Operating Systems 2230

Computer Science & Software Engineering

Lecture 9: Concurrency in Operating Systems

Process management in operating systems can be classified broadly into three categories:

Multiprogramming involves multiple processes on a system with a single processor.

Multiprocessing involves multiple processes on a system with multiple processors.

Distributed processing involves multiple processes on multiple systems.

All of these involve cooperation, competition, and communication between processes that either run simultaneously or are interleaved in arbitrary ways to give the appearance of running simultaneously.

Concurrent processing is thus central to operating systems and their design.

Principles and Problems in Concurrency

Concurrency is the interleaving of processes in time to give the appearance of simultaneous execution. Thus it differs from *parallelism*, which offers genuine simultaneous execution. However the issues and difficulties raised by the two overlap to a large extent:

- sharing global resources safely is difficult;
- optimal allocation of resources is difficult;
- locating programming errors can be difficult, because the contexts in which errors occur cannot always be reproduced easily.

Parallelism also introduces the issue that different processors may run at different speeds, but again this problem is mirrored in concurrency because different processes progress at different rates.

A Simple Example

The fundamental problem in concurrency is processes interfering with each other while accessing a shared global resource. This can be illustrated with a surprisingly simple example:

```
chin = getchar();
chout = chin;
putchar(chout);
```

Imagine two processes `P1` and `P2` both executing this code at the "same" time, with the following interleaving due to multi-programming.

1. `P1` enters this code, but is interrupted after reading the character x into `chin`.
2. `P2` enters this code, and runs it to completion, reading and displaying the character y.
3. `P1` is resumed, *but* `chin` now contains the character y, so `P1` displays the wrong character.

The essence of the problem is the shared global variable `chin`. `P1` sets `chin`, but this write is subsequently lost during the execution of `P2`. The general

solution is to allow only one process at a time to enter the code that accesses `chin`: such code is often called a *critical section*. When one process is inside a critical section of code, other processes must be prevented from entering that section. This requirement is known as *mutual exclusion*.

Mutual Exclusion

Mutual exclusion is in many ways the fundamental issue in concurrency. It is the requirement that when a process `P` is accessing a shared resource `R`, no other process should be able to access `R` until `P` has finished with `R`. Examples of such resources include files, I/O devices such as printers, and shared data structures.

There are essentially three approaches to implementing mutual exclusion.

- Leave the responsibility with the processes themselves: this is the basis of most software approaches. These approaches are usually highly error-prone and carry high overheads.
- Allow access to shared resources only through special-purpose machine instructions: i.e. a hardware approach. These approaches are faster but still do not offer a complete solution to the problem, e.g. they cannot guarantee the absence of deadlock and starvation.
- Provide support through the operating system, or through the programming language. We shall outline three approaches in this category: semaphores, monitors, and message passing.

Semaphores

The fundamental idea of semaphores is that processes "communicate" via global counters that are initialised to a positive integer and that can be accessed

only through two *atomic operations* (many different names are used for these operations: the following names are those used in Stallings Page 216):

semSignal(x) increments the value of the semaphore **x**.

semWait(x) tests the value of the semaphore **x**: if $\mathbf{x} > 0$, the process decrements **x** and continues; if $\mathbf{x} = 0$, the process is blocked until some other process performs a **semSignal**, then it proceeds as above.

A critical code section is then protected by bracketing it between these two operations:

```
semWait (x);
<critical code section>
semSignal (x);
```

In general the number of processes that can execute this critical section simultaneously is determined by the initial value given to **x**. If more than this number try to enter the critical section, the excess processes will be blocked until some processes exit. Most often, semaphores are initialised to one.

Monitors

The principal problem with semaphores is that calls to semaphore operations tend to be distributed across a program, and therefore these sorts of programs can be difficult to get correct, and very difficult indeed to prove correct!

Monitors address this problem by imposing a higher-level structure on accesses to semaphore variables. A monitor is essentially an object (in the Java sense) which has the semaphore variables as internal (private) data and the semaphore operations as (public) operations. Mutual exclusion is provided by allowing only one process to execute the monitor's code at any given time.

