AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Systemy operacyjne
Wykład 05N

Wersja 2024

dr inż. Marek Wilkus **http://home.agh.edu.pl/~mwilkus**
Wydział Inżynierii Metali i Informatyki Przemysłowej
AGH Kraków

Na podstawie programu opracowanego przez dr inż. Krzysztofa Wilka

AGH

Grudzień 2020 - Wypadek na uniwersytecie w Kyoto

„The backup script includes a find command to delete log files older than 10 days. In addition to functional improvement of the script, ***the variable name passed to the find command for deletion*** *was changed to improve visibility and readability. However, there was a lack of consideration in the release procedure of this modified script. We were not aware of the side effects of this behavior and released the [updated] script,* ***overwriting [a bash script] while it was still running,*** *[…] This resulted in the reloading of the modified shell script in the middle of the execution, resulting in* ***undefined variables.*** *As a result, the original log files in /LARGE0 [backup disc storage] were deleted instead of the original process of deleting files saved in the log directory."*

- Wynik: Nieodwracalnie skasowano 77TB danych.
- Co mogło się stać?

AGH

```
#Komentarz 1
#Komentarz 2
...
sciezka=/LARGE0
rsync ... #Utworz kopie wszystkiego do $sciezka
sciezka=/LARGE0/LOG
find $sciezka -type f -ctime +10 -exec rm -f {} \;
```

- Co mogło pójść nie tak?
- Załóżmy, że jesteśmy w trakcie tworzenia kopii zapasowej (rsync). Które linia wykona się POTEM?

AGH

```
1 #Komentarz 1
2 #Komentarz 2
3 ...
4 sciezka=/LARGE0
5 rsync ... #Utworz kopie wszystkiego do $sciezka
6 sciezka=/LARGE0/LOG
7 find $sciezka -type f -ctime +10 -exec rm -f {} \;
8
```

- Załóżmy, że jesteśmy w trakcie tworzenia kopii zapasowej (rsync) - linia 5. Które linia wykona się POTEM?
- Oczywiście linia 6 - przypisanie nowej ścieżki i usuwanie starych logów z kopii.
- W trakcie kopii zapasowej jednak usunięto komentarz...

AGH

```
1 #Komentarz 1
2 ...
3 sciezka=/LARGE0
4 rsync ... #Utworz kopie wszystkiego do $sciezka
5 sciezka=/LARGE0/LOG
6 find $sciezka -type f -ctime +10 -exec rm -f {} \;
7
8
```

- W trakcie kopii zapasowej jednak usunięto komentarz…
- Pomimo tego, że nadal działa rsync, dla Basha działa linia nr 5 - czyli przypisanie. Ale ono nie wystąpi - działa rsync.
- Wykona się linia 6 - **natychmiastowe usunięcie wszystkich plików starszych niż 10 dni z całej kopii.**

AGH

Co z tego powinniśmy wiedzieć?

- Nie modyfikować skryptów w trakcie ich działania!
- Sprawdzać, na jakie ścieżki wskazują zmienne!
- Rozpatrywać przypadki awarii:
  - A co jeżeli filesystem nie jest zamontowany?
  - A co jeżeli ścieżka nie istnieje?
  - A co jeżeli nie można przejść do katalogu?
- Postępować z rm -f jak z naładowanym rewolwerem! Nigdy nie wskazywać nim miejsc, których nie chcielibyśmy usunąć!

Sygnały - jak to działa?

- Blok kontrolny procesu posiada tablicę sygnałów dostępnych do zadania oraz listę sygnałów do wykonania.
- Wysyłając sygnał system:
  - Odnajduje proces po ID na podstawie bloku kontrolnego,
  - Przegląda tablicę dostępnych sygnałów. Gdy znajdzie konkretny sygnał, odczytuje z tej tablicy adres procedury obsługi sygnału - adres w kodzie procesu.
  - Jeżeli jest równy stałej SIG_IGN, ignorujemy sygnał - nie jest on obsługiwany przez program.
  - Jeżeli sygnał to SIG_DFL, system uruchamia procedurę obsługi sygnału domyślnego nie z procesu, ale z własnych funkcji. Dzięki temu system może wykonać pewne czynności obsługi sygnału za proces.
- Adres znalezionego sygnału obsługiwanego przez program jest zapisywany do listy sygnałów do wykonania.

Teraz proces musi znaleźć się w kolejce procesów gotowych, o ile już tam nie jest bo np. czekał na coś.

Sygnały - c.d.
Zmodyfikowany proces w kolejce procesów gotowych

- Przełączenie kontekstu na proces powoduje, że system przywraca go, ale zamiast wykonywać od przerwanego miejsca, wykonuje **od funkcji obsługującej sygnał**.
- Po zakończeniu procedury obsługującej sygnał, wymuszane lub oczekiwane jest przełączenie kontekstu, a w ramach wznowienia system doprowadza blok kontrolny do porządku i wykonanie rozpoczyna się już od prawidłowego miejsca w programie.

- Skąd system wie, że funkcja sygnału się zakończyła i może "sprzątnąć bałagan"? Zależnie od systemu:
  - Adres zwrotny funkcji obsługi sygnałów jest intencjonalnie nieprawidłowy, co powoduje przerwanie.
  - Ostatnie polecenie funkcji obsługi sygnału uruchamia funkcję systemową.
  - Pamięć funkcji systemowej jest tymczasowo mapowana dla procesu, więc i system i proces mają do niej dostęp a jej adres jest ładowany na stos wywołań. Opuszczenie funkcji sygnałów powosuje powrót do funkcji systemowej. Dzięki temu funkcja sygnałów może nawet zwrócić wartość dla jądra (Linux).

Sygnały - c.d.
Proces melduje się na sygnał

- Kod procesu do uruchomienia kodu odpowiedniego sygnału wykorzystuje funkcję **signal** lub **sigaction**.
- Funkcja ta wiąże numer sygnału z podaną przez użytkownika funkcją.
- Listę sygnałów w danym systemie uzyskamy wydając polecenie **kill -l** (l jak list).
- Można uniknąć zakleszczeń oraz wykonywania odpowiedzi na sygnał w trakcie wykonywania innego sygnału nadając sygnałom blokowanie funkcją **sigprocmask**.

```
mcbx@m4800:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
mcbx@m4800:~$
```


Sygnały - Istotne sygnały

- SIGKILL - Wykonuje minimalny kod zakończenia procesu, bez zapisywania pracy czy opróżniania buforów. Nie można go przechwycić.
- SIGTERM - Wykonuje prawidłowe zakończenie programu.
- SIGSTOP - Interpretowane przez system - zatrzymanie programu. Proces aktywujemy wysyłając SIGCONT
- SIGHUP - Informuje proces, że właśnie stracił permanentnie dostęp do terminala, na którym działał.
- SIGINT - Przerwanie działania procesu. Użytkownik wcisnął Ctrl-C.
- SIGILL - Proces wykonał nieprawidłową operację.
- SIGSEGV - Proces dokonał dostępu do pamięci pod adres sobie niedostępny (i.e. innego procesu).
- SIGUSR1, SIGUSR2 - dla procedur użytkownika.

