1. **Co to jest zdarzenie w standardzie POSIX i jakie możliwości obsługi zdarzenia przewiduje standard.**

Zdarzenie to rozszerzenie pojęcia sygnału. Jest to mechanizm do powiadamiania procesów przez system operacyjny o zdarzeniach np. błędach (nienormalnych zdarzeniach które wydarzyły się podczas wykonywania procesu). W takiej sytuacji system operacyjny dostarcza sygnał, który przekazuje informację o tym, co się zdarzyło i sygnalizuje potrzebę jego obsłużenia. Pomysł zdarzenia jest taki, ze powiadomienie o nim, nie jest jedyną możliwoscią. W przypadku nienormalnej sytuacji proces1 mógłby zrobić coś więcej niż tylko powiadomić o tym proces2. Lub np. jeśli dopuścimy wielowątkowość procesów, możemy odpalić nowy wątek i przekazać mu obsługę tego zdarzenia.

Mechanizm zdarzeń tj. supersygnałów, których obsługa może być złożona. Może polegać na dostarczeniu tradycyjnego sygnału, ale także odpaleniu wątku w procesie. Mamy 3 rodzaje sytuacji:

- nic nie zrobimy

- wyślemy sygnał

- przekażemy obsługę zdarzenia do nowego wątku.

Zdarzenie opisuje struktura sigevent (wykład 8)

Zdarzenie jest to rozszerzenie pojęcia sygnału (supersygnał). Mechanizm do powiadamiania procesów przez sytem operacyjny o zdarzeniach np. błędach. System operacyjny dostarcza sygnał (informację o tym co się zdarzyło). Trzy reakcje na zdarzenia 1. Nic nie robimy 2. Wysyłamy sygnał 3. odpalenie wątku obslugującego zdarzenie.

1. **Wyjaśnić cel, mechanizm działania i ograniczenia protokołu dziedziczenia priorytetu.**

**=**Wyjaśnić cel, mechanizm działania i ograniczenia protokołu pułapu priorytetu.

1. **Napisać szkic programu, który dokładnie co sekundę próbuje odczytać wiadomości z dwóch różnych kolejek.**

timer.it\_value.tv\_sec = 0;

timer.it\_value.tv\_nsec = 0;

timer.it\_interval.tv\_sec = 1;

timer.it\_interval.tv\_nsec = 0;

timer\_settime(timer\_id, 0, time, NULL);

int queue = 0;

do {

pause();

if(queue == 0)

{

readFromQueue(0);

queue = 1;

}

Else

{

readfromQueue(1);

queue = 0;

}

} while(run);

1. **Usługi warstwy 2 Profibus**

- SDN: wysłanie danych bez potwierdzenia (m – m)

- SDA: wysłanie danych z potwierdzeniem (m – m)

- SRD: wysłanie danych i żądanie danych (m – s)

- CSRD: cykliczne wysyłanie i żądanie danych (m – s)

port odpytywania

lista odpytywania

1. **Szkielet programu, który tworzy trzy procesy potomne (zapisane w plikach pp1, pp2, pp3) i oczekuje na zakończenie pracy jednego z nich.**

*pid fork()* – tworzy nowy proces i tablicę stanu – identyczną kopię procesu macierzystego – dziedziczy przestrzeń adresową, wskaźniki, otwarte pliki i semafory

*pid* – id procesu potomnego || 0 gdy jesteśmy w procesie potomnym

*int exec() –* zmiana obszaru danych i obszaru programu

*posix\_spawn()* – tworyz nowz proces, dla którego definiujemy argumentami dziedziczenie

int status;

pid = fork();

if(pid !=0) { … }

else {

exec(pp1,…);

}

pid = fork();

if(pid !=0) { … }

else {

exec(pp2,…);

}

pid = fork();

if(pid !=0) { }

else {

exec(pp3,…);

}

wait(&status);

**lub**

void func()  
{  
 posix\_spawn(&pid1, „pp1”, NULL, NULL, args1, env1);  
 posix\_spawn(&pid2, „pp2”, NULL, NULL, args2, env3);  
 posix\_spawn(&pid3, „pp3”, NULL, NULL, args3, env2);  
 waitpid(-1, &status, 0);  
}

1. **Szkielet programu, który tworzy trzy procesy potomne (zapisane w plikach pp1, pp2, pp3) i oczekuje na zakończenie pracy tego drugiego**

Void func()

{

Posix\_spawn(&pid1, „pp1”, NULL, NULL, argv1, env1);

Posix\_spawn(&pid2, “pp2”, NULL, NULL, argv2, env2);

Pthread\_wait(pid2, &status, 0);

}

1. **Sposób adresowania wiadomości w sieci CAN.**

Poszczególne węzły nie mają adresu, jest to sieć magistralowa o topologii rozgłoszeniowej. Nie ma adresacji węzłów, są numerowane (adresowane) zmienne. Każda zmienna w tej sieci ma unikalne ID. Sieć jest bezpołączeniowa, wszystkie węzły rozgłaszają wiadomości broadcastem do innych węzłów. Po ID jest rozpoznawane czy nas to interesuje czy nie.

W sieci CAN nie na bezpośredniej adresacji wiadomości, ponieważ wykorzystywane jest rozgłaszanie komunikatów – każdy węzeł ma dostęp do wiadomości. Dlatego do adresowania wykorzystuje się pole Id. Zawiera ono informację o tym jakie dane są przesyłane. Każdy kontroler podpięty do sieci na podstawie informacji zawartych w polu Id decyduje o tym czy ta wiadomość jest mu potrzebna czy nie.

1. **Szkielet programu monitora realizującego stos liczb całkowitych w systemie zgodnym ze standardem POSIX.**

void push(int num)

{

mutex\_lock(mutex);

buf[count] = num;

++count;

if(count == 1)

cond\_signal(cond);

mutex\_unlock(mutex);

}

void pop(int \*num)

{

Mutex\_lock(mutex);

If(count == 0)

Cond\_wait(cond, mutex);

--count;

\*num = buf[count];

Mutex\_unlock(mutex);

}

1. **Szkielet programu procesu, który tworzy semafor nienazwany i wykