Przydział zasobów

Jedną z funkcji systemów operacyjnych jest przydział, albo alokacja, zasobów. Nadzór systemu jest konieczny wtedy, gdy z danego zasobu może korzystać wyłącznie jeden proces na raz.

Przykładami takich zasobów są:

- urządzenia peryferyjne, jak drukarki, napędy optyczne, itp.,
- porty komunikacyjne: szeregowe, równoległe, USB, itp.,
- globalne urządzenia programowe: semafory, obszary pamięci, itp.,
- blokady wyłączności nakładane na dostęp do plików, lub ich fragmentów.

Dwa podstawowe zasoby systemu komputerowego, obsługiwane przez system operacyjny i przydzielane procesom dla umożliwienia ich wykonywania to: procesor oraz pamięć. Te zasoby są przydzielane za pomocą dedykowanych, wyspecjalizowanych algorytmów, i ich szczegółowe własności będą omawiane oddzielnie.

Wywłaszczanie zasobów

Wywłaszczaniem zasobów nazywamy ich odzyskiwanie po tym jak zostały przydzielone procesom, ale zanim dany proces zdecydował się dobrowolnie dany zasób zwolnić.

Procesor i pamięć są przykładami zasobów, które system operacyjny w razie potrzeby jest w stanie skutecznie wywłaszczać od procesów. Wykonujący się proces można usunąć z procesora, by go następnie wznowić w niezauważalny dla niego sposób w cyklu quasi-równoległego wykonywania. A jeśli procesowi została przydzielona pamięć, której potrzebuje inny proces/inne procesy, to dzięki mechanizmom pamięci wirtualnej można temu pierwszemu procesowi odebrać mniej potrzebne strony pamięci, lub ostatecznie cały proces wymieść na dysk, i użyć jego pamięci dla innych celów. Po wznowieniu, jego pamięć zostanie przywrócona do pierwotnego stanu drogą stronicowania.[1]

Jednak wielu zasobów nie da się bezkarnie wywłaszczać. Nie można odebrać procesowi drukarki nie mając pewności na jakim etapie jest jego zadanie drukowania, nie można odebrać mu portu komunikacyjnego jeśli jest możliwość, że rozpoczął on już komunikację z jakimś podsystemem, oraz nie można odebrać mu semafora, bo ten może zabezpieczać toczącą się transakcję, itd.

[1] Niekoniecznie jest to prawda w systemach czasu rzeczywistego. Obowiązują tam inne zasady i wywłaszczenie procesu z procesora albo odebranie przydzielonej mu pamięci może naruszyć wymagania czasowe, które system operacyjny może znać lub nie.

Uzyskiwanie dostępu do zasobów

Dla pewnych zasobów, korzystające z nich procesy muszą same obsługiwać kontrolę dostępu do nich, ponieważ system operacyjnych nie zna logiki rządzącej tym dostępem.

Do zarządzania dostępem do zasobu można posłużyć się semaforem lub muteksem. Przed użyciem zasobu proces zajmuje semafor (operacja `down`, być może czeka na dostęp), a po użyciu zasobu zwalnia semafor (operacja `up`, nigdy nie musi czekać). W przypadku korzystania z więcej niż jednego zasobu na raz, dostęp do nich musi być uzyskiwany sekwencyjnie.

```
typedef int semaphore;
semaphore resource_1;


void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}
```

Powstawanie zakleszczeń

W przypadku różnych procesów konkurujących o dostęp do zasobów możliwe jest napisanie programu w taki sposób, że kontrola dostępu do zasobu przebiega poprawnie (kod po lewej), albo w taki sposób, że powstaje **zakleszczenie** (*deadlock*):

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

Definicja zakleszczenia

Można sformułować następującą definicję zakleszczenia [Tanenbaum, Modern Operating Systems, 3rd Edition]:

> W przypadku zbioru procesów do zakleszczenia dochodzi wtedy, gdy każdy proces w zbiorze oczekuje na zdarzenie, które może spowodować inny proces z tego zbioru.

Ponieważ wszystkie procesy oczekują, żaden z nich nie spowoduje zdarzenia, na które czeka ktoś inny. Procesy z tego zbioru są zatem zakleszczone i pozostaną tak na zawsze. Zakładamy tu, że nic (np. sygnał) nie może przerwać tego czekania.

Typowym zdarzeniem, na które oczekują procesy jest przydział zasobu chwilowo zajmowanego przez inny proces. Taki przypadek zakleszczenia nazywany jest **zakleszczeniem zasobów** (nieco później rozważymy inne rodzaje zakleszczenia).

Warunki powstawania zakleszczenia zasobów

Aby mogło dojść do zakleszczenia zasobów muszą być spełnione cztery warunki:

wzajemne wykluczanie — w danym momencie, każdy zasób jest albo przypisany dokładnie do jednego procesu, albo jest dostępny

wstrzymywanie i oczekiwanie — procesy posiadające zasoby przydzielone wcześniej mogą żądać nowych zasobów

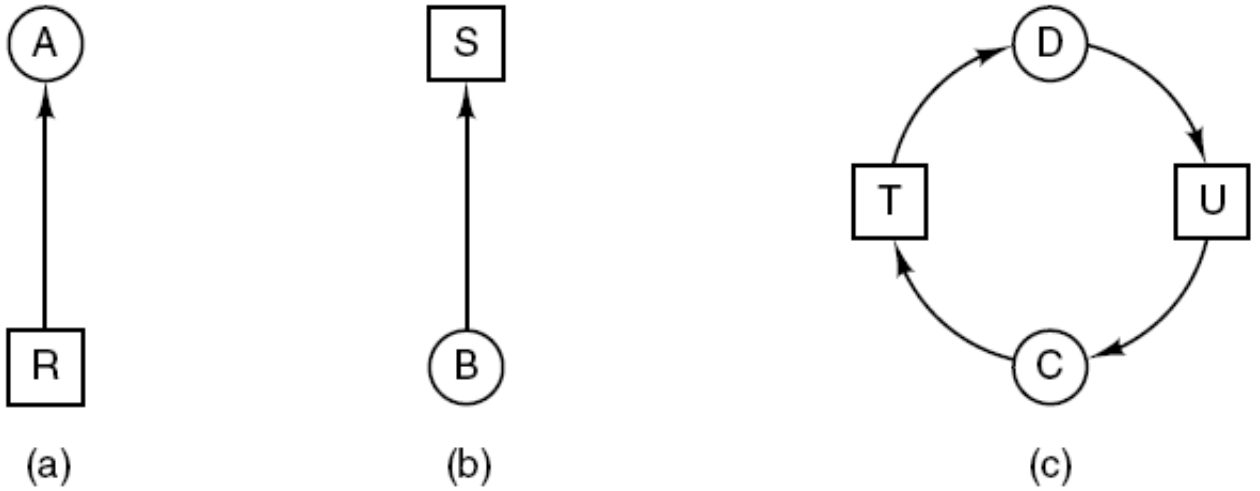
brak wywłaszczania — przydzielone zasoby nie mogą być zabrane procesom; mogą być jedynie dobrowolnie zwolnione

cykliczne oczekiwanie — musi istnieć cykl oczekiwania na zasoby złożony z dwóch lub więcej procesów: każdy proces w tym łańcuchu oczekuje na zasób będący w posiadaniu następnego procesu w łańcuchu

Powyższe warunki są warunkami koniecznymi powstania zakleszczenia. Gdy którykolwiek z nich nie jest spełniony, wtedy do zakleszczenia na pewno nie dojdzie.

