Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
WS 2013/2014

AG Betriebssysteme FB3
Prof. Dr. Jan Peleska, Florian Lapschies

Abgabetermin: 09.12.2013

Hinweise zu Übungszettel 3

1 Deadlock-Erkennung mit Wartegraphen

Um die größten Unklarheiten zu dem dritten Übungsblatt auszuräumen, haben wir Euch noch ein paar Hinweise zur Lösung zusammengetragen.

1.1 Aufbau des Wartegraphen

Der Wartegraph wird über die Funktionen *update*(), *wait*(), und *wakeup*() aufgebaut.

Unsere Wartegraphbibliothek verfügt über keine Funktion zum expliziten Anlegen neuer Prozesse (d.h. Knoten im Wartegraph). Das bedeutet, dass diese u.U. erst angelegt werden müssen, wenn diese das erste Mal referenziert werden.

1.1.1 update(p, ss)

Mit dieser Methode wird dem Algorithmus mitgeteilt, dass ein Prozess p im weiteren Verlauf die Semaphoren in ss inkrementieren wird. Dabei kann die Menge ss auch leer sein.

In unserer Bibliothek sollen nicht die Zähler von Semaphoren verwaltet werden, da ausschließlich Mutexe unterstützt werden. Mit *update*() wird lediglich mitgeteilt, dass ein Prozess eine Menge von Semaphoren inkrementieren wird.

Für den Wartegraph bedeutet dies, dass die eingehenden Kanten von Prozess p neu gezeichnet werden. Zum einen müssen also eingehende Kanten entfernt werden, die mit einer Semaphore markiert sind, die nicht in ss enthalten ist. Zum anderen müssen für alle Semaphoren $s \in ss$, für die p vorher nicht angekündigt hat ein Up zu machen, Prozesse p' gesucht werden, die auf diese Semaphore warten. Für jeden dieser Prozesse p' muss dann eine neue mit s markierte Kante von p' nach p gezogen werden.

1.1.2 wait(p, sem)

Mit dieser Methode wird dem Algorithmus mitgeteilt, dass der Prozess p ein blockierendes Down auf die Semaphore s ausgeführt hat und nun wartet. In unserem Szenario kann ein Prozess nur auf eine Semaphore zur Zeit warten.

Für den Wartegraphen bedeutet dies, dass mit s markierte Kanten von p hin zu allen Prozessen p', die angekündigt haben in Zukunft ein Up auf s auszuführen, angelegt werden müssen.

Um die Bibliothek robust zu gestalten, solltet Ihr noch den Fall abfangen, dass der Prozess p schon bereits auf eine Semaphore s' wartet, wenn $wait(p, s)$ aufgerufen wird. Wenn $s = s'$, besteht kein Handlungsbedarf. Wenn jedoch $s \neq s'$, wirft Ihr eine Exception (ja, sowas geht auch in C++).

1.1.3 wakeup(p)

Mit dieser Methode wird dem Algorithmus mitgeteilt, dass der Prozess p nun nicht mehr blockiert ist.

Für den Wartegraph bedeutet dies, dass alle ausgehenden Kanten von p entfernt werden.

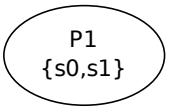
Man könnte meinen, dass wenn ein Prozess p mit *wakeup*() aufwacht und er vorher auf eine Semaphore s warten musste, muss es auch einen anderen Prozess p' geben, der ein Up auf diese Semaphore aufgerufen hat. Diesem Gedankengang folgend, müsste dann aus der Menge von Semaphoren, auf denen p' ein Up

machen wird, s entfernt werden. Tatsächlich kann es aber auch eine ganze Reihe von anderen Gründen zum Aufwachen geben, wie zum Beispiel das Empfangen eines Signal. Unsere Bibliothek soll hiervon abstrahieren und wird lediglich über das Aufwachen eines Prozesses mit *wakeuop*() informiert. Es liegt in der Verantwortung des Nutzers der Bibliothek unter Umständen noch zusätzlich *update*() aufzurufen.

