Systemy wbudowane - wykład do samodzielnego opracowania
RToS (4)

Przemek Błaśkiewicz

20 maja 2020

System czasu rzeczywistego – QNX

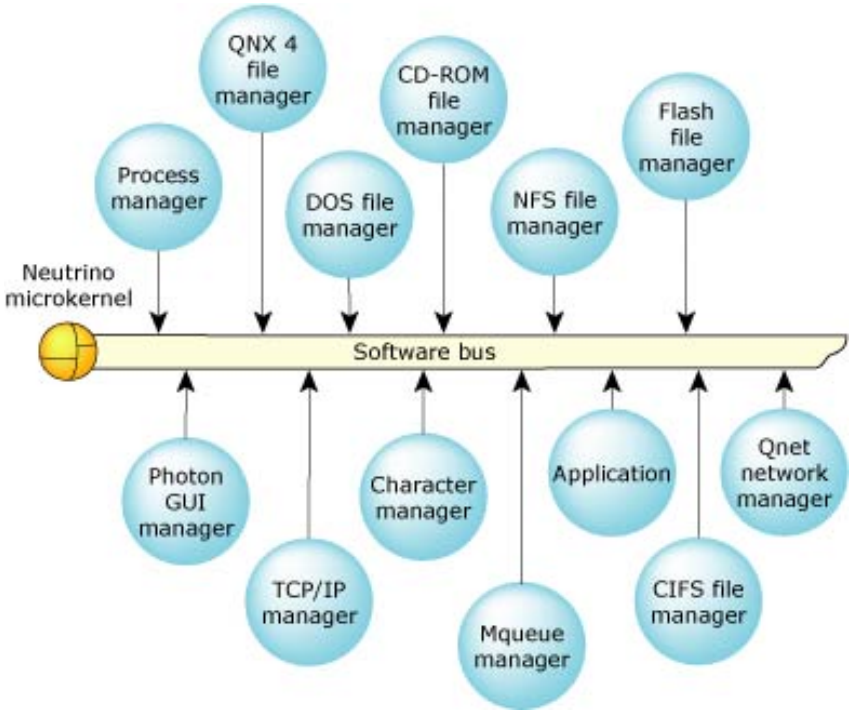
Jako przykład komercyjnego RTOS (i ciekawy ogólnie przypadek OS) omówimy ogólne właściwości systemu QNX w kontekście szeregowania zadań. Trochę historii:

- Początkowo zwany Quantum UNIX, przemianowany na QNX
- Pierwsze sukcesy w początkach lat '80.
- QNX Neutrino powstał w 1996r.
- Początkowo na procesory 8088 (QNX), dzisiaj wersja QNX Neutrino 7, wspierająca systemy 32/64-bitowe i ARM, PowerPC, XScale oraz normę POSIX.

QNX Neutrino - podstawowe założenia

- System składa się z mikrojądra (tu $\approx$ 500kB, ale są mniejsze) oraz procesów. Mikrojądro odpowiada za:
  - wywłołania POSIX (semafory, mutex'y, ...),
  - usługę przekazywania wiadomości (w tym przez sieć),
  - planowanie procesów,
  - przekierowywanie przerwań HW.
- Procesy dostarczają każdej innej usługi w systemie (w tym są sterownikami, serwerami usług, etc.)
- Procesy kokmunikują się dzięki mechanizmom IPC (model klient/serwer).

QNX Neutrino - architektura



Jak widać, system ma budowę modułową, w odróżnieniu od *monolitycznego jądra* pracującego np. w LinuxRT. Wszelkie usługi (o znaczeniu dla użytkownika) są dostarczane przez *niezależne* wątki, komunikujące się za pomocą magistrali programowej.

To właśnie zarządzaniem tą magistralą zajmuje się mikrojądro, jako jednym ze swoich głównych zadań. Taka architektura jest dobrym przykładem *rozproszonego* systemu operacyjnego: faktycznie, mikrojądra działające na różnych procesorach mogą komunikować się między sobą tak samo, jak procesy z mikrojądrem na jednym procesorze.