Grafy alokacji zasobów

Stany procesów uzyskujących dostęp do zasobów można opisać za pomocą **grafów alokacji zasobów**, na których procesy są reprezentowane przez węzły okrągłe, zasoby jako węzły kwadratowe, a łuki reprezentują posiadanie oraz żądanie zasobów.



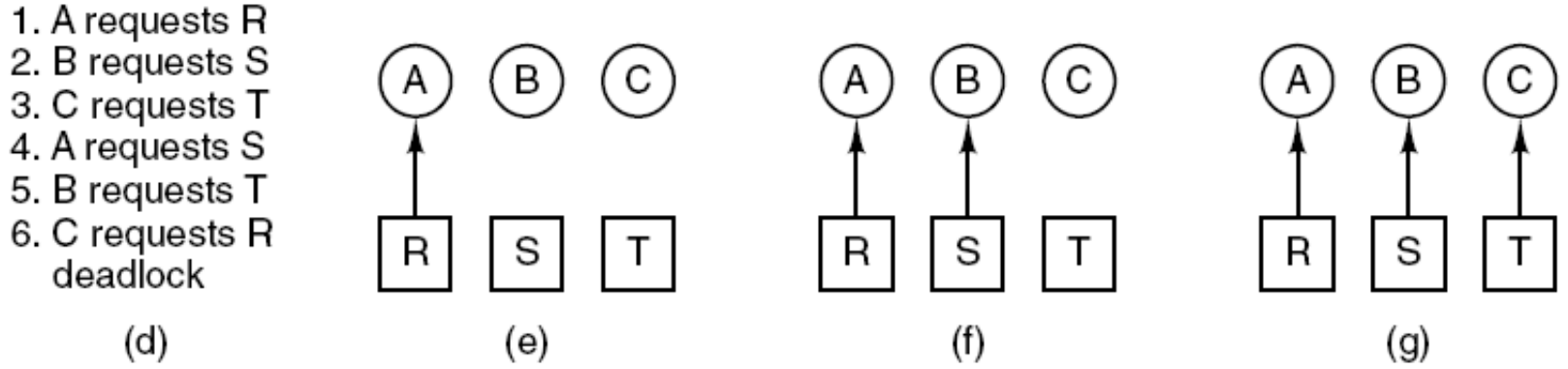
Rysunek (a) przedstawia proces (A) posiadający zasób [R].
Rysunek (b) przedstawia proces (B) żądający dostępu do zasobu [S].
Rysunek (c) przedstawia dwa zakleszczone procesy: (C) i (D). Proces (C) posiada zasób [U] i czeka na zasób [T]. Proces (D) posiada zasób [T] i czeka na zasób [U].

Przykład — sekwencja alokacji zasobów

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R
(a)	(b)	(c)

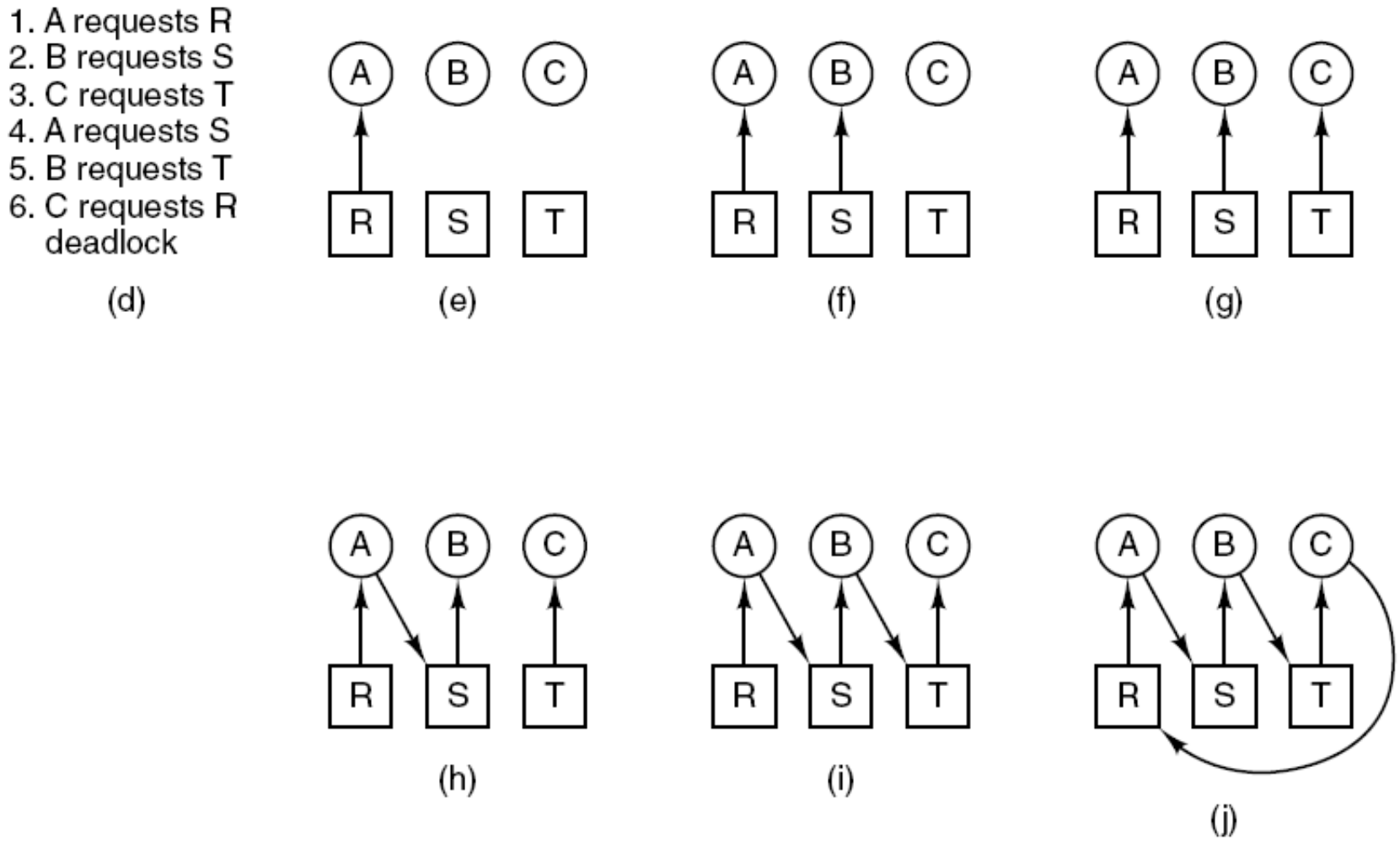
Rozważmy przykład alokacji zasobów na powyższych rysunkach: (a),(b),(c). Jeśli procesy będą uruchamiane sekwencyjnie, najpierw A do zakończenia, potem B do zakończenia, i w końcu C, wtedy do zakleszczenia nie dojdzie.