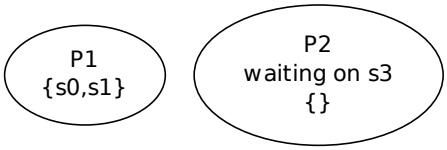
Um die Bibliothek robust zu gestalten, solltet Ihr noch den Fall abfangen, dass der Prozess p aktuell gar nicht blockiert ist. Auch hier bietet sich das Werfen einer Exception an.

1.1.4 Beispielablauf

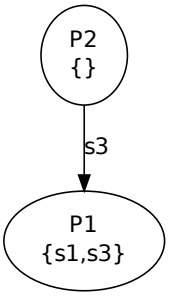
1. $update(P_1, \{s_0, s_1\})$
   Prozess P_1 wird im weiteren Verlauf jeweils Ups auf die Semaphoren $\{s_0, s_1\}$ ausführen. Da P_1 vorher nicht genannt wurde, wird für ihn ein neuer Knoten angelegt.



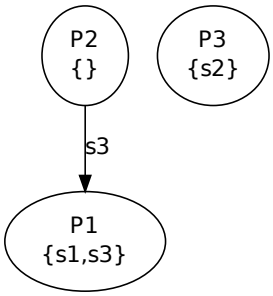
2. $wait(P_2, s_3)$
   Prozess P_2 führt ein blockierendes Down auf Semaphore s_3 aus. Da auch P_2 vorher nicht genannt wurde, wird für ihn ein neuer Knoten angelegt. Aktuell hat der Prozess P_2 noch für keine Semaphore angekündigt ein Up auszuführen. Da es bislang keine Prozesse gibt, die zukünftig ein Up auf s_3 ausführen, werden (noch) keine mit s_3 markierte Kanten dem Graphen hinzugefügt. Trotzdem muss Eure Bibliothek vermerken, dass P_2 auf s_3 wartet, damit im weiteren Verlauf u.U. solche Kanten angelegt werden können.



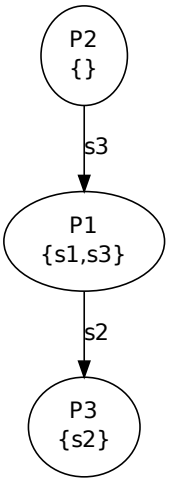
3. $update(P_1, \{s_1, s_3\})$ Wenn nun erneut *update*() für Prozess P_1 aufgerufen wird, wird die Menge der Semaphoren für die P_1 verspricht ein Up auszuführen auf $\{s_1, s_3\}$ geändert. Nun wird eine mit s_3 markierte Kante von P_2 nach P_1 gezeichnet, da P_2 auf s_3 wartet und P_1 diese Situation durch ein Up auflösen könnte.



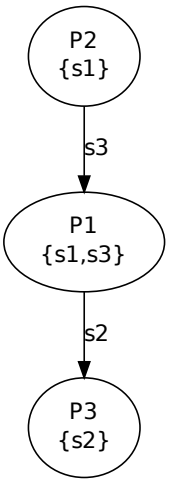
4. $update(P_3, \{s_2\})$



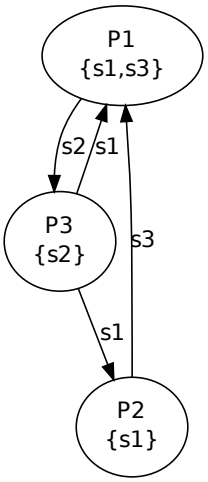
5. $wait(P_1, s_2)$



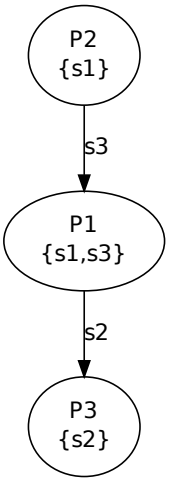
6. $update(P_2, s_1)$



7. $wait(P_3, s_1)$



8. $wakeup(P_3)$



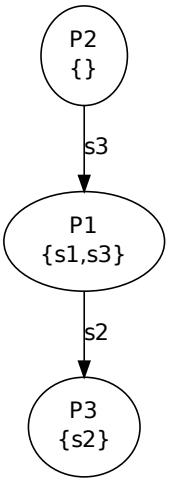
1.2 Deadlockprüfung

Ein Deadlock liegt genau dann vor, wenn einer der beiden Trivialfälle vorliegt oder wenn der Wartegraph eine nicht-triviale starke Zusammenhangskomponente aufweist, aus der kein Weg hinausführt.