- W POSIX tylko podstawowe sygnały mają stałe numery!

Systemy rozproszone

Systemy rozproszone

Wg Wikipedii:

- **System rozproszony** to zbiór niezależnych urządzeń (komputerów) połączonych w jedną, spójną logicznie całość. Połączenie najczęściej realizowane jest przez sieć komputerową. Urządzenia są wyposażone w oprogramowanie umożliwiające współdzielenie zasobów systemowych.

- Jedną z podstawowych cech systemu rozproszonego jest jego **transparentność**, inaczej przezroczystość, która stwarza na użytkownikach systemu rozproszonego wrażenie pojedynczego i zintegrowanego systemu.

Systemy rozproszone - cechy

- Współdzielenie zasobów - wielu użytkowników systemu może korzystać z danego zasobu (np. drukarek, plików, usług, itp.)
- Otwartość - podatność na rozszerzenia, możliwość rozbudowy systemu zarówno pod względem sprzętowym, jak i oprogramowania
- Współbieżność - zdolność do przetwarzania wielu zadań jednocześnie
- Skalowalność - zachowanie podobnej wydajności systemu przy zwiększeniu skali systemu (np. liczby procesów, komputerów, itp.)
- Odporność na błędy - zdolność działania systemu mimo pojawiania się błędów (np. poprzez utrzymywanie nadmiarowego sprzętu)
- Transparentność, przeźroczystość - postrzeganie systemu przez użytkownika jako całości, a nie poszczególnych składowych.

Praktyka

- Wiele systemów realizuje **pewne cechy** systemu rozproszonego.
- Nie istnieje praktycznie stosowany system operacyjny całkowicie rozproszony.
- Wiele cech systemów rozproszonych można zaimplementować w ramach usług systemów operacyjnych lub programów narzędziowych.
  - Niektóre z tych cech są problematyczne, gdyby były implementowane w systemach – stąd lepiej, gdy są zaimplementowane w zewnętrznych programach.

Otwartość

- Usługi muszą być zgodne ze standardowymi regułami opisującymi ich składnię i semantykę (np. protokoły sieciowe).
- Specyfikacja interfejsu musi być kompletna i neutralna.

Programy od różnych dostawców muszą współpracować ze sobą, o ile spełniają warunek zgodności interfejsów.

Przenośność – aplikacja stworzona dla jednego systemu może być uruchomiona w innym bez potrzeby dokonania jakichkolwiek zmian.

Przezroczystość

- Dostępu
- Położenia
- Wędrówki (migracji)
- Przemieszczenia
- Zwielokrotnienia
- Współbieżności
- Awarii
- Trwałości

Dostępu: Przez ujednolicenie metod dostępu do danych i ukrywanie różnic w ich reprezentacji.

Przezroczystość (c.d.)

- **Położenia**: użytkownicy nie mogą określić położenia zasobu, np. na podstawie jego nazwy lub jednolitego identyfikatora.
- **Wędrówki**: można przenosić zasoby między serwerami bez zmiany odwoływania się do nich.
- **Przemieszczenia**: możliwość przenoszenia zasobów nawet podczas ich używania.
- **Zwielokrotniania**: użytkownik nie zauważa faktu zwielokrotniania zasobów
- **Współbieżności**: możliwość współbieżnego przetwarzania nie powodującego utraty spójności.
- **Awarii**: niezauważalne zastępowanie uszkodzonych węzłów.
- **Trwałości**: maskowanie sposobu przechowywania zasobu (pamięć lub dysk).

Skalowalność

- Pod względem rozmiaru (możliwość dodawania nowych zasobów i użytkowników).
- Geograficzna (rozrzucenie zasobów i użytkowników po całym świecie).
- Administracyjna (skuteczna, mimo że administracja systemem jest rozrzucona).

Skalowalność - realizacja

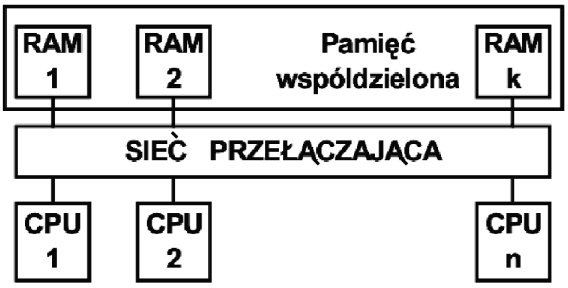
1. Ukrywanie opóźnień komunikacji
   - Komunikacja asynchroniczna
   - Część obliczeń po stronie klienta.
2. Rozproszenie (np. DNS).
3. Zwielokrotnianie
   - Równoważenie obciążenia,
   - Zwiększanie dostępności
   - Zwiększenie niezawodności
   - Caching

Problem spójności danych!

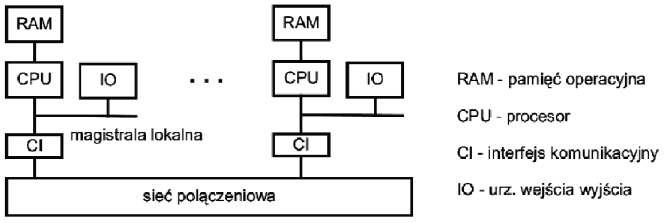
Realizacja sprzętowa

- Systemy
  - wieloprocesory (pamięć dzielona) (w tym systemy wielordzeniowe)
  - multikomputery (pamięć odrębna)
- Architektura
  - szyna
  - przełącznik

Multiprocesory (systemy wieloprocesorowe) najbardziej ogólny schemat



Multikomputery (systemy wielokomputerowe)



Systemy homogeniczne

- Sieci systemowe – grupa komputerów homogenicznych połączonych siecią
  - architektura połączeń – szyna lub przełącznik.
  - topologia połączeń – siatki i hiperkostki.
- Realizacje:
  - Procesory o masywnej równoległości (specjalna sieć). (MPP - Massive parallel processing).
  - Klastry, grupy stacji roboczych (sieć standardowa).