QNX - procesy i wątki

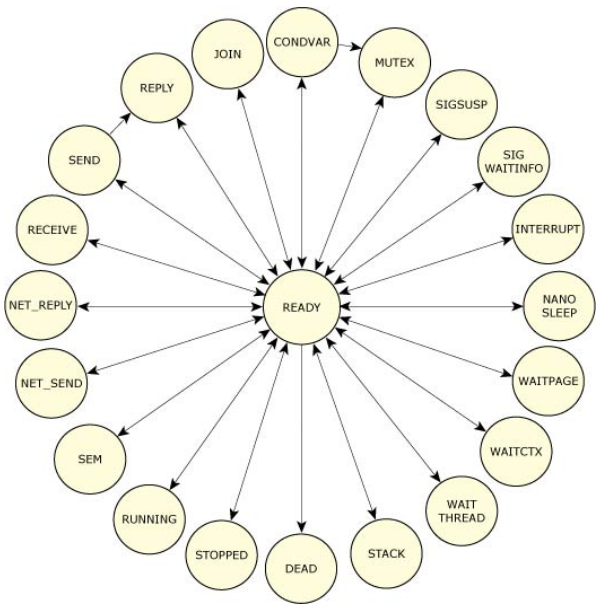
- Wątek jest podstawową „jednostką” obliczeniową/planowania w QNX.
- Proces jest zbiorem wątków, określającym ich wspólną przestrzeń adresową.
- Każdy wątek posiada swój kontekst: IP, SP, rejestry; ma swoją maskę sygnałów; są one przeładowywane w trakcie wywłaszczenia.

To znaczy, że:

- Przełączanie wątków w tym samym procesie jest b.szybkie! ♠ (dlaczego?)
- Przełączanie wątków w innych procesach jest szybkie, bo załatwia to zoptymalizowane jądro.
- Procesy działają niezaleznie od siebie, awaria jednego nie powoduje efektów lawinowych.
- Poprawność działania wątku/procesu można *analizować w izolacji* od innych wątków/procesów.

QNX – stany wątków

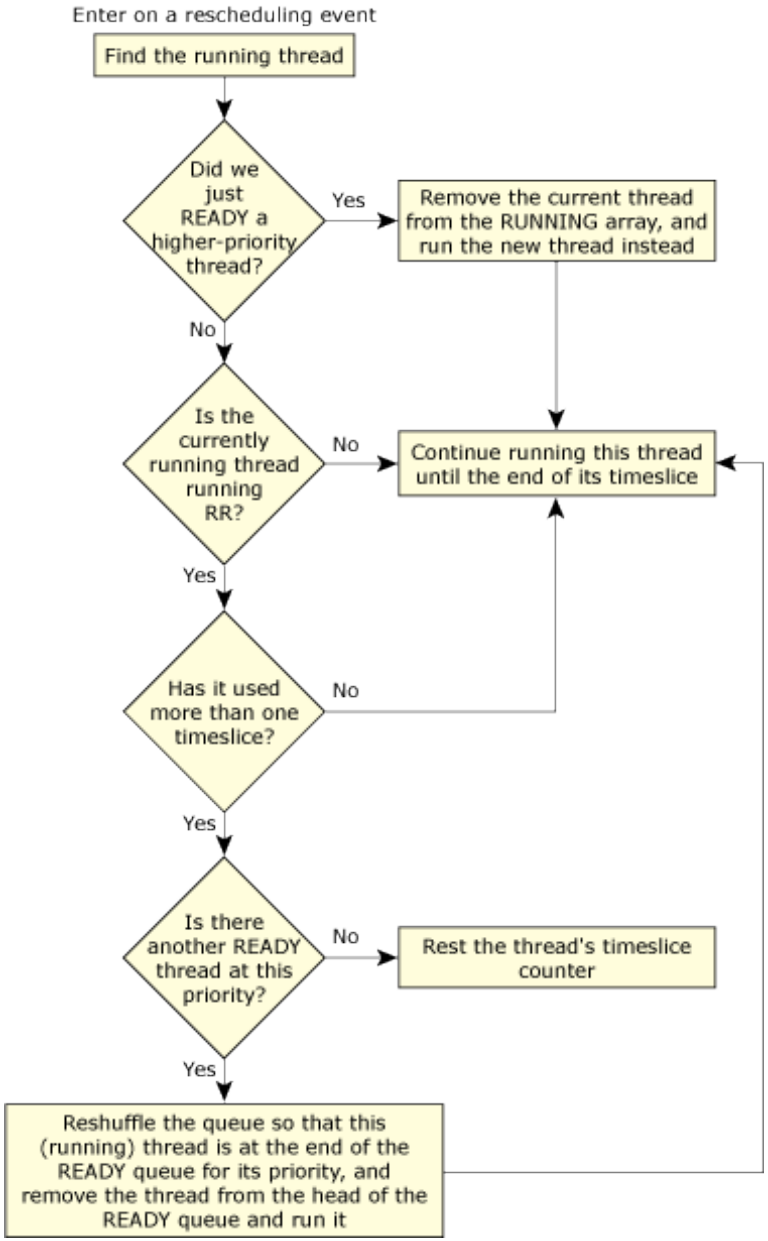
Wątek może być w jednym z nastu stanów, w zależności od tego, w którym miejscu wykonywania się znajduje. Ogólnie wątek może być wykonywany, może być gotowy do wykonania oraz może być zablokowany (czekać na coś, co nie zależy od jego samego).