Jednak jeśli te same procesy, wywołujące te same żądania zasobów w tej samej kolejności, będą wykonywane współbieżnie, i sekwencja żądań zasobów będzie jak na rysunku (d), wtedy tym razem dojdzie do zakleszczenia.



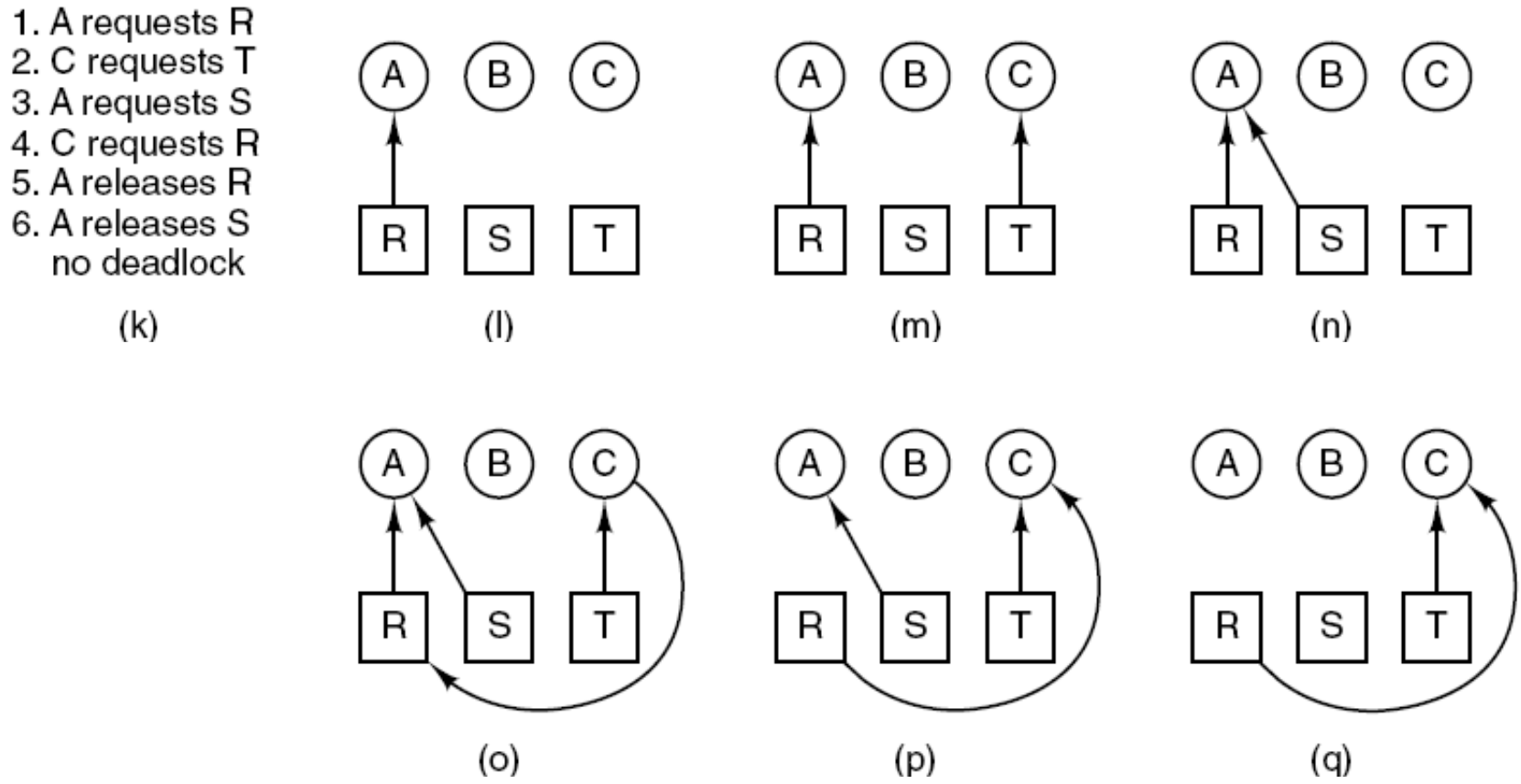
Przykład — powstanie zakleszczenia

Powstanie zakleszczenia w poprzednim przykładzie można przedstawić na grafie alokacji zasobów. Na rysunku (j) widać zakleszczenie po wydaniu przez proces (C) żądania dostępu do zasobu [R].



Przykład — eliminacja zakleszczenia

Jednak powstawanie zakleszczeń w przypadkach takich jak w poprzednim przykładzie jest kwestią przypadku. Gdyby system operacyjny, zamiast przydzielać zasoby procesowi (B), uruchamiał tylko procesy (A) i (C) — przypadkiem, lub świadomie, przewidując nadchodzącą porażkę — do zakleszczenia by nie doszło, co ponownie widać na grafach alokacji zasobów.



Postępowanie z zakleszczeniami

Ogólnie można sformułować następujące podejścia do przeciwdziałania zakleszczeniom:

wykrywanie i usuwanie zakleszczenia — (*deadlock detection and recovery*) dopuszczamy do powstania zakleszczenia, po czym wykrywamy je, i podejmujemy działania w celu jego wyeliminowania