Oprogramowanie

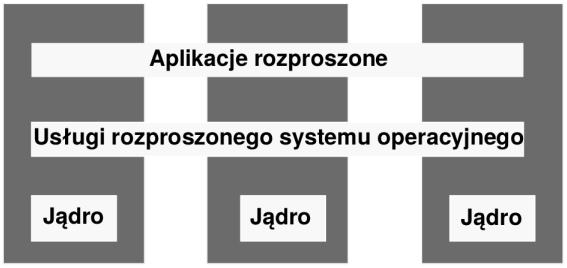
1. Systemy operacyjne dla komputerów rozproszonych:

   a) ściśle powiązane (zarządzanie wszystkimi globalnymi zasobami przez system) – w wieloprocesorach, komputerach homogeniczych.
   - Ukrywają rozproszenie i zarządzają zasobami sprzętowymi.

   b) luźno powiązane (zbiór współpracujących komputerów z lokalnymi OS) – sieciowe systemy operacyjne.
   - Oferują lokalne usługi klientom zdalnym.

2. Oprogramowanie warstwy pośredniej...
   - ...zapewnia przezroczystość rozproszenia.

Wielokomputerowy system operacyjny



Rozproszona pamięć dzielona

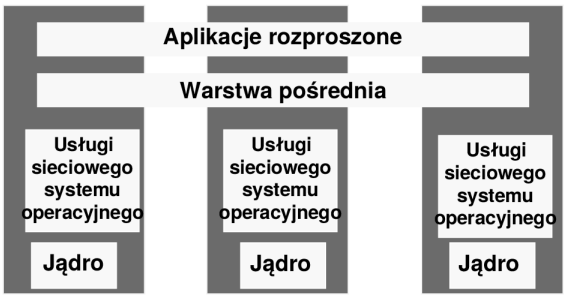
- Stronicowana rozproszona pamięć dzielona jako forma komunikacji.
- Problemy z efektywnością:
  - zwielokrotniania stron tylko do ich odczytu,
  - zwielokrotnianie wszystkich stron bez ich użytkowania,
  - rezygnacja ze ścisłej spójności,
  - fałszywe dzielenie (false sharing - gdy 2 procesy na dwóch procesorach odwołują się do różnych zmiennych, ale na tej samej stronie, strona ta ciągle „wędruje" od procesora do procesora). Naprawiane przez dostosowanie rozmiaru strony.

Sieciowy system operacyjny

Usługi sieciowego systemu operacyjnego

- Praca zdalna (np. rlogin, Remote Desktop)
- Kopiowanie plików (np. rcp, SFTP)
- Zdalne uruchamianie oprogramowania i lokalna obsługa jego interfejsu (np. X forwarding),
- Sieciowy system plików
  - Serwer
  - Klient

Oprogramowanie warstwy pośredniej (middleware)



Warstwa pośrednia

- Sieciowy system operacyjny nie oferuje przezroczystości rozproszonego systemu operacyjnego.
- Uzyskanie systemu rozproszonego wymaga wprowadzenia dodatkowej warstwy oprogramowania - warstwy pośredniej, nadbudowującej nad usługami sieciowymi usługi dla systemu rozproszonego.

Warstwa pośrednia (c.d.)

- Sieciowe systemy operacyjne w zakresie komunikacji oferują interfejs gniazd (ang. sockets), umożliwiający komunikację pomiędzy rozproszonymi procesami, ale wymagający wskazania lokalizacji poszczególnych procesów (np. poprzez adresy IP).
- W systemie rozproszonym warstwa pośrednia może dostarczać mechanizmów transparentnej komunikacji, w której procesy identyfikowane są w sposób abstrakcyjny, niezależny od lokalizacji procesów.

Założenia do warstwy pośredniej

- Wszystko jest plikiem (z Unixa) – komunikacja jako zapis/odczyt pliku.
- Zdalne wywołania procedur (RPC) – procedury zdalne jak lokalne (ukrywanie komunikacji).
- Obiekty rozproszone – obiekt na jednej maszynie, interfejs do niego na wielu.
- Model dokumentów rozproszonych (WWW).
  - Wiele cech przydatnych w WWW było już testowanych i nie rozszerzyło standardu (NLS - 1960s, Memex - 1950s). Np. współpraca, dwustronne łącza, podejście client-agnostic.

Usługi warstwy pośredniej
Co zapewnia warstwa pośrednia?

- Komunikacja – RPC, zdalne obiekty, przezroczysty dostęp do rozproszonych plików, baz danych, dokumenty WWW.
- Nazewnictwo – lokalizacja zasobów – skalowalność.
- Trwałość – pliki, bazy danych, rozproszona pamięć dzielona.
- Transakcje rozproszone – atomowość, dane na wielu maszynach, maskowanie awarii.
- Bezpieczeństwo.

Otwartość warstwy pośredniej

- Nadbudowa nad systemem – uniezależnienie od systemu. Middleware jest często osobnym programem/pakietem.
- Zależność aplikacji od warstwy pośredniej, nie specyficznych funkcji systemu.
- Gdy występuje niekompletność interfejsów warstwy pośredniej – istnieje konieczność odwoływania się bezpośrednio do systemu.
- Zgodność warstwy pośredniej ze standardem, ale nieprzenośność aplikacji.
  - np. serwer baz danych na różnych platformach udostępnia to samo API, ale nie ma pełnej binarnej zgodności.

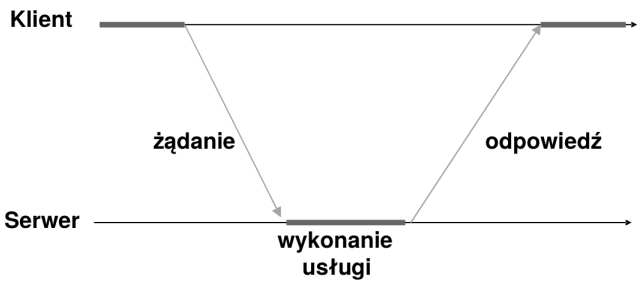
Przykłady warstw pośrednich

- Gniazda (ang. sockets)
- RPC (Remote Procedure Call)
- DCE (Distributed Computing Environment)
- CORBA (Common Object Request Broker Architecture)
- DCOM (Distributed Component Object Model)
- RMI (Remote Method Invocation)

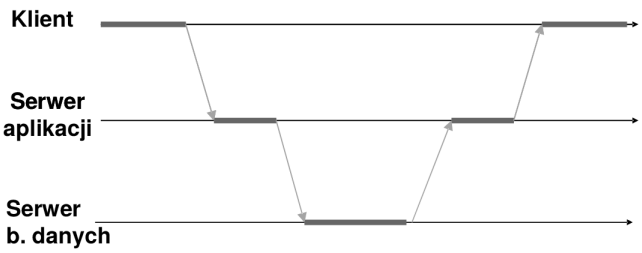
Porównanie

	Rozproszony OS		Sieciowy OS	Warstwa pośrednia
	Wieloproc.	Wielokomp.		
Przezroczystość	b. duża	duża	mała	duża
Jeden OS?	tak	tak	nie	nie
Bez kopiowania?	tak	nie	nie	nie
Komunikacja	pamięć dzielona	komunikaty	pliki	zależna od modelu
Zarządzanie zasobami	globalne, centralne	globalne, rozproszone	lokalne	lokalne
Skalowalność	nie	umiarkowana	tak	zmienna
Otwartość	zamknięty	zamknięty	otwarty	otwarty

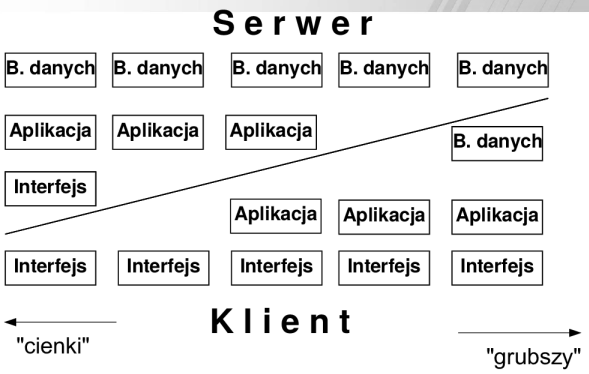
Model Klient-serwer



Model trójwarstwowy



Co ma robić warstwa klienta?

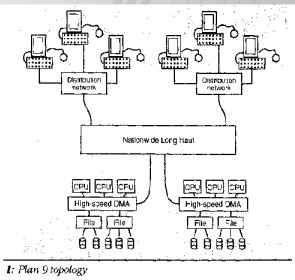


W praktyce: Plan 9

- Zauważmy, że jeżeli bardzo przyłożymy się do realizacji zasady „wszystko jest plikiem”, to kwestia rozproszonego systemu operacyjnego zawiera się w odpowiednim systemie plików.
- Na tej zasadzie działa system operacyjny Plan 9, gdzie system plików jest oparty o wiadomości. Te wiadomości można przesyłać przez różne kanały komunikacji.
- Dodatkowo każdy proces posiada informacje o systemie plików dla siebie → czyli każdy proces może nieco inaczej widzieć system plików (a więc i np. urządzeń!).
- Możliwe jest scalenie wielu katalogów w jeden (union directory) na potrzeby perspektywy systemu lub jednego procesu → jeszcze większa przezroczystość.

Plan 9 - architektura

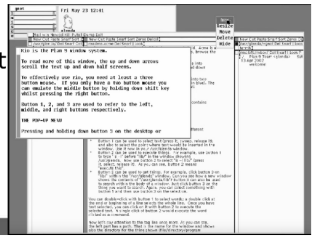
- Każde urządzenie jest reprezentowane jako plik.
- Każdy plik ma swoje miejsce w systemie.
- Każdy proces „widzi” system plików w specyficzny dla siebie sposób.
- Ponieważ jako „urządzenie” możemy rozumieć zarówno np. dysk sieciowy jak i procesor drugiego komputera, program może działać korzystając z CPU jednego komputera, całościowo używać pamięci innego (niezbędne elementy sprowadzane są do komputera z CPU), a wyniki zapisywać na dysk jeszcze innego.



1: Plan 9 topology

Plan 9 - interfejs

- Działa na tym powłoka uniksowa.
- Możliwe jest użytkowanie systemu okienkowego RIO.
- Proces jest katalogiem.
- Katalog-proces zawiera pliki. Przez wysyłanie do nich odpowiedniej zawartości sterujemy procesami.
- Tak samo sterujemy urządzeniami - każdym…
- ...niezależnie od tego czy jest częścią komputera, czy jest zdalne.



Systemy czasu rzeczywistego

Systemy czasu rzeczywistego

- Definicja IEEE:

  System czasu rzeczywistego (real time) to system, którego poprawność działania zależy nie tylko od poprawności logicznych rezultatów, lecz również od czasu, w jakim te rezultaty są osiągane (czasu reakcji).

Systemy czasu rzeczywistego (RTOS)

- Tryb przetwarzania w czasie rzeczywistym jest takim trybem, w którym programy przetwarzające dane napływające z zewnątrz są zawsze gotowe, a wynik ich działania jest dostępny nie później niż po zadanym czasie. Moment nadejścia kolejnych danych może być losowy (asynchroniczny) lub ściśle określony (synchroniczny).
- System czasu rzeczywistego jest systemem interaktywnym, który utrzymuje ciągły związek z asynchronicznym środowiskiem, np. środowiskiem, które zmienia się bez względu na system, w sposób niezależny.
- Oprogramowanie czasu rzeczywistego odnosi się do systemu lub trybu działania, w którym przetwarzanie jest przeprowadzane na bieżąco, w czasie wystąpienia zewnętrznego zdarzenia, w celu użycia rezultatów przetwarzania do kontrolowania lub monitorowania zewnętrznego procesu.

Rys historyczny

- W pierwszych komputerach oprogramowanie miało pełną kontrolę bezpośrednio nad sprzętem - każdy program odpowiednio napisany był RTOS.
- Późniejsze systemy oferowały możliwości wywłaszczenia systemu przez oprogramowanie. Nie jest to najbezpieczniejsza możliwość, lecz umożliwia np. Zmianę systemu operacyjnego bez przeinicjowania komputera (MonkeyLinux). Programy, które musiały działać jako RT wykorzystywały tą możliwość.
- Wprowadzenie pełnej wielozadaniowości stworzyło potrzebę systemów RT oraz systemów, w których tylko niektóre zadania są RT.