READY, RUNNING, STOPPED – gotowy do wykonania/pracuje/skończył
SEND, REPLY – (klient) wysłał wiadomość do serwera i czeka na jej odbiór/na odpowiedź
RECEIVE – (serwer) oczekuje na wiadomość od klienta
CONDVAR, MUTEX, SEM – oczekuje na sygnał na zmiennej warunkowej/na mutex/na semafor
...

Mikrojądro zbiera informacje o stanach wątków, utrzymuje kolejkę wątków READY i realizuje algorytm planowania/uruchamia jednego z nich.

QNX – przełączanie wątków



QNX zawsze zezwala na wykonanie wątkowi w stanie READY o najwyższym priorytecie.
Jednak często w systemie działa więcej wątków o tym samym priorytecie...
Dodatkowo wykonanie wątku jest zawieszone na czas gdy jądro wykonuje *kernel call* (np. obsługę pamięci), obsługuje wyjątek lub przerwanie sprzętowe.

QNX – powody zmiany działającego wątku

Aktualnie wykonywany wątek zostanie wstrzymany a kontekst wymieniony, gdy wątek:

- zostanie wywłaszczony: gdy inny wątek o wyższym priorytecie staje się READY. Jest to zdarzenie trudne do zaprognozowania, zależy od liczby wątków działających w systemie, ich szybkości działania, rozkładu priorytetów... Określenie momentu ani czasu trwania wywłaszczenia nie jest możliwe.
- zostanie zablokowany: zacznie oczekiwać na sygnał, mutex, wywołanie IPC, itd. To zdarzenie jest nieco łatwiejsze do przewidzenia: znane dokładne miejsca w kodzie, gdzie następuje odwołanie „na zewnątrz" wątku (np. `send()`). Można określić *gdzie* nastąpi zablokowanie, trudniej określić *na jak długo*.
- ustąpi (*yield*): gdy dobrowolnie oddaje procesor (wywołanie `sched_yield()`). Jest to celowe działanie programisty – wątek oddaje procesor wtedy, kiedy w żaden sposób nie może z niego teraz skorzystać (♠ pomyśl, jakie to mogą być sytuacje).