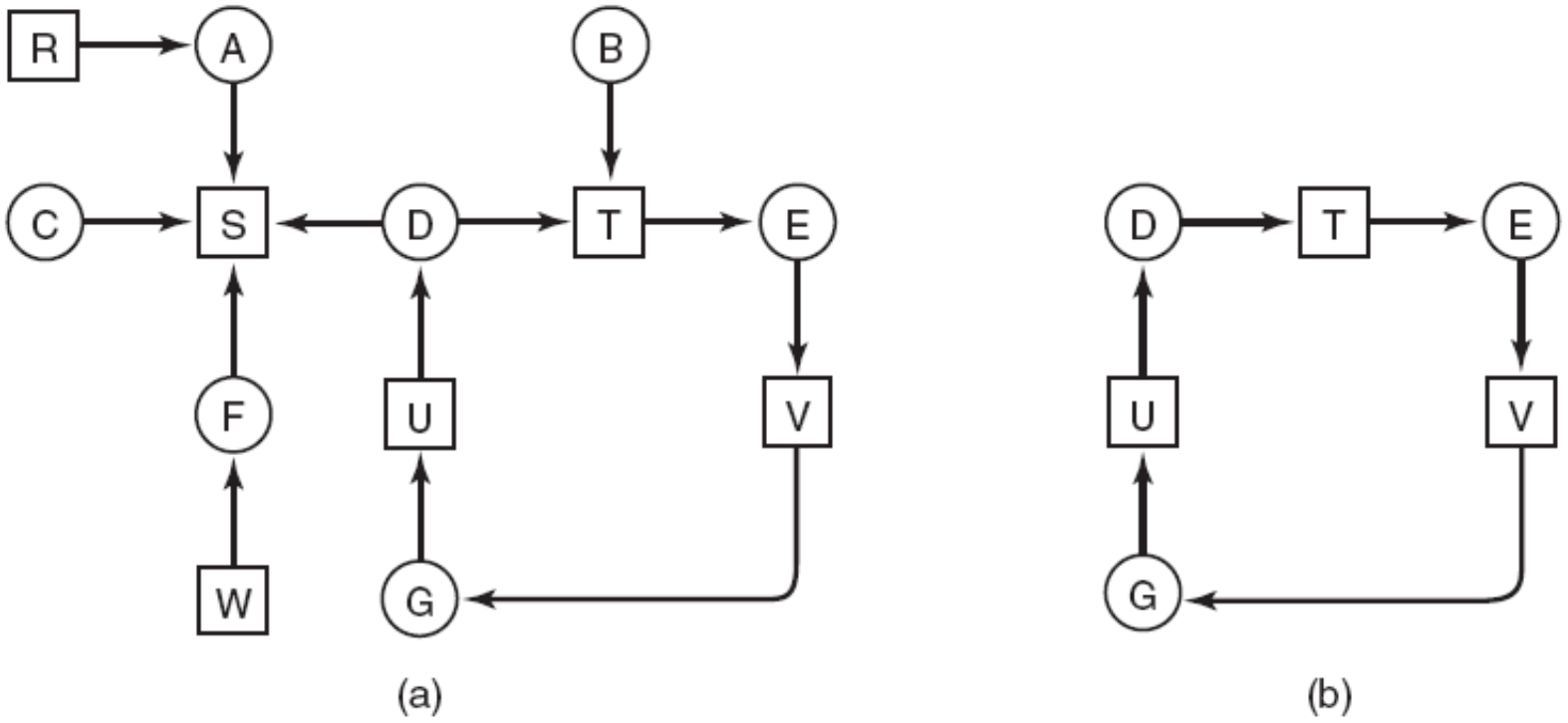
unikanie zakleszczenia — (*deadlock avoidance*) nie dopuszczamy do powstania zakleszczenia poprzez ostrożną alokację zasobów

zapobieganie zakleszczeniom — (*deadlock prevention*) nie dopuszczamy do powstania zakleszczenia poprzez wyeliminowanie jednego z warunków koniecznych powstania zakleszczenia

Do powyższej listy można dodać jeszcze jedną „metodę" postępowania z zakleszczeniami, zwaną **algorytmem strusia** (*ostrich algorithm*). Polega on na całkowitym zignorowaniu problemu zakleszczeń. Problemy te powstają bowiem rzadko, i są tylko jednym z możliwych zagrożeń przy tworzeniu wielowątkowych, współbieżnych systemów. Jednocześnie, przeciwdziałanie zakleszczeniom którąkolwiek z powyższych metod jest kosztowne, i ten koszt system operacyjny musiałby ponosić przy każdej operacji przydziału zasobów. Większość współczesnych systemów operacyjnych wychodzi z założenia, że nie opłaca się przeciwdziałać zakleszczeniom, i tego nie robi.

Wykrywanie zakleszczeń dla pojedynczych egzemplarzy zasobów

W poniższym grafie alokacji zasobów (a) istnieje cykl, oznaczający wystąpienie zakleszczenia. Na rysunku (b) wyodrębnione zostały procesy i zasoby biorące udział w zakleszczeniu.



Algorytm wykrywania zakleszczenia dla pojedynczych egzemplarzy

1. DIa każdego węzła N w grafie wykonaj poniższe kroki rozpoczynając od węzła N.

2. Stwórz pustą listę L; wszystkie łuki określ jako niezaznaczone.

3. Dodaj bieżący węzeł na koniec listy L, i sprawdź, czy węzeł występuje na L dwa razy. Jeśli tak, to graf zawiera cykl. STOP.

4. Sprawdź, czy z tego węzła wychodzą dowolne niezaznaczone łuki. Jeśli tak, to przejdź do kroku 5, a jeśli nie, to przejdź do kroku 6.

5. Losowo wybierz niezaznaczony wychodzący łuk, i go zaznacz. Następnie przejdź po tym łuku do następnego węzła i skocz do kroku 3.

6. Jeśli jest to węzeł początkowy, to graf nie zawiera cyklu. STOP.
   W przeciwnym wypadku osiągnęliśmy martwy koniec. Usuń węzeł z listy i przejdź do poprzedniego węzła, czyli tego, który był bieżący przed węzłem aktualnie analizowanym. Oznacz go jako bieżący i przejdź do kroku 3.

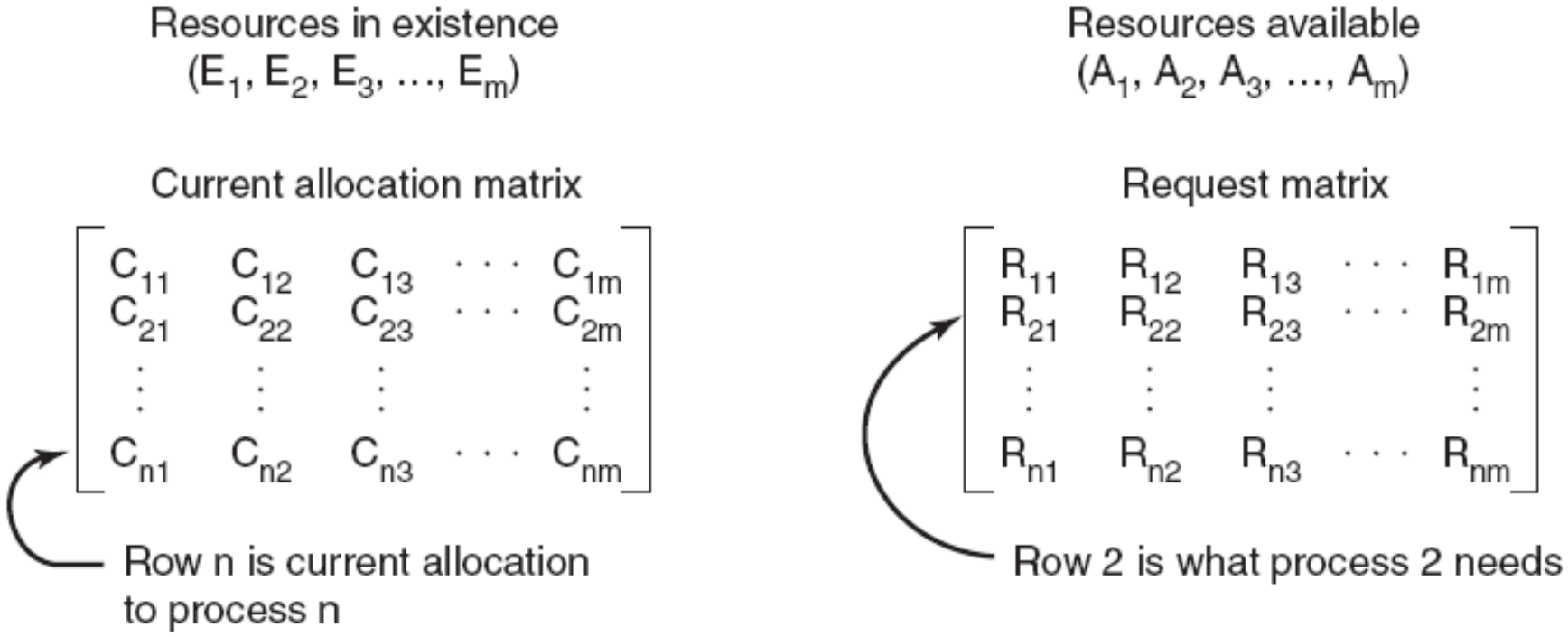
Przydatność analizy grafu do analizy zakleszczeń