Rys historyczny - wielozadaniowość

- RTOS musi używać wywłaszczania w celu umożliwienia działania procesu RT.
- Proces RT może uruchamiać się z b. wysokim priorytetem (w systemach typu Flex 9) lub informować system o swoim stanie, działając na poziomie jądra (adaptacje Uniksów).
- Systemy „Event-driven” - możliwe do zaimplementowania na mikrokontrolerach. Zdarzenie wyzwala przerwanie, wyzwalające proces RT.
  - Problem: „zapchanie” przerwaniami, patrz zawieszenie komputera podczas lądowania na Księżycu w misji Apollo 11

Rys historyczny

- Systemy „Time Sharing” - z podziałem czasu - umożliwiają płynne mieszanie zadań RT i pozostałych.
  - Problem: Wymagane jest częste zwolnienie mniej istotnych zadań przez zadanie RT.

Systemy czasu rzeczywistego

- System czasu rzeczywistego odpowiada w sposób przewidywalny (w określonym czasie) na bodźce zewnętrzne napływające w sposób nieprzewidywalny.
- System komputerowy działa w czasie rzeczywistym, jeżeli wypracowane przez ten system decyzje są realizowane w tempie obsługiwanego procesu. Inaczej mówiąc, system działa w czasie rzeczywistym, jeżeli czas reakcji systemu jest niezauważalny przez proces (decyzja jest wypracowana we właściwym czasie) **

Wg:
Lal K., Rak T., Orkisz K : "RTLinux – system czasu rzeczywistego", HELION, 2003.
** - Plaza R., Wróbel E.: „Systemy czasu rzeczywistego", Wydawnictwo Naukowo – Techniczne, Warszawa 1988.

RTOS - przykłady

- Sekwencja awaryjnego wyłączania silnika rakietowego.
- System zbierania danych (np. pomiarów w procesie produkcyjnym).
- System kontrolny ABS w samochodzie.
- System dostarczania paliwa do silników samolotu.
- Odtwarzanie plików MPEG w stacjonarnych odtwarzaczach.
- Kontroler serwomechanizmu.
- Systemy podtrzymywania życia w urządzeniach medycznych

Czy RTOS muszą być szybkie?

- **Real time** oznacza nie "szybki", lecz "przewidywalny".
- **Gwarantowany pesymistyczny czas reakcji** nie oznacza szybkiego czasu reakcji, a jedynie czas reakcji z góry określony.
- System czasu rzeczywistego może wydawać się wolniejszy od "zwykłego" systemu operacyjnego. Wynika to z faktu, ze techniki stosowane do przyspieszania pracy systemu operacyjnego (pamięć podręczna, wielopotokowe procesory, etc.) wprowadzają element indeterminizmu. Indeterminizm jest niedopuszczalny w przypadku systemu czasu rzeczywistego, gdyż uniemożliwia zapewnienie przewidywalności systemu.

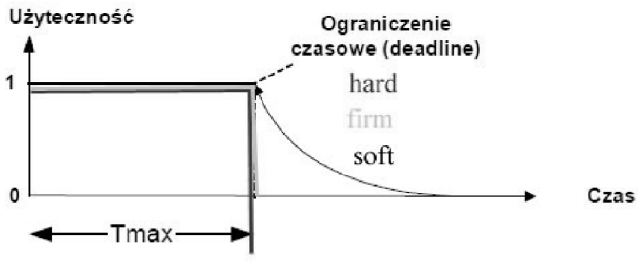
Podział RTOS

- Hard (rygorystyczne)
  - Gwarantują terminowe wypełnianie krytycznych zadań. Wymaga to ograniczenia wszystkich opóźnień w systemie.
  - Pamięć pomocnicza jest na ogół bardzo mała albo nie występuje wcale. Wszystkie dane są przechowywane w pamięci o krótkim czasie dostępu lub w pamięci, z której można je tylko pobierać (ROM).
  - Prawie nie spotyka się w systemach czasu rzeczywistego pamięci wirtualnej.
  - Dlatego rygorystyczne systemy czasu rzeczywistego pozostają w konflikcie z działaniem systemów z podziałem czasu i nie wolno ich ze sobą mieszać.

Podział RTOS

- Soft (łagodne)
  - Krytyczne zadanie do obsługi w czasie rzeczywistym otrzymuje pierwszeństwo przed innymi zadaniami i zachowuje je aż do swojego zakończenia.
  - Opóźnienia muszą być ograniczone - zadanie czasu rzeczywistego nie może w nieskończoność czekać na usługi jądra.
  - Łagodne wymagania dot. czasu rzeczywistego umożliwia godzenie ich z systemami innych rodzajów.
  - Zastosowanie w technikach multimedialnych, kreowaniu sztucznej rzeczywistości itd.
  - Znajdują one swoje miejsce wszędzie tam, gdzie istnieje potrzeba systemów o bardziej rozbudowanych możliwościach.
- Firm (mocne)
  - Wymagania pośrednie pomiędzy hard a soft,
  - Nie wykonanie zadania w terminie skutkuje nieprzydatnością wyników, ale nie zagraża katastrofą.

Użyteczność odpowiedzi



Proces RT w systemie standardowym

Opóźnienia (w sytuacji, gdy łatwo można je zmierzyć)

Przykładowe czasy opóźnień dla różnych systemów operacyjnych (Pentium 100MHz)

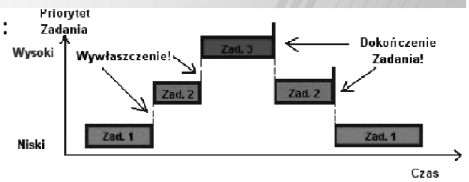
System	Tryb pracy	Opóźnienie
Windows 98/ 2000/ XP	in real time	100µs – 100ms
Linux	soft real-time	1ms
Linux IEEE 1003.1d	hard real-time	10µs – 100µs
Linux RT	hard real-time	1µs – 10µs
Jądro RTOS	hard real-time	1µs – 10µs

Post Scriptum (2024)

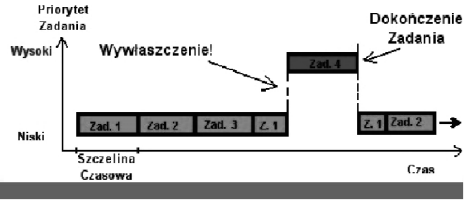
- Linux w trybie PREEMPT_RT – 4-400µs, ale przy równolegle działających zadaniach nie-RT!
  - Dużo zależy od istniejących procesów RT i nie-RT, konfiguracji sprzętowej, zajętości CPU i intensywności wykorzystania urządzeń wejścia-wyjścia.
  - Znaczny wpływ zadania RT na inne zadania.
  - System wydaje się w ogólnym odczuciu wolniejszy od nie-RT, ale nie dla zadania RT.