QNX – planowanie wątków

W przypadku, gdy kilka wątków o tym samym (najwyższym) priorytecie jest READY, planista jądra musi podjąć decyzję o wykonaniu któregoś z nich.
Wątki mogą określać, jaki algorytm w takim przypadku ma być wykonany.

- FIFO – wątek jest wykonywany, dopóki:
  - traci kontrolę (jest zablokowany, j.w.)
  - jest wywłaszczony przez proces z wyższym priorytetem
- RR (round-robin) – rozszerzenie wersji FIFO:
  - każdemu wątkowi programista może przydzielić określony kwant czasu, przez który wątek ma się wykonywać;
  - znając liczbę wątków i przydzielone im czasy można ustalić, jak często dany wątek ma się wykonywać.
- planowanie sporadyczne

QNX - planowanie sporadyczne

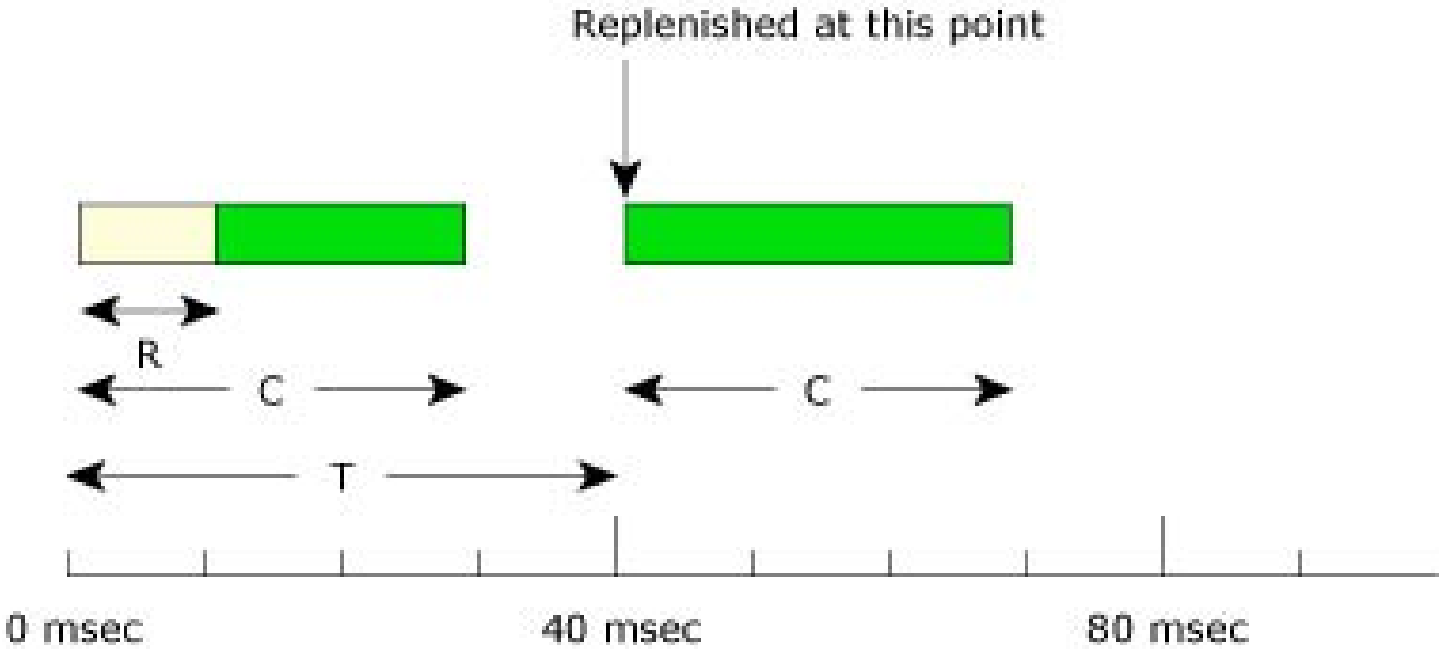
Planowanie sporadyczne

Narzuca limit czasu, przez który dany wątek będzie wykonywany w danym okresie. Umożliwia obsługę zdarzeń aperiodycznych i dotrzymanie czasów ukończenia pozostałych wątków.