Jak mogliśmy się przekonać, grafy alokacji zasobów pozwalają łatwo wykryć sytuację gdy może dojść do zakleszczenia (albowiem dla danej czasowej sekwencji operacji do tego zakleszczenia może nie dojść).

Ponadto, zwykły graf alokacji zasobów nie pozwala analizować sytuacji przydziału wielu egzemplarzy konkretnych zasobów. Ponieważ takie sytuacje zdarzają się często w praktyce, potrzebne jest inne podejście do analizy zakleszczeń.

Wykrywanie zakleszczeń dla wielu egzemplarzy zasobów

W przypadku istnienia wielu egzemplarzy pewnych zasobów konieczne jest inne podejście. Oznaczmy liczbę klas zasobów m i ogólną liczbę egzemplarzy zasobu i przez E_i. $E = (E_1, E_2, ..., E_m)$ będzie wektorem istniejących zasobów. Liczbę dostępnych egzemplarzy zasobu i oznaczymy A_i, i $A = (A_1, A_2, ..., A_m)$ będzie wektorem dostępnych zasobów.



C będzie macierzą bieżącej alokacji, a R — macierzą żądań dla n procesów.

Algorytm wykrywania zakleszczeń

Zauważmy, że dla przyjętych oznaczeń:

$$\forall_j [\sum_{i=1}^{n} C_{ij} \ + A_j = E_j]$$

Przyjmijmy, że dla dwóch wektorów A i B zapis $A \leqslant B$ oznacza, że wszystkie elementy A są mniejsze lub równe odpowiednim elementom B.

Algorytm początkowo traktuje wszystkie procesy jako nieoznaczone. W trakcie pracy oznacza te procesy, które mogą się wykonać korzystając z puli dostępnych zasobów. Po zakończeniu proces może zwrócić wszystkie swoje zasoby do puli, co umożliwia wykonanie się innym procesom.

1. Wybierz nieoznaczony proces P_i którego wiersz R_i macierzy R: $R_i \leqslant A$
   Jeśli nie ma takiego procesu, to algorytm kończy działanie.

2. Jeśli taki proces zostanie znaleziony, niech jego numerem będzie i, dodaj i-ty wiersz macierzy C (C_i) do A, oznacz proces i, i powróć do kroku 1.

Po zakończeniu algorytmu, wszystkie nieoznaczone procesy pozostają zakleszczone. Algorytm jest niedeterministyczny, ale jego wynik jest zawsze taki sam.

Wykrywanie zakleszczeń — przykład

Current allocation matrix

Request matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Początkowo jedynie dla procesu $i = 3$ mamy $R_i \leqslant A$. W efekcie otrzymujemy:

$$A = (2\ 2\ 2\ 0)$$

Teraz proces $i = 2$ ma $R_i \leqslant A$ i może się wykonać. Otrzymujemy:

$$A = (4\ 2\ 2\ 1)$$

W końcu można uruchomić proces $i = 1$. W systemie nie ma zakleszczeń.

Realizacja wykrywania zakleszczeń w systemie

Pozostaje pytanie: kiedy system powinien podejmować wykrywanie zakleszczeń, wykonując powyższy algorytm?

Jedna możliwość jest aby robić to po każdym zgłoszeniu żądania zasobów przez dowolny proces. W takim przypadku system najszybciej jak to jest tylko możliwe otrzyma informację o zagrażającym zakleszczeniu, i będzie mógł podjąć odpowiednie działania.

Jednak powyższa metoda jest kosztowna. Alternatywną metodą jest okresowe wykonywanie algorytmu, i/lub wtedy, gdy obciążenie procesora spadnie poniżej pewnej wartości. Zauważmy, że gdy pewna liczba procesów zostanie zakleszczonych, nie będą one wykonywane, co powinno spowodować spadek obciążenia procesora.

Usuwanie zakleszczeń przez wywłaszczanie

Niekiedy jest możliwe wywłaszczenie pewnych zasobów od wybranego procesu. Może to wymagać ręcznej interwencji operatora.

Na przykład, wywłaszczenie drukarki może odbyć się przez zawieszenie procesu, wyjęcie jego wykonanych już wydruków z drukarki, przydzielenie jej oczekującemu na nią procesowi, a po jego zakończeniu włożenie wydrukowanego papieru z powrotem do drukarki, i wznowienie zawieszonego procesu.

Usuwanie zakleszczeń przez wycofywanie operacji

Program można przygotować do operacji usuwania zakleszczeń poprzez tworzenie **punktów kontrolnych** zapisujących stan programu w pliku na dysku. Punkt kontrolny zawiera obraz pamięci, stany rejestrów, jak również informację o przydziale zasobów.

Po wykryciu zakleszczenia, system określa jaki(e) zasób(y) jest(są) potrzebny(e), i który z zakleszczonych procesów je posiada i mógłby zostać cofnięty do punktu kontrolnego. Gdy zasoby zostaną uwolnione, mogą być przydzielone innemu zakleszczonemu procesowi.

Usuwanie zakleszczeń przez zabijanie procesów

Czasami prościej niż wycofać proces do punktu kontrolnego jest po prostu go zabić.

Oczywiście zabity proces należy później uruchomić ponownie. Byłoby to bezproblemowe, gdyby operacje wykonywane przez proces były **idempotentne**, to znaczy takie, które można wykonywać wiele razy, a one tworzą takie (te) same wyniki.

Przykłady procesów idempotentnych:

- kompilacja jakiegoś systemu oprogramowania
- sprawdzanie spójności systemu plików, albo stanu macierzy RAID
- itp.

Natomiast realizacja pakietu przelewów bankowych (np. miesięcznych wypłat dla pracowników jakiejś firmy) nie jest idempotentna, i normalnie nie można jej bezpiecznie przerwać, i wykonać powtórnie. Ogólnie, procesy, które nie tylko obliczają jakieś wyniki, ale tworzą efekty uboczne, typowo nie są idempotentne.