Szeregowanie zadań

- Priorytetowe:



- Z podziałem czasu:

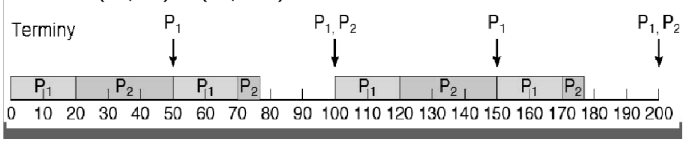


Planowanie procesów

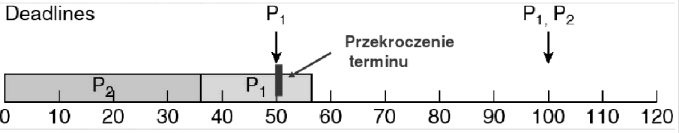
- Przyjmujemy periodyczny (okresowy) model procesów.
- Każdy proces może być opisany przez następujące parametry:
  - Okres **p** (period) tzn. czas pomiędzy kolejnymi zdarzeniami wymagającymi obsługi przez proces.
  - Termin **d** (deadline) w którym zdarzenie musi być obsłużone (od momentu zajścia zdarzenia).
  - Czas **t** (time) potrzebny procesowi na obsługę zdarzenia.
- Zachodzi relacja **0≤t≤d≤p**
- Stopień wykorzystania procesora jest równy **u=t/p**.
- Warunek konieczny wykonywalności szeregowania: suma stopni wykorzystania procesora **Σu≤1**.
- Proces oznajmia swoje parametry t,d,p planiście. Planista albo podejmuje się wykonania procesu gwarantując dotrzymania terminu albo odrzuca proces.

Rate-monotonic scheduling

- Procesy są planowane na podstawie statycznego priorytetu równego częstotliwości zdarzeń 1/p.
- Proces o wyższym priorytecie wywłaszcza proces o niższym priorytecie.
- Przykład: dwa procesy:
  - P1: p=50, d=50, t=20 (P1 ma krótszy okres => większą częstotliwość i priorytet).
  - P2: p=100, d=100, t=35
  - Całkowite obciążenie procesora (20/50) + (35/100)=0.75



- Spośród klasy algorytmów z priorytetami statycznymi jest to algorytm optymalny, w takim sensie, że jeżeli nie dotrzymuje terminów, to żaden inny algorytm z tej klasy również nie dotrzyma terminów.
- Przykład: zakładamy, że proces P2 ma większy priorytet:



- Przykład, w którym dotrzymanie terminów nie jest możliwe:
  - P1: p=50, d=50, t=25 (wyższy priorytet)
  - P2: p=80, d=80, t=35.
- Całkowite wykorzystanie procesora
  (25/50)+(35/80)=0.94.
- Wydaje się że procesy można zaplanować, ale:

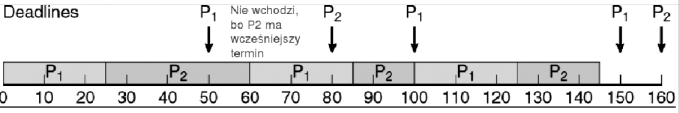
- W pesymistycznym przypadku algorytm nie gwarantuje dotrzymania terminów, gdy całkowite wykorzystanie procesora:

$$u \geq 2(2^{1/N}-1)$$

- Dla liczby procesów N=2 otrzymujemy u≈0.83

Algorytm "Najpierw najwcześniejszy termin" (EDF – Earliest deadline first)

- Przykład:
  - P1: p=50, d=50, t=25 (wyższy priorytet)
  - P2: p=80, d=80, t=35.
- Priorytet przypisywany dynamicznie:



RTOS - Wymagania

- RTOS musi być wielowątkowy i wywłaszczalny.
- W momencie gdy OS nie jest oparty na deadlinach, musi istnieć pojęcie priorytetu wątku.
- OS musi wspierać mechanizm przewidywalnej synchronizacji wątków.
- Musi istnieć dziedziczenie priorytetów.
- Zachowanie OS powinno być znane i przewidywalne.

Dokładne oszacowanie parametrów czasowych

- Opóźnienie przerwań (czyli czas od wygenerowania przerwania do rozpoczęcia wykonywania zadania) - musi być zgodne z wymaganiami aplikacji, i musi być przewidywalne. Wartość ta zależy od liczby jednocześnie oczekujących przerwań.
- Dla każdego przerwania systemowego – maksymalny czas, jaki zajmujemy. Czas powinien być przewidywalny i niezależny od liczby obiektów w systemie.
- Maksymalny czas na jaki OS i sterowniki maskują przerwania.

Tworzenie RTOS z istniejącego systemu

W niektórych przypadkach, aby uzyskać RTOS, podejmuje się próby modyfikacji lub wykorzystania istniejących systemów operacyjnych. Obserwuje się dwa główne podejścia do tej kwestii:

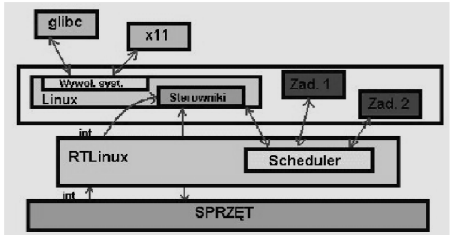
1) Próby balansowania pesymistycznego i średniego czasu reakcji, "robienie ogólno-zadaniowego systemu operacyjnego a`la real-time".
Próby optymalizowania dwóch, zasadniczo przeciwstawnych, parametrów rzadko prowadzą do olśniewających rezultatów, a w przypadku systemów operacyjnych prowadzą raczej do soft RTOSów.

Tworzenie RTOS – drugie podejście

2) Rozbicie systemu na dwa systemy operacyjne - real-time i "zwyczajny" (nie real-time).
Podejście to wydaje się być często skuteczne, czego przykładem jest choćby RTLinux. Niemniej jednak uważane jest za konserwatywne i potencjalnie ograniczające możliwości końcowego użytkownika.

- Windows mogą, przy użyciu specjalnego oprogramowania, być konwertowane do systemów real-time.
Niestety, jedyne co udaje się osiągnąć to soft real-time.

RTLINUX



RTLinux – procesy czasu rzeczywistego

- Procesy czasu rzeczywistego to programy zdefiniowane przez użytkownika, które wykonują się zgodnie z podanym harmonogramem (mogą być okresowe, zasypiać się na określony czas) i mają ścisłe wymagania czasowe.
- Procesy czasu rzeczywistego implementuje się jako ładowalne moduły jądra. Z punktu widzenia RTLinuksa procesy RT to wątki jądra RT.
- Scheduler RTLinuksa uważa, że jest tylko jeden prawdziwy proces, który ma wiele wątków. Jeden z tych wątków jest wybierany do wykonania.
- Linux jest tylko jednym z wątków, ma najniższy priorytet.

RTLinux – Procesy czasu rzeczywistego

- Procesy wykonują się w jednej przestrzeni adresowej, zatem przy zmianie kontekstu nie trzeba unieważniać rejestrów asocjacyjnych (TLB – translation look-aside buffers). Gdyby procesy wykonywały się we własnej przestrzeni adresowej, przy każdej zmianie kontekstu trzeba by unieważniać rejestry TLB, co przy częstym przełączaniu kontekstu daje duże obniżenie wydajności i utrudnia przewidywalność.
- Przy wywołaniach systemowych nie trzeba zmieniać poziomu uprzywilejowania.

RTLinux – Procesy czasu rzeczywistego

- Przełączanie kontekstu jest łatwiejsze - przełącza się tylko kontekst sprzętowy: zachowuje się zawartość rejestrów na stosie i zmienia wskaźnik stosu tak, żeby wskazywał nowy stos nowego procesu.
Przełączanie kontekstu jest programowe, a nie sprzętowe, bo sprzętowe przełączenie kontekstu na procesorach x86 jest powolne.
- Program i jego dane nie podlegają stronicowaniu. Nie mogą zatem zostać wysłane na dysk. Nie występują błędy braku strony, więc nie ma opóźnień z tym związanych.

RTLinux – Procesy czasu rzeczywistego

- Błąd w programie użytkownika, jakim jest proces RT, może spowodować załamanie się systemu. Pisanie programów RT wymagana takiego samego natężenia uwagi, co programowanie jądra systemu operacyjnego.
Czytaj: **Procesy RT działają jak wątki jądra.**
- Dodatkowym ograniczeniem nałożonym na procesy RT jest fakt, że ich zasoby są definiowane statycznie. W szczególności nie ma (w standardowym RTLinuksie) wsparcia dla dynamicznej alokacji pamięci.
- W nowszych wersjach RTLinuksa dostępny jest moduł mbuff, który umożliwia korzystanie z dynamicznie alokowanej pamięci kosztem przewidywalności.

QNX
Klasyczny UNIX + RT

- Został opracowany na początku lat 80 przez założycieli kanadyjskiej firmy Quantum Software System, Limited
- Jego początkowa nazwa QUNIX (Quick UNIX), ze względu na zbyt duże podobieństwo do nazwy UNIX, nie mogła przetrwać zbyt długo i w kilka miesięcy później, za sprawą firmy AT&T, musiała zostać zmieniona i przybrała znane do dzisiaj brzmienie: QNX.
- Pierwsza wersja, była przeznaczona na komputery klasy IBM PC oraz wymagała 64kB pamięci RAM i 180kB napęd dyskietek. Pomimo swojej prostoty, umożliwiała już uruchomienie kilku procesów jednocześnie.
- *"gdyby firma IBM wybrała system QNX dla mikrokomputera IBM PC, wprowadzenie na rynek modelu AT mogłoby zostać opóźnione, gdyż aplikacje uruchamiane pod systemem QNX w komputerach PC zachowują się tak, jakby zostały uruchomione pod systemem DOS w komputerze 386"*

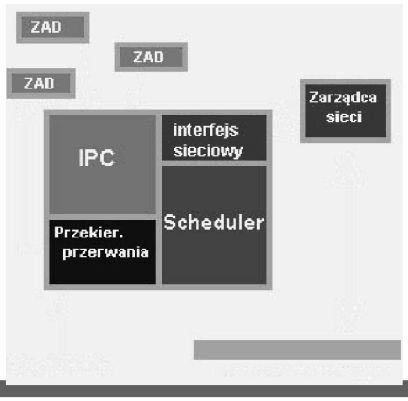
QNX - mikrojądro

- Rozmiar ok. 8kB (jądro UNIX > 700kB), stąd nazwa mikrojądra - Neutrino.
- To, co odróżnia ten system ten od rodziny UNIX, to przede wszystkim struktura modułowa oraz architektura oparta o przesyłanie komunikatów (klient - serwer).
- QNX daje możliwość zdeterminowania czasu reakcji na zdarzenia występujące w systemie.
- Dzięki rozbudowanym możliwościom definiowania priorytetów, QNX jest stosowany jako system służący do sterowania automatyką przemysłową, gdzie pewne zdarzenia są krytyczne (np. otwarcie zaworu bezpieczeństwa w zbiorniku kiedy gwałtownie wzrasta ciśnienie) i muszą być zawsze obsłużone na czas.

Budowa mikrojądra w QNX

- IPC - (ang. Interprocess communication) – obsługuje komunikację między procesami.
- Network Interface - przeźroczysta komunikacja pomiędzy procesami w obrębie sieci lokalnej.
- Hardware Interrupt Redirector – przechwytuje pojawiające się przerwania oraz przekazuje je do odpowiednich procesów obsługujących je. Część ta sama nie obsługuje przerwań!
- Realtime Scheduler - decyduje, który proces ma uzyskać dostęp do procesora w danej chwili (POSIX 1003.4 - dotyczy zagadnień czasu rzeczywistego).