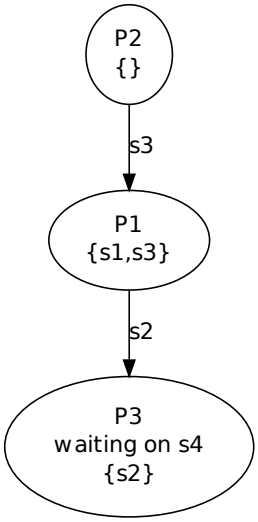
1.2.1 Trivialfall 1

Wenn ein Prozess auf eine Semaphore ein blockierendes Down gemacht hat, aber kein Prozess angekündigt hat ein Up auf diese Semaphore auszuführen, liegt ein Deadlock vor, auch wenn der Graph keine nicht-triviale starke Zusammenhangskomponente aufweist.

1. Ausgangssituation:



2. $wait(P_3, s_4)$
   Prozess P_3 macht auf Semaphore s_4 ein blockierendes Down, aber kein Prozess hat angekündigt ein Up auf s_4 auszuführen. In diesem Fall liegt ein Deadlock vor, auch wenn der Graph keine nicht-triviale starke Zusammenhangskomponente aufweist.

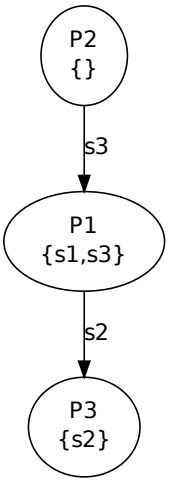


3. $deadlock() \rightsquigarrow true$

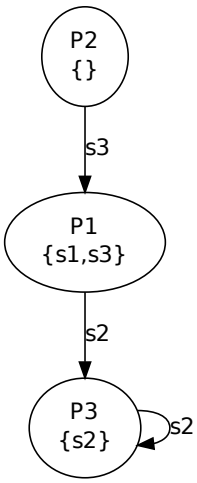
1.2.2 Trivialfall 2

Wenn ein Prozess auf eine Semaphore ein blockierendes Down gemacht hat und dieser Prozess der einzige Prozess ist, der in Zukunft ein Up auf diese Semaphore ausführen wird, dann liegt ein Deadlock vor, auch wenn der Graph keine nicht-triviale starke Zusammenhangskomponente aufweist.

1. Ausgangssituation: Prozess P_3 wird zukünftig ein Up auf Semaphore s_2 machen.



2. $wait(P_3, s_2)$
   Prozess P_3 macht auf Semaphore s_2 ein blockierendes Down, aber P_3 ist der einzige Prozess, der angekündigt hat ein Up auf s_2 auszuführen. In diesem Fall liegt ein Deadlock vor, auch wenn der Graph keine nicht-triviale starke Zusammenhangskomponente aufweist.



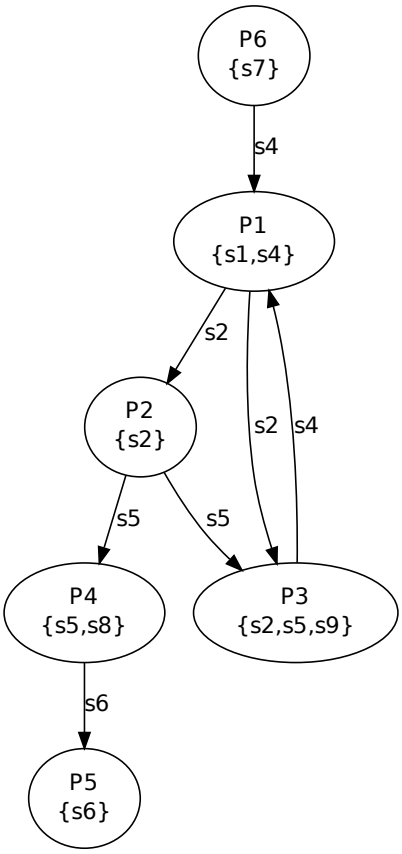
3. $deadlock() \rightsquigarrow true$

1.2.3 Nicht-triviale starke Zusammenhangskomponente

Ebenfalls liegt ein Deadlock vor, wenn der Wartegraph mindestens eine Nicht-triviale starke Zusammenhangskomponente Z aufweist, aus der kein Weg hinausführt. Solche eine ist gegeben, wenn...