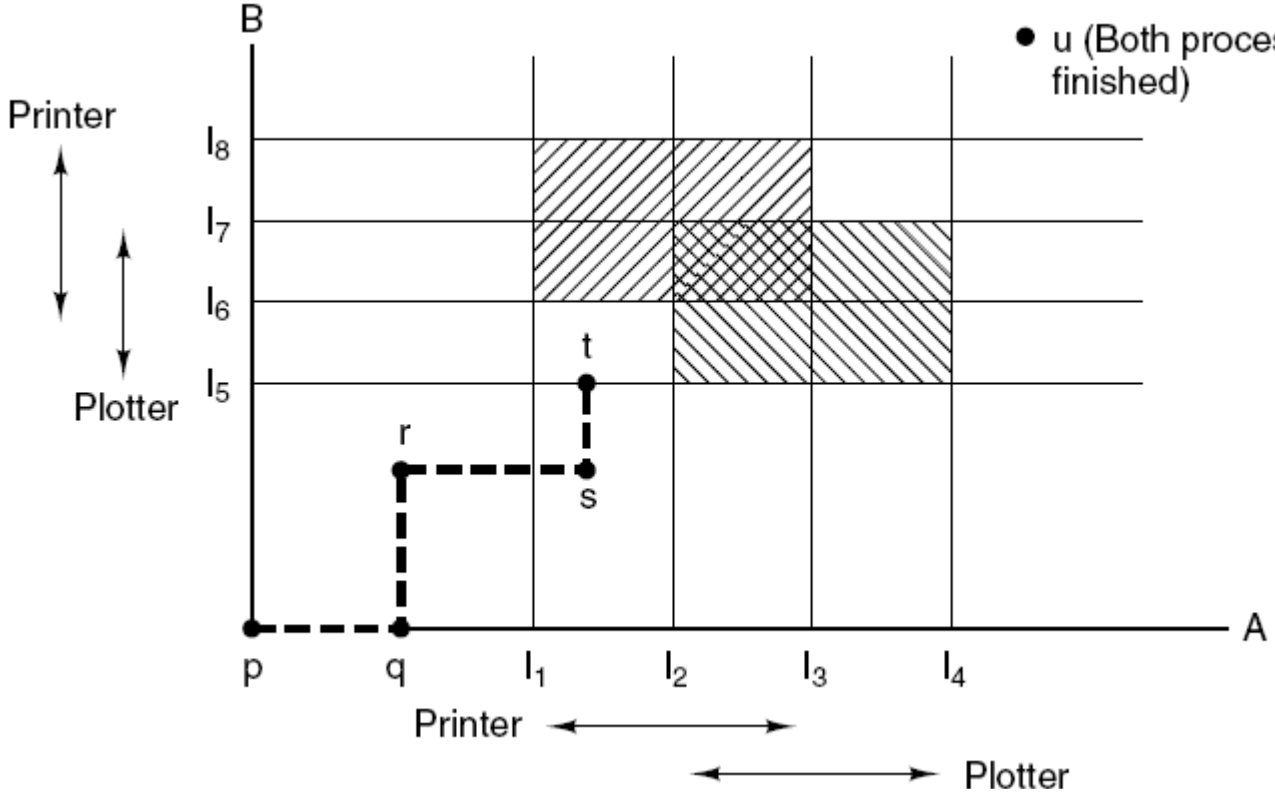
Unikanie zakleszczeń

Unikanie zakleszczeń jest zasadniczo innym podejściem, dążącym do niedopuszczenia do zakleszczenia, poprzez wykrywanie sytuacji, kiedy mogłoby do niego dojść, i zablokowanie operacji bezpośrednio do niego prowadzących.

Etap pierwszy, wykrywanie sytuacji mogących spowodować zakleszczenie, jest bardzo podobny do wcześniejszego algorytmu wykrywania wystąpienia zakleszczenia.

Natomiast etap drugi, niedopuszczenie do zakleszczenia, jest prosty i elegancki, w odróżnieniu od metod usuwania zakleszczeń. Po prostu, blokujemy krytyczną operację przydziału zasobów, i do zakleszczenia nie dochodzi. System nie musi martwić się jak proces rozwiąże problem odmowy przydziału zasobu, to jest teraz prywatny problem procesu.

Trajektorie zasobów

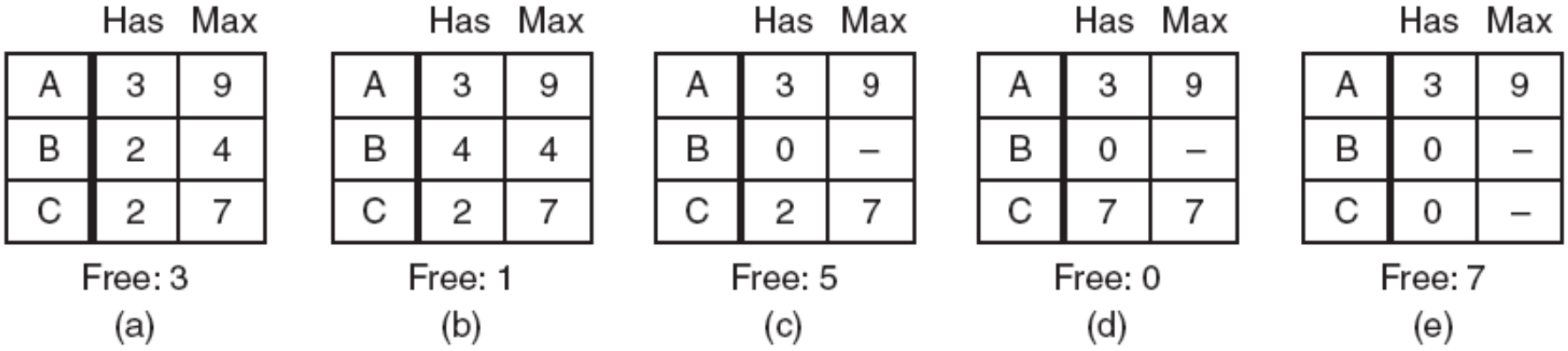


Na diagramie widać wykonywanie kolejnych instrukcji dwóch procesów: A i B. Trajektoria wykonania może biec wyłącznie do góry i w prawo. Obszary zakreślone skośnymi liniami oznaczają równoczesny przydział zasobów procesom i są wykluczone.

W momencie s proces A przydzielił już drukarkę, a w momencie t proces B żąda plotera. Jeśli to żądanie zostanie spełnione, to system wejdzie w obszar ograniczony współrzędnymi $[(I_1, I_5), (I_4, I_8)]$, z którego nie ma wyjścia, ponieważ w momencie (I_2, I_6) dojdzie do zakleszczenia. Cały ten obszar jest **niebezpieczny**.