Monitors are significantly easier to validate than "bare" semaphores for at least two reasons:

- all synchronisation code is confined to the monitor; and
- once the monitor is correct, any number of processes sharing the resource will operate correctly.

Message Passing

With an approach based on message passing, processes operate in isolation from each other (i.e. they do not share data), and they exchange information where necessary by the sending and receiving of messages.

Synchronisation between processes is defined by the *blocking policy* attached to the sending and receiving of messages. The most common combination is

Non-blocking send: When a process sends a message, it continues executing without waiting for the receiving process.

Blocking receive: When a process attempts to receive a message, it blocks until the message is available.

With this blocking policy, mutual exclusion can be achieved for a set of processes that share a mailbox `box`. Some number of messages (usually one) is sent to `box` initially by the system, then each process executes the following code when it wants to enter the critical section:

```
receive (box);
<critical code section>
send (box);
```

The similarities to the semaphore approach are obvious.

Deadlock

Deadlock is defined as the permanent blocking of a set of processes that either compete for global resources or communicate with each other. It occurs when each process in the set is blocked awaiting an event that can be triggered only by another blocked process in the set.

Consider Figure 6.2 (all figures are taken from Stallings' web-site), in which both processes `P` and `Q` need both resources `A` and `B` simultaneously to be able to proceed. Thus `P` has the form `get A, ... get B, ..., release A, ..., release B`, and `Q` has the form `get B, ... get A, ..., release B, ..., release A.`

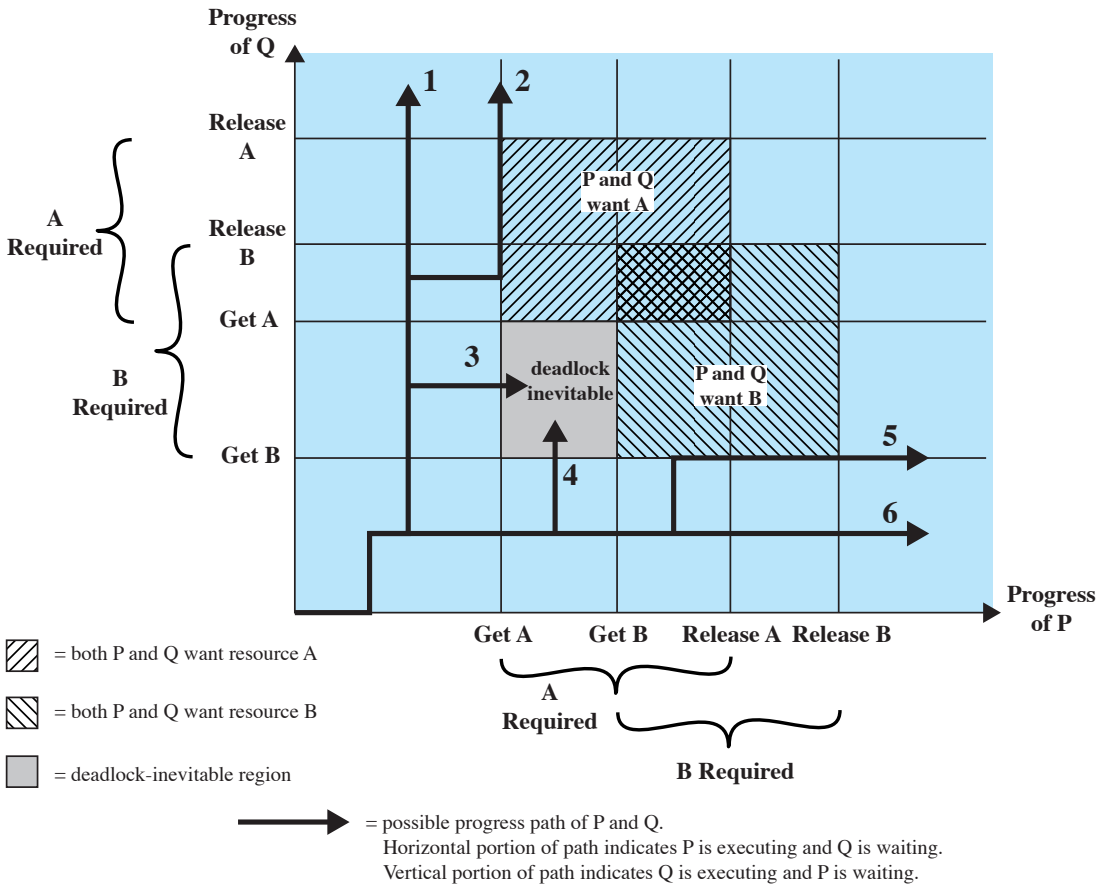

Figure 6.2 Example of Deadlock

The six paths depicted in Figure 6.2 correspond to the following.