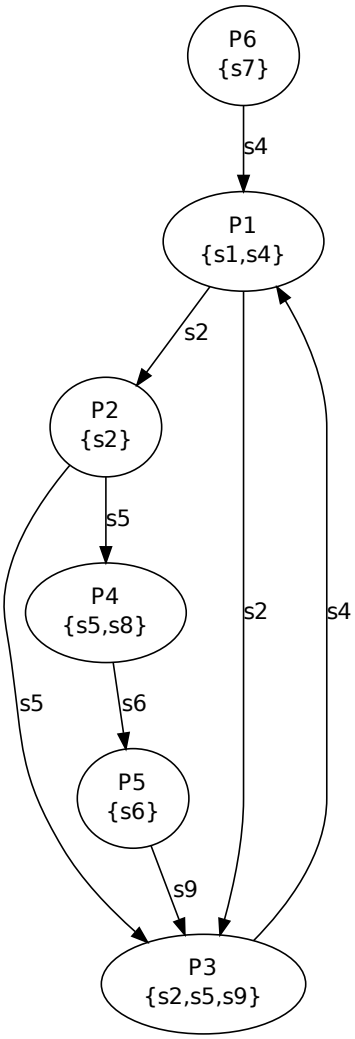
- Z ist Subgraph mit mindestens zwei Knoten.
- Von jedem Knoten von Z gibt es mindestens einen gerichteten Pfad zu jedem anderen Knoten von Z.
- Jede Kante die von einem Knoten von Z ausgeht, führt auch zu einem Knoten von Z.

Beispiel 1: 1. Ausgangssituation:



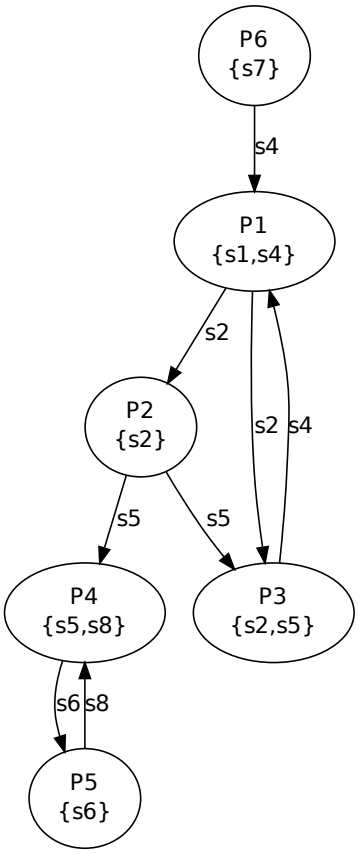
2. $deadlock() \rightsquigarrow false$
   Die Knoten $\{P_1, P_2, P_3\}$ sind keine Nicht-triviale starke Zusammenhangskomponente, aus der kein Weg hinausführt, da es bei P_2 eine ausgehende Kante gibt, die zu einem Knoten (P_5) führt, der nicht teil dieser starken Zusammenhangskomponente ist.

Beispiel 2: 1. Ausgangssituation:



2. $deadlock() \rightsquigarrow true$
   Der Deadlock wird daran erkannt, dass die Knoten $\{P_1, P_2, P_3, P_4, P_5\}$ Teil einer Nichttrivialen starken Zusammenhangskomponente sind, aus der kein Weg hinaus führt.

Beispiel 3: 1. Ausgangssituation:



2. $deadlock() \rightsquigarrow true$
   Die Knoten $\{P_1, P_2, P_3\}$ bilden zwar eine nicht triviale Zusammenhangkomponente, jedoch führt von von P_2 nach P_4 eine Kante aus dieser heraus.
   Der Deadlock wird aber trotzdem daran erkannt, dass die Knoten $\{P_4, P_5\}$ Teil einer Nichttrivialen starken Zusammenhangskomponente sind, aus der kein Weg hinaus führt.