Wątek wykonuje się, dopóki nie zablokuje się lub nie zostanie wywłaszczony.
Dodatkowo jego priorytet po pewnym czasie jest zmniejszany na pewien okres.

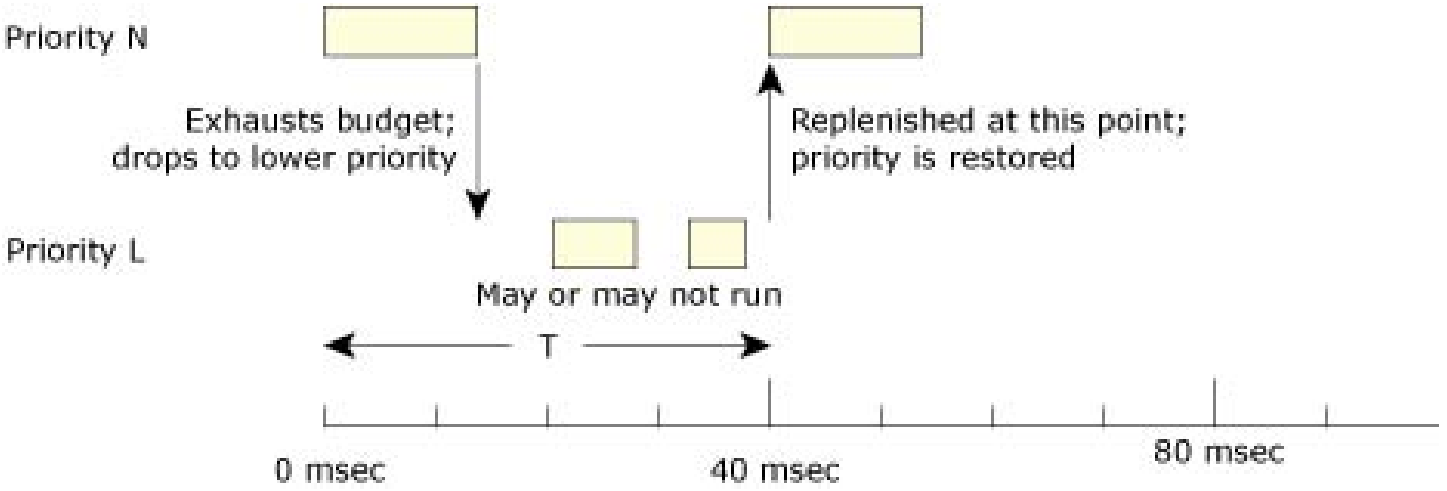
Daje to w efekcie ograniczenie górne na czas wykonywania się wątku w danym przedziale czasu. Jest to przydatne np. dla planowania *rate monotonic (RM)* z poprzedniego wykładu, gdy w systemie pojawiają się zadania periodyczne i aperiodyczne: gwarantuje, że żadne z takich zadań nie „przejmie" procesora, uniemożliwiając działanie innych wątków.

QNX - planowanie sporadyczne



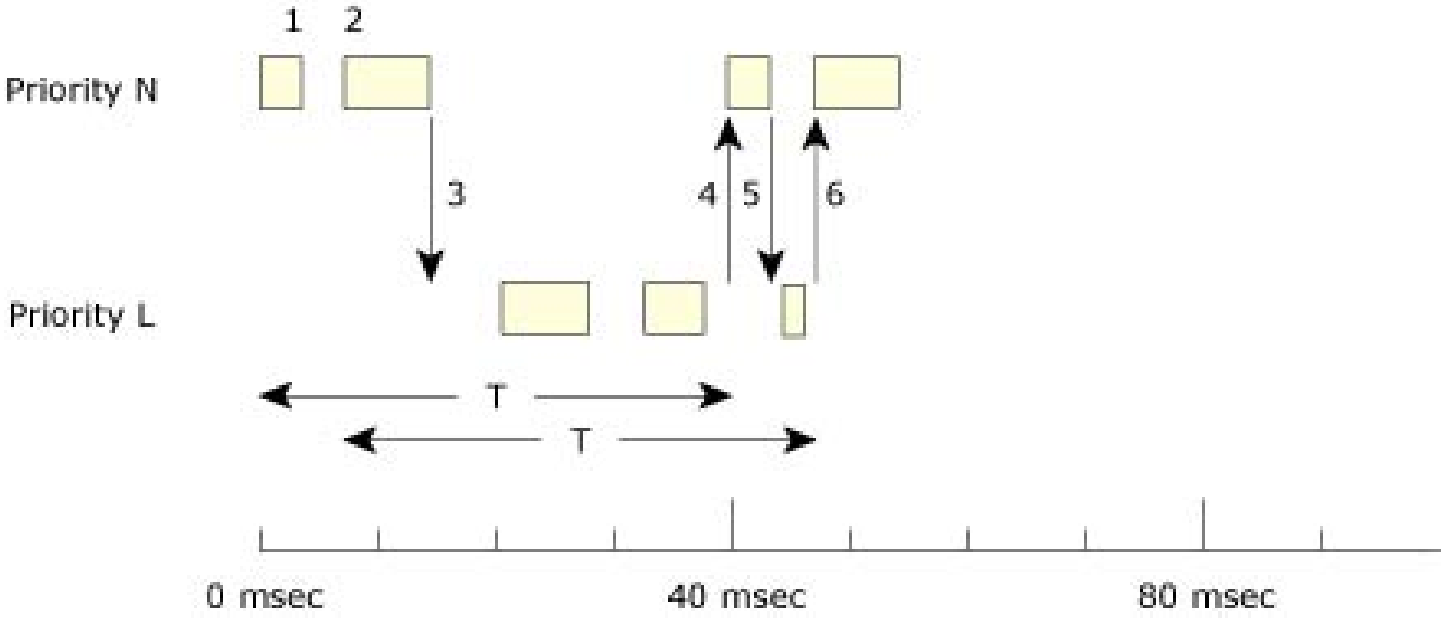
Wątek wykonuje się z pewnym budżetem czasu C, w którym jego priorytet jest stały. Uruchomienie wątku przez R czasu uszczupla jego budżet. Co określony czas T budżet jest odnawiany. Jeśli wątek został wywłaszczony, zanim wykorzystał swój budżet, zostanie on odnowiony po czasie T.

QNX - planowanie sporadyczne



Jeśli wątek w ciągu ostatniego okresu T pracował przez czas dłuższy niż C (wyczerpał budżet), to jego priorytet spada z N do L. Do następnego odświeżenia budżetu działa z niższym priorytetem, podlegając wywłaszczeniom na rzecz innych procesów o wyższych priorytetach. Z nadejściem kolejnego okresu T jego priorytet wraca do N i wątek kontynuuje „wydawanie budżetu" z wyższym priorytetem.

QNX - planowanie sporadyczne



Jeśli w trakcie wykonania wątek był wywłaszczany wielokrotnie, to przy każdym wywłaszczeniu startowany jest licznik odliczający czas T, po którego upłynięciu przywracany jest do budżetu czas wykorzytany do momentu wywłaszczenia.

Zadanie startuje w czasie (0). W momencie (1) następuje wywłaszczenie i od tego momentu po czasie T (chwila (4)) następuje odnowienie budżetu o czas między (0) a (1). Podobnie wywłaszczenie w (3) powoduje odnowienie w (6) o czas między (2) i (3).

Pathfinder

Wróćmy na Marsa... W 1996 r. sonda Pathfinder, zaraz po wylądowaniu na Czerwonej Planecie – zaczęła się resetować. W skrócie, operacja sondy polegała na zbieraniu informacji (zadanie *I* z niskim priorytetem), wysyłaniu ich na Ziemię (średni priorytet, długi czas trwania, zadanie *W*) i organizowaniu komunikacji w magistrali – zadanie *O* (podobnie jak w QNX, operujący w sondzie VxWorks organizował komunikację wewnętrznie za pomocą takiej „software bus") z wysokim priorytetem. Dostęp do magistrali (zasób krytyczny), był realizowany za pomocą mutex'ów. Problem pojawiał przy następującej kolejności zdarzeń:

1. *I* blokuje mutex, bo chce zebrać i przesłać dane meteo;
2. *W* wywłaszcza *I*, bo ma wyższy priorytet (mutex zablokowany!);
3. *O* wywłaszcza *W* i próbuje dostać się do magistrali... zostaje zablokowane w oczekiwaniu na zwolnienie mutexa przez *I*;
4. powraca zadanie *W* (aktualnie READY z najwyższym priorytetem) i *wykonuje się do końca*;
5. powraca zadanie *I* (*O* dalej jest zablokowane na mutexie) i *wykonuje się do końca*;
6. powraca zadanie *O*.

Pathfinder (2)

Jak widać zadanie *W* wykonało się przed *O*, mimo że w momencie pojawienia się *W* to *O* było READY.
Dodatkowo zadanie *I* musiało czekać na ukończenie pracy dłużej, niż to przewidzieli programiści (wpychało się w to zadanie *W*). Wewnętrzny system (Watchdog ♠) wyłapywał to i resetował system. Choroba została nazwana i mogła zostać naprawiona.

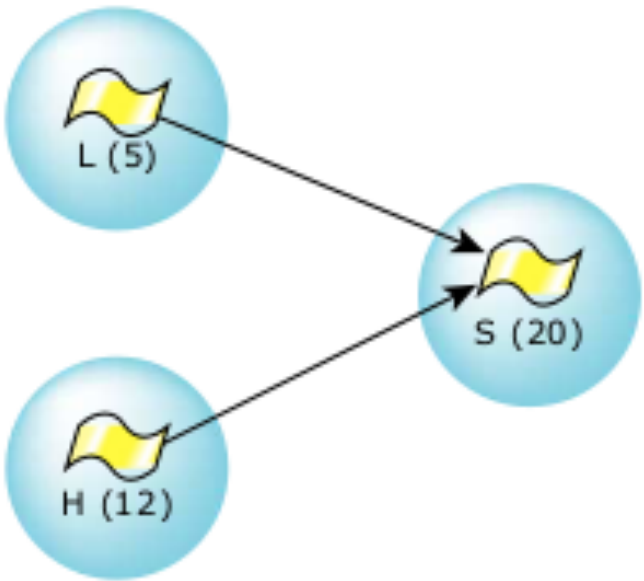
Powyższe zdarzenie nazywa się *inwersją priorytetów* i omówię je na następnych slajdach. ♠ Przeczytajcie T.Durkin: *The Vx-Files: What the Media Couldn't Tell You About Mars Pathfinder* oraz `https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft`.
Inną ciekawą rzeczą dotyczącą systemów tego typu jest następująca sprawa:

- w kodzie opracowywanym w JPL było dużo kodu debugowego;
- oczywiście kod był testowany i weryfikowany;
- przed startem **nie** usunięto kodu debugowego, bo *kod bez sekcji debugowych nie był testowany* – bo jak?
- na Ziemi została idealna kopia sondy (patrz Apollo 13...).

Odwracanie priorytetów ♠

Pomimo że RTOS mają uruchamiać proces READY z najwyższym priorytetem, czasami proces o niższym priorytecie zajmuje czas CPU.

Przypadek 1. Nie kombinujemy z priorytetami



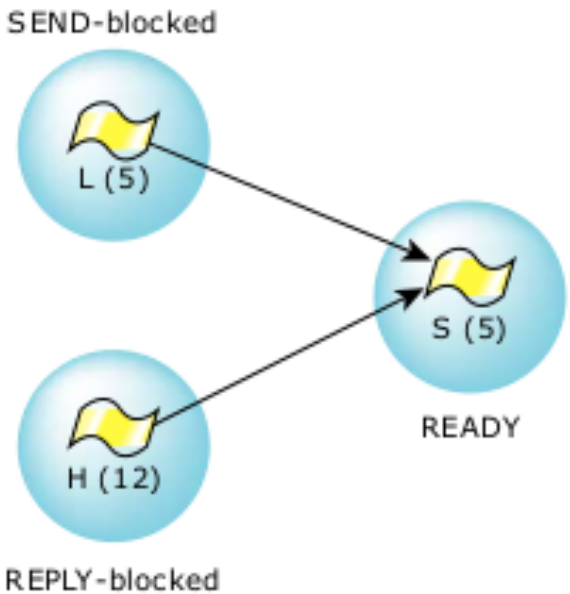
S - serwer (priorytet 20),
H - zadanie z wysokim priorytetem (12),
L - zadanie z niskim priorytetem (5)

1. H jest wykonywany, S oczekuje (RECEIVE);
2. H wykonuje `sleep(100)`, L jest wykonywany;
3. L "budzi" S, żądając długiego obliczenia (>100ms)
4. S staje się READY na czas obsługi żądania L;
5. po 100ms H jest READY, ale to S jest wykonywany, bo ma wyższy priorytet.

wykonanie L spowodowało że H nie jest wykonywany mimo że jest READY

Odwracanie priorytetów ♠

Przypadek 2.
Dziedziczenie priorytetów



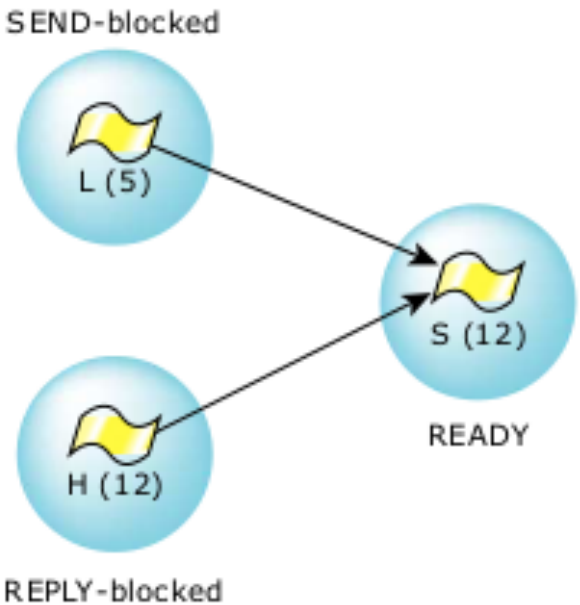
S dziedziczy priorytet po L

1. H jest wykonywany, S oczekuje (RECEIVE);
2. H wykonuje `sleep(100)`, L jest wykonywany z priorytetem 5;
3. L "budzi" S, żądając długiego obliczenia (>100ms)
4. S staje się READY, na czas obsługi żądania L otrzymuje priorytet 5
5. po 100ms H jest READY i żąda obliczeń od S, ale S ma priorytet 5 (niski) i inne zadania ograniczają jego czas na obsługę żądania L.

L spowodowało że H musi czekać na dostęp do zasobu (S)

Odwracanie priorytetów ♠

Przypadek 3.
Przewartościowanie priorytetów



S otrzymuje najwyższy priorytet spośród zablokowanych klientów

- QNX dokonuje przewartościowania priorytetów automagicznie
- po zakończeniu obsługi, priorytet S jest przywracany (lub nie, jeśli S przechodzi do stanu RECEIVE);

Działa? ♠