Stany bezpieczne i niebezpieczne

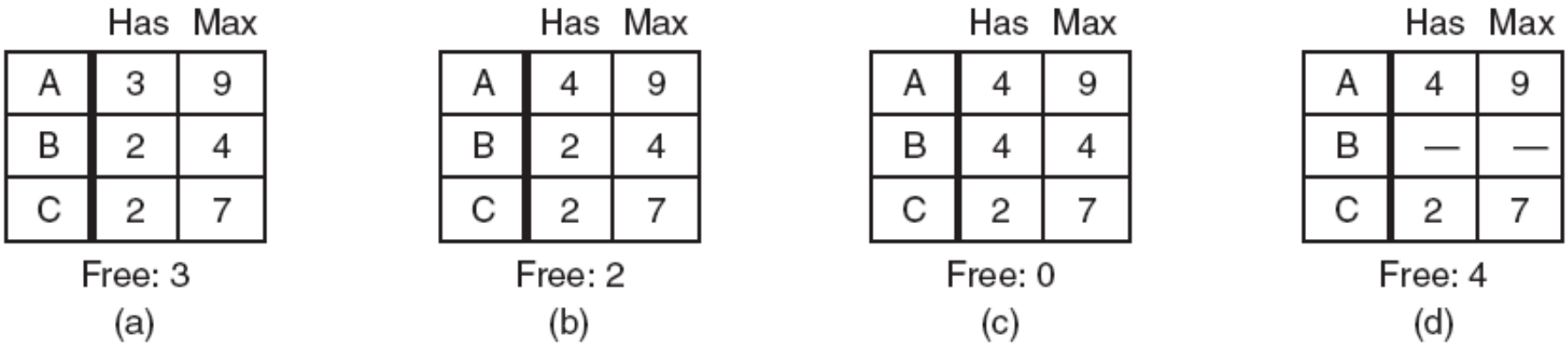
Będziemy opisywali stany przydziału zasobów dla zbioru procesów za pomocą przedstawionych wcześniej wektorów E i A oraz macierzy C i R. Stan nazywamy **bezpiecznym** jeśli istnieje pewien sposób szeregowania procesów pozwalający im wykonać się do końca, nawet jeśli jednocześnie zażądają maksymalnej liczby zasobów.



Przedstawiony powyżej przykład ilustruje przydział pojedynczego zasobu z wieloma egzemplarzami. Liczby w tabelkach wyrażają aktualny stan posiadania egzemplarzy tego zasobu, oraz maksymalny przydział dla każdego procesu. Sekwencja rysunków dowodzi, że stan przedstawiony na rysunku (a) jest bezpieczny.

Stany bezpieczne i niebezpieczne (2)

Rozważmy teraz taki sam stan początkowy, jak w poprzednim przykładzie (rysunek (a) poniżej). Jednak proces A zażądał dodatkowej jednostki zasobu, i otrzymał go (rysunek (b)). Ten stan nie jest bezpieczny.



Jedynym procesem, co do którego można mieć pewność, że poprawnie wykona się do końca jest B (rysunki (c) i (d)). Jednak nawet po wykonaniu B i zwolnieniu jego zasobów, żaden z procesów A i C nie będzie mógł poprawnie się wykonać.

Zauważmy, że stan przedstawiony na rysunku (d) nie jest stanem zakleszczenia. Nie ma wcale pewności, że do zakleszczenia dojdzie. Gdyby proces A, przed zażądaniem dodatkowych zasobów, na jakiś czas przynajmniej jeden zwolnił, proces C mógłby się poprawnie zakończyć, a po nim A. Zatem o ile stan bezpieczny gwarantuje możliwość poprawnej pracy systemu, w stanie niebezpiecznym jedynie nie ma takiej gwarancji.

Algorytm bankiera

Analizę powyższych przykładów można uogólnić do prostego algorytmu bezpiecznego przydziału zasobów.

Algorytm bankiera[2] (Dijkstra 1965) jest prostym uogólnieniem wcześniejszego algorytmu sprawdzania wystąpienia zakleszczenia. Jego działanie można sformułować za pomocą reguły:

> Dla każdego żądania przydziału, sprawdź czy prowadzi ono do stanu bezpiecznego, i gdy tak, to przydziel zasób. Gdy nie, odrzuć żądanie.

[2] Nazwa algorytmu nawiązuje do analogii bankiera przydzielającego pożyczki grupie klientów. Musi on tak gospodarować pulą posiadanych środków aby przydzielać kredyty umożliwiające działanie niektórym klientom, podczas gdy innym kredyty są wstrzymywane. Jeśli wszyscy klienci będą w stanie poprawnie zrealizować swe cele, i następnie spłacić swoje kredyty, bankier wykonał swoje zadanie.

Algorytm bankiera — przykłady dla pojedynczego zasobu

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Trzy stany alokacji zasobów: (a) bezpieczny, (b) bezpieczny, (c) niebezpieczny.

Algorytm bankiera dla wielu zasobów

Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Przedstawiony powyżej stan zasobów przydzielonych i jeszcze potrzebnych procesom jest bezpieczny. Procesy mogą wykonać się na przykład w kolejności: D, A, B, C, E.

Jeżeli w tym stanie proces B zażądałby jednej drukarki, to żądanie to można spełnić, ponieważ stan wynikowy nadal jest bezpieczny.

Jeżeli jednak następnie proces E również zażądałby drukarki, to tego żądania nie można spełnić bo stan wynikowy nie byłby bezpieczny.

Algorytm bankiera dla wielu zasobów (2)

Algorytm sprawdzania, czy stan jest bezpieczny:

1. Znajdź w macierzy R rząd i odpowiadający procesowi P_i, którego wszystkie niespełnione żądania zasobów R_i mogą być zaspokojone przez dostępne zasoby $R_i \leqslant A$.

   Jeśli nie można wybrać takiego procesu to w systemie może dojść do zakleszczenia.

2. Wybrany proces może poprawnie wykonać się do końca. Oznacz proces jako zakończony, i dodaj jego posiadane zasoby do wektora A.

3. Powtarzaj kroki 1 i 2 dopóty, dopóki albo wszystkie procesy zostaną oznaczone jako zakończone, ale pozostaną procesy, których żądań nie da się spełnić. W tym drugim przypadku mamy do czynienia z potencjalnym zakleszczeniem.

Unikanie zakleszczeń — podsumowanie

Teoretycznie, algorytm bankiera rozwiązuje problem zakleszczeń w elegancki sposób, bez uciekania się do opisanych wcześniej zabiegów usuwania powstałych zakleszczeń.