1. `Q` acquires both resources, then releases them. `P` can operate freely later.
2. `Q` acquires both resources, then `P` requests `A`. `P` is blocked until the resources are released, but can then operate freely.
3. `Q` acquires `B`, then `P` acquires `A`, then each requests the other resource. Deadlock is now inevitable.
4. `P` acquires `A`, then `Q` acquires `B`, then each requests the other resource. Deadlock is now inevitable.
5. `P` acquires both resources, then `Q` requests `B`. `Q` is blocked until the resources are released, but can then operate freely.
6. `P` acquires both resources, then releases them. `Q` can operate freely later.

Contrast the above example with the situation where `P` does not need the resources simultaneously: `P` has the form `get A, ... release A, ..., get B, ..., release B`, and Q is defined as before (Figure 6.3).

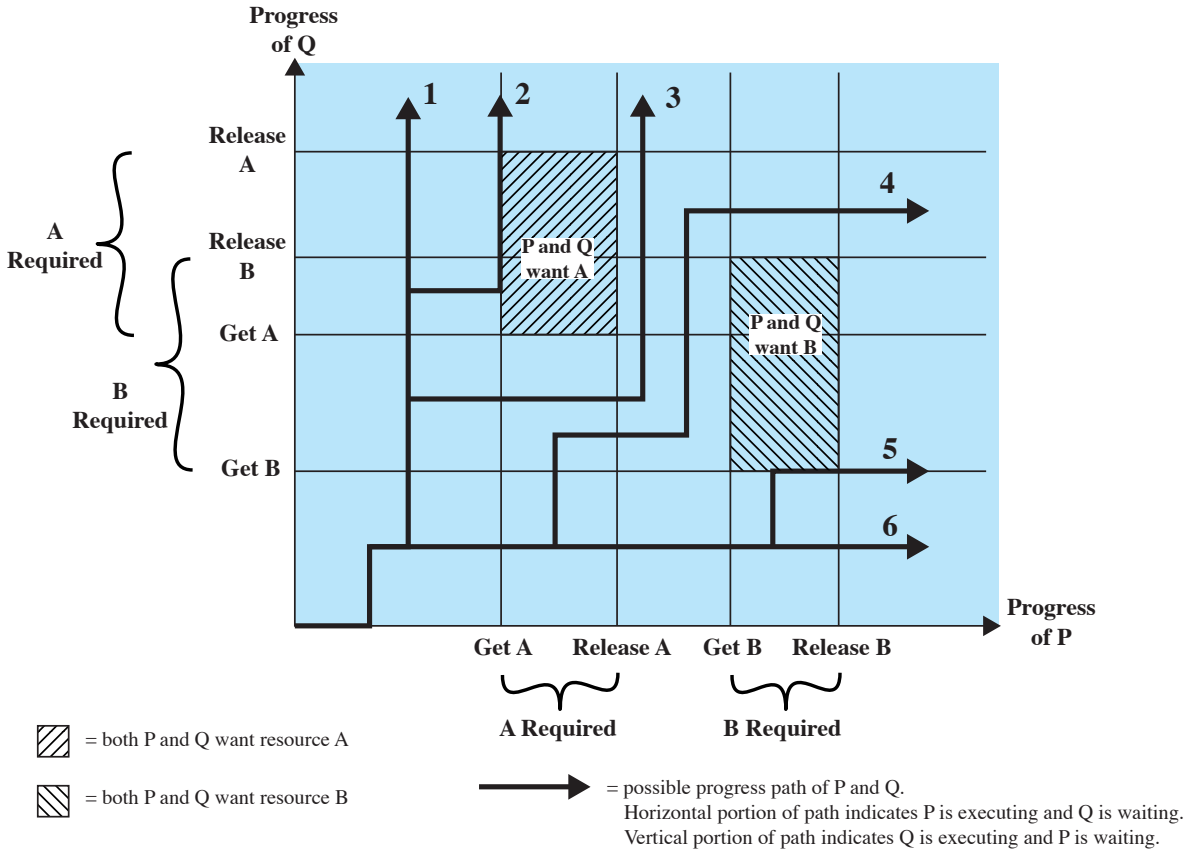

Figure 6.3 Example of No Deadlock [BACO03]

