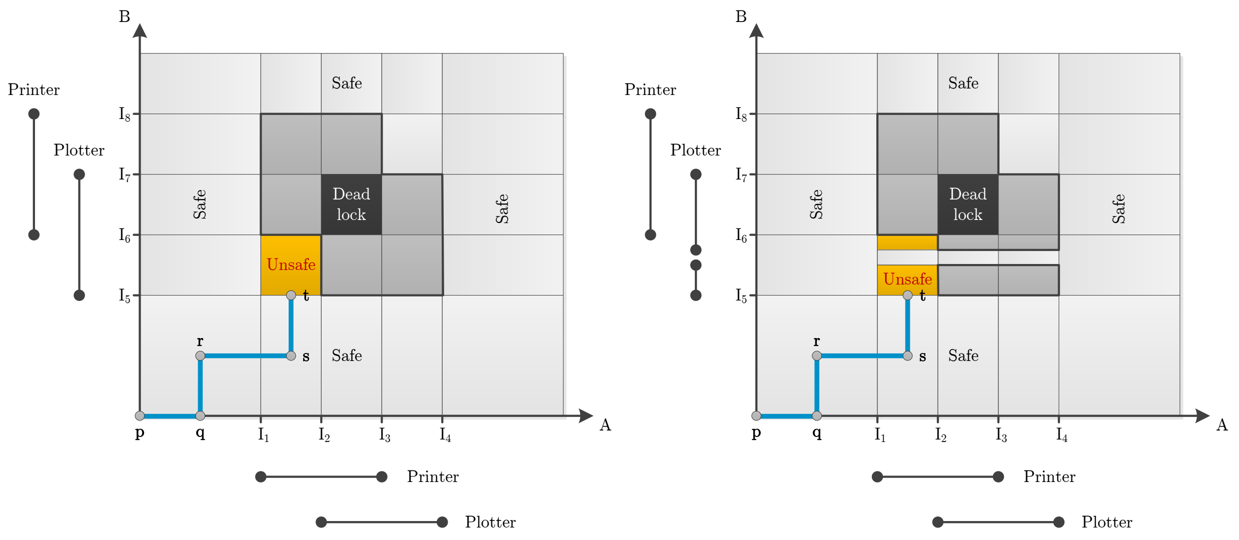


Abbildung 1: Safe and Unsafe States

Processes A and B both require the printer and spooler resources to run to completion. Depending on the scheduling and actual allocation, we stay in a safe state, run into an unsafe state that will eventually lead to a deadlock (left) or run into an unsafe state that does not lead to a deadlock (right).

It is the task of deadlock avoidance *to only grant resource requests if they maintain a safe state. For that purpose, the system must know in advance which resources (and their maximum number, if multiple are required and exist) each process is going to use.*

When a resource request would lead to an unsafe state, the request is not granted and the requesting process has to wait. As the system is in a safe state, it is guaranteed that the request can eventually be granted (once other processes have released resources and/or completed their execution).

If the system gets into an unsafe state, it may or may not deadlock, depending on how the processes allocate resources.

Question 8.5: Resource Allocation Graphs

a. What kinds of vertices exist in a resource allocation graph (RAG)?

Solution:
A RAG has vertices for processes and for the various resource types. To be able to easily distinguish these two kinds, vertices for processes have a circular shape, whereas vertices for resources have a rectangular shape.

b. How are resource types with multiple instances per resource represented in a RAG?

Solution:
For each instance of a resource of a certain type, a dot is drawn inside the resource vertex for that resource type.

c. What is a request edge?

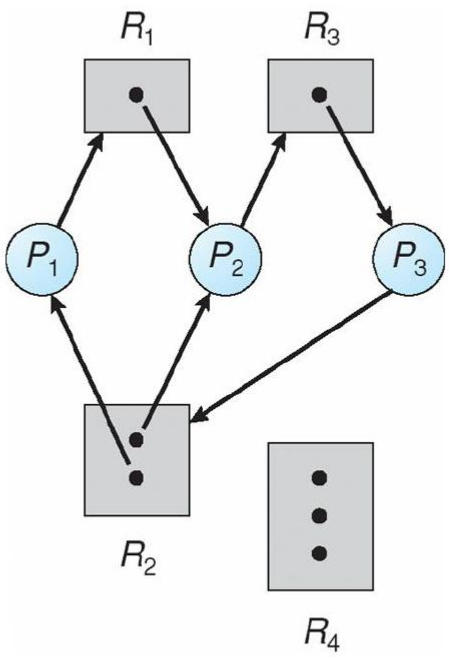
Solution:
A request edge is a directed edge from a process vertex to a resource vertex. It indicates that the process wants to allocate a resource of that type.

d. What is an assignment edge?

Solution:
An assignment edge is a directed edge from a specific instance of a resource to a process vertex. It indicates that the resource is assigned to that process.

e. Does a cycle in a RAG always mean that a deadlock occurred?

Solution:
No. A cycle in a RAG only indicates a deadlock if it involves a set of resource types each of which has only a single instance. If a resource type with multiple instances belongs to that cycle, the system may or may not be in a deadlocked state.



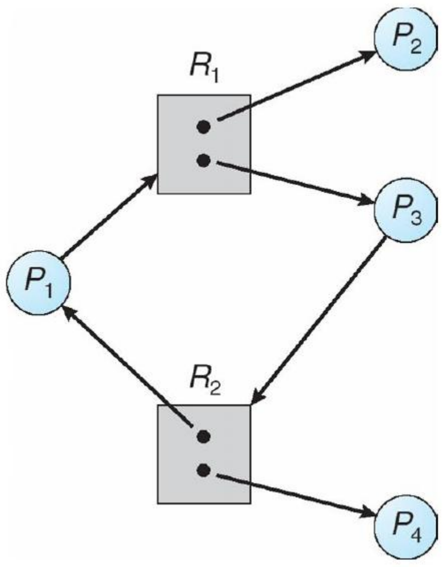


Abbildung 2: Both graphs have cycles. Left: Deadlock. Right: No Deadlock.