Jednak wymaga on pełnej znajomości rozkładu przyszłych żądań przydziału zasobów wszystkich procesów. Jest to możliwe raczej tylko w zamkniętych systemach, gdzie istnieje stała pula procesów, i ich charakterystyka jest znana. Ale w takich systemach również możliwe są inne rozwiązania problemu zakleszczeń, prostsze niż analiza stanów bezpiecznych. Co więcej, analiza stanów bezpiecznych może nie gwarantować poprawnego sposobu szeregowania zadań, pomimo iż system może wykonać się skutecznie. Natomiast jak wspomniano wcześniej, pełna analiza sekwencji przydziałów zasobów przy każdym nowym żądaniu może być zbyt kosztowna w skali systemu. I na koniec, szeregowanie procesów zgodne z bezpiecznym przydziałem zasobów może nie być praktyczne (może wymagać zbyt długiego wstrzymywania niektórych procesów).

Dlatego unikanie zakleszczeń ma charakter teoretyczny, niezbyt przydatny praktycznie.
Zatem, czy istnieją metody bardziej praktyczne?

Można takie zaproponować w oparciu o warunki konieczne powstawania zakleszczeń. Gdyby udało się wykluczyć przynajmniej jeden z tych warunków, to zakleszczenia nie byłyby w ogóle możliwe. Prowadzi to do **zapobiegania zakleszczeniom**.

Zapobieganie: warunek wzajemnego wykluczania

Gdyby zasoby nie były przydzielane procesom na wyłączność, do zakleszczeń nie mogłoby dojść. Jak to osiągnąć przy korzystaniu z zasobów przez wiele procesów?

Rozważmy dostęp do drukarki. Procesy mogą przydzielać ją sobie na wyłączność, ale alternatywnie drukowanie może być obsługiwane przez jeden centralny proces, zwany *spooler*em. Udostępnia on interfejs funkcji drukowania, i procesy korzystają z niego w celu skorzystania z drukarki.

To podejście można zastosować również do przydziału innych wybranych zasobów. Należy jednak uważać. Jest teoretycznie możliwe powstanie zakleszczenia przy korzystaniu z demona obsługującego dostęp do urządzenia. Jeśli demon opóźnia wykonanie operacji do momentu otrzymania wszystkich danych, a pojemność bufora ma ograniczoną, to dwa procesy mogą jednocześnie rozpocząć transmisję danych na urządzenie, ale potem opóźniać ją, powodując częściowe zapełnienie bufora, i niemożność dokończenia któregokolwiek zadania.

Zatem stosowanie tej metody warto połączyć z rozważnym korzystaniem z zasobów. Przydział zasobu powinien następować wtedy gdy jest to absolutnie konieczne, oraz gdy proces jest gotowy do szybkiego i skutecznego wykonania operacji na zasobie.

Zapobieganie: warunek wstrzymywania i oczekiwania

Jak można zapobiec temu, żeby procesy posiadające pewne zasoby, zawieszały się w oczekiwaniu na inne?

Można wymagać, aby proces przydzielił sobie wszystkie potrzebne mu zasoby przed rozpoczęciem przetwarzania. Jeśli nie będzie mógł przydzielić wszystkich zasobów, to nie przydzieli żadnych, będzie na nie czekał, nie blokując jednak innych procesów.

Jednak nie wszystkie procesy wiedzą z góry, których zasobów będą potrzebowały (gdyby wiedziały, możnaby stosować unikanie zakleszczeń za pomocą algorytmu bankiera). W innych przypadkach możemy uzyskać nieoptymalne korzystanie z zasobów. Jeśli proces najpierw długo korzysta z jednego zasobu, a potem krótko z innego, to mógłby przez długi czas niepotrzebnie blokować oba zasoby.

Rozwiązaniem może być wymaganie, aby procesy przed każdą kolejną fazą obliczeń, wymagającą innej konfiguracji zasobów, zwolniły wszystkie zasoby dotychczas przetrzymywane, i uzyskały dostęp do wszystkich aktualnie potrzebnych zasobów od nowa.

Zapobieganie: warunek braku wywłaszczania

Zanegowanie braku wywłaszczania oznacza możliwość wywłaszczania zasobów.

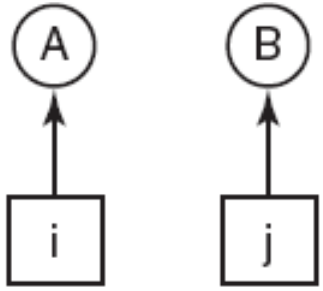
Przykładowym możliwym sposobem wywłaszczania zasobów jest by proces żądający zasobu, który nie jest obecnie dostępny, musiał zwrócić wszystkie posiadane już zasoby. Potem musiałby on od początku żądać i czekać na wszystkie zasoby. Byłby on wznowiony dopiero wtedy, gdy wszystkie te zasoby będą dostępne.

Zapobieganie: warunek cyklicznego oczekiwania

Warunek cyklicznego oczekiwania można zanegować na kilka sposobów. Na przykład, można żądać, aby każdy proces mógł przydzielić sobie tylko jeden zasób. Niestety, w większości przypadków jest to nie do przyjęcia.

Innym sposobem może być globalne ponumerowanie zasobów, na przykład jak na rysunku poniżej, oraz przyjęcie zasady: wiele zasobów może być przydzielonych, ale tylko w kolejności zgodnej z ich numerami. W tej sytuacji graf alokacji zasobów nigdy nie będzie miał cykli, i zakleszczenia nie mogą powstać.

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD-ROM drive



Rozważmy przykład na powyższym rysunku. Proces (A) ma przydzielony zasób [i] a proces (B) ma zasób (j). Do zakleszczenia mogłoby dojść, gdyby teraz proces (A) zażądał zasobu [j] a proces (B) zasobu (i). Jednak jeśli są to różne zasoby, to albo $i < j$ albo na odwrót, i powyższa sekwencja alokacji zasobów prowadząca do zakleszczenia, byłaby zabroniona.

Jest to również prawdą w przypadku zbioru procesów. Niech k będzie najwyższym numerem już przydzielonego zasobu. Proces, który go przetrzymuje, nigdy nie zażąda zasobu o numerze niższym. Najwyżej może zażądać jeszcze zasobu o numerze wyższym, ale te są wszystkie wolne. Proces ostatecznie zakończy się, i zwolni wszystkie zasoby. W tej sytuacji uwolniony będzie następny proces przetrzymujący zasób o najwyższym numerze, i on również po jakimś czasie skończy i zwolni swoje zasoby.

Algorytm pozostaje słuszny, jeśli zażądamy przydziału zasobów w kolejności niemalejących numerów (a niekoniecznie rosnących), oraz żeby proces nie mógł jedynie przydzielać zasobów o numerze większym od tych, które aktualnie posiada (a niekoniecznie tych które przydzielił wcześniej i zwolnił).

Numeracja zasobów jest rozwiązaniem problemu zakleszczeń. Jego wadą jest trudność znalezienia globalnej numeracji zasobów, która zapewniłaby optymalne działanie systemu w każdym przypadku.

Referencje

Większość materiału w tej prezentacji, w tym przykładów i rysunków, pochodzi z podręcznika Andrew S. Tanenbauma: Modern Operating Systems, Third Edition, Pearson Education, 2008, polskie wydanie: Systemy Operacyjne, Wydanie III, Helion 2010.