Again the six paths show the possible execution sequences. Make sure you understand why deadlock is impossible in this case.

Conditions for Deadlock

Three policy conditions are necessary for deadlock to be possible.

Mutual exclusion Only one process may use a resource at one time.

Hold and wait A process may hold some resources while waiting for others.

No preemption No process can be forced to release a resource.

A fourth condition is required for deadlock to actually occur.

Circular wait A closed chain of processes exists, such that each process is blocked waiting for a resource held by another process in the set.

Three approaches exist for dealing with deadlock.

Prevention involves adopting a static policy that disallows one of the four conditions above.

Avoidance involves making dynamic choices that guarantee prevention.

Detection and recovery involves recognising when deadlock has occurred, and trying to recover.

Deadlock Prevention

We can prevent deadlock from occurring by adopting either

an indirect policy that disallows one of the three necessary conditions, or

a direct policy that disallows sets of processes that can exhibit a circular wait.

Mutual exclusion

In general, mutual exclusion is essential and cannot be disallowed.

Hold and wait

We could force a process to request all of its resources at one time: if successful, it would be able to proceed to completion; if unsuccessful, it would be holding no resources that could block other processes.

The principal problems are that

- a modular program may not be aware of all of its resource requirements;
- a process needing several resources may be held up a long time waiting for them all to be available simultaneously;
- resources allocated to a process may not be used for a long time, leading to inefficient allocations.

No preemption

We could force a process that holds `R` and then unsuccessfully requests `R'` to release `R`. Or we could force a process that holds `R` to release it, if it is requested by another process.

These approaches work only with resources whose state can easily be saved and restored, and even then they lead to obvious inefficiencies.

Circular wait

We could force all processes to request resources in the same order. Then two processes both needing `A` and `B` would both request `A` first, guaranteeing that `B` would be available for the successful process.

Disallowing circular wait suffers from similar inefficiencies as disallowing hold and wait.

Deadlock Avoidance

Deadlock avoidance is subtly different from deadlock prevention: we allow the three necessary conditions for deadlock, but we dynamically allocate resources in such a way that deadlock never occurs. There are two principal approaches.

- Do not start a process if its demands might lead to deadlock. This strategy is very conservative and thus inefficient.
- Do not grant a resource request if this allocation might lead to deadlock. The basic idea is that a request is granted only if some allocation of the remaining free resources is sufficient to allow all processes to complete. (Consider Figure 6.2 under this approach.)

Both strategies require processes to state their resource requirements in advance.

Deadlock Detection and Recovery

Both prevention and avoidance of deadlock lead to conservative allocation of resources, with corresponding inefficiencies. Deadlock detection takes the opposite approach:

- make allocations liberally, allowing deadlock to occur (on the assumption that it will be rare),
- apply a detection algorithm periodically to check for deadlock, and
- apply a recovery algorithm when necessary.

The detection algorithm can be applied at every resource allocation, or less frequently, depending on the trade-off between the likelihood of deadlock occurring and the cost of the algorithm. Detection algorithms are broadly similar to the avoidance algorithms discussed previously, and are able to identify which processes are deadlocked under the current resource allocation.

Recovery algorithms vary a lot in their severity:

1. Abort all deadlocked processes. Though drastic, this is probably the most common approach!
2. Back-up all deadlocked processes. This requires potentially expensive roll-back mechanisms, and of course the original deadlock may recur.
3. Abort deadlocked processes one at a time until the deadlock no longer exists.
4. Preempt resources until the deadlock no longer exists.

With the last two approaches, processes or resources are chosen to minimise the global "loss" to the set of processes, by minimising the loss of useful processing with respect to relative priorities.