Budowa mikrojądra QNX



Mikrojądro QNX

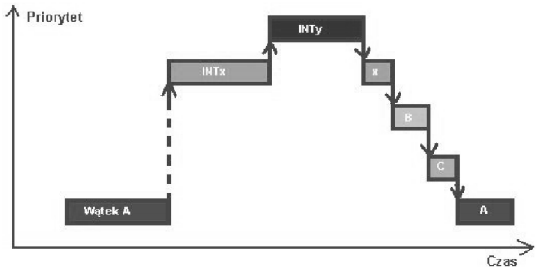
- Procesy zarządzające są traktowane w identyczny sposób jak procesy użytkowe. Można je ładować, uruchamiać, zawieszać oraz usuwać niezależnie od siebie. Poza tym czynności te mogą być wykonywane dynamicznie w czasie normalnej pracy system.
- W zależności od wymagań zewnętrznych dany proces zarządzający może zostać zainstalowany bądź usunięty. Wyjątkiem jest tutaj tzw. **Process Manager**, który musi być zawsze obecny w systemie.
- Procesy zarządzające od procesów użytkowych odróżniają priorytety. P.z. mają najwyższe. Dodatkowo zyskują poziomy uprzywilejowania, które zezwalają im na realizację niektórych instrukcji mikroprocesora.
- W systemie QNX zaimplementowano trzy poziomy uprzywilejowania:
  - poziom 0 - mikrojądro
  - poziom 1 - procesy zarządzające (np. Proc, Fsys itp.)
  - poziom 2 - nie jest aktualnie wykorzystywany
  - poziom 3 - procesy użytkowe

Szeregowanie zadań w QNX

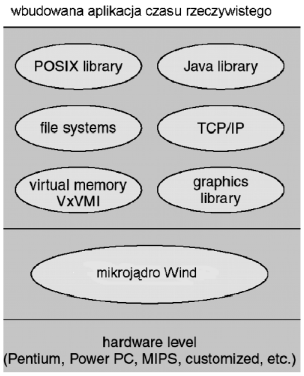
- Algorytmy szeregujące wybierają zadanie gotowe do wykonania i najważniejsze, czyli to o najwyższym priorytecie w danej chwili (priorytet 0÷31).
- Są trzy algorytmy szeregowania zadań (o tym samym priorytecie).

1) Kolejka FIFO. Zadanie wybrane do wykonywania, kontynuuje swoje działania aż samo odda sterowanie (np. wątek zostanie zablokowany lub proces wywoła funkcję systemową) bądź zostanie wywłaszczone przez zadanie o wyższym priorytecie.
2) Przydział czasu procesora (ang. time slice), który wynosi 50ms. Wówczas zadanie zostaje wywłaszczone, poza warunkami opisanymi we wcześniejszym algorytmie, również wtedy, kiedy wykorzysta przydzielony mu czas.
3) Adaptacyjny (ang. adaptive scheduling). Jeśli dane zadanie wykorzysta przydzielony czas procesora i nie zostanie zablokowane, to jego priorytet jest dekrementowany. Wtedy zazwyczaj następuje przełączenie kontekstu do innego zadania. Oryginalna wartość priorytetu jest natychmiast przywracana, kiedy zadanie przechodzi do stanu blokowania. Algorytm ten znajduje zastosowanie w sytuacjach, kiedy zadania wykonujące ogromne ilości obliczeń dzielą czas procesora z zadaniami interaktywnymi.

Obsługa przerwań w QNX



VxWorks 5.x



VxWorks 5.x

- Opcjonalne podsystemy: grafiki, systemy plików, Java, sieci TCP/IP, biblioteka Posix, pamięć wirtualna. Pozwala to na minimalizację zajętości (ang. footprint) pamięci.
- Mikrojądro (ang. microkernel) Wind
  - wywłaszczalne
  - gwarantowany czas reakcji na przerwania
- Zastosowanie: samochody, urządzenia konsumenckie, przełączniki sieciowe oraz Marsjańskie łaziki Spirit i Opportunity.

FreeRTOS

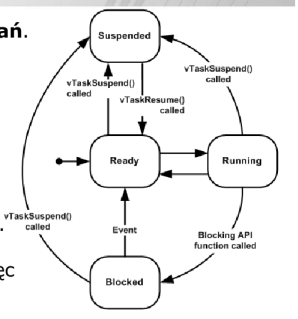
- Wielowątkowy, przeznaczony na systemy wbudowane i mikrokontrolery RTOS.
- Wspiera tryby oszczędzania energii, również działanie „uśpienia” gdy system aktywowany jest przerwaniem.
- Każdy proces czasu rzeczywistego może mieć priorytet.
- Możliwe jest decydowanie o postępowaniu z procesami RT.

Procesy RT w FreeRTOS

- Allocate only - obiekt jest ładowany do pamięci i tam pozostaje. Nie ma strat czasu wynikłych z jego przemieszczania lub przywracania.
- Allocate and free - konwencjonalna, lecz uproszczona możliwość ładowania i zwalniania pamięci np. przez zmienne.
- Blokowe „Allocate and free” - bardziej złożony algorytm likwidujący fragmentację kosztem czasu.
- Jak wyżej, lecz z możliwością dzielenia stosu programów na różne bloki pamięci
- Konwencjonalna alokacja i zwalnianie pamięci.

Zadanie w FreeRTOS

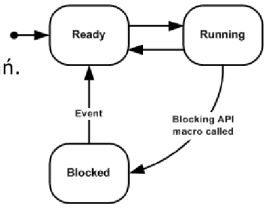
- Proces RT składa się z zestawu **zadań**.
- Każde zadanie uruchamia się we własnym kontekście, unikając zależności od stanu innych zadań lub systemu.
  - ...a więc każde zadanie ma własny stos - więcej zajętej pamięci.
- W urządzeniach o mniejszej ilości RAM rolę zadań pełnią **co-routines**. Są one ograniczone pod względem własnego stosu (dzielą wspólny) więc i wywołań innych funkcji, także systemowego API.



- Każde zadanie może znajdować się w konkretnych stanach.

Zadanie (Task) vs co-routine

- Co-routine może być gotowe, działać lub być zablokowana. Nie ma wstrzymywania jak w przypadku zadań.
- Co-routine nie ma własnego stosu. Stąd ograniczone jest wołanie API systemu i innych funkcji.
- Tak jak i zadania, co-routines mogą mieć priorytety, lecz są one zawsze poniżej priorytetów zadań!
  - → Co-routine wykona się gdy nie istnieje żadne zadanie o priorytecie wyższym niż zadanie „bezczynności”.
- Współdzielony stos nie gwarantuje stabilności zmiennych. Najprawdopodobniej zmienne alokowane w jednym uruchomieniu co-routine nie będą zachowane w następnym.



Dziękuję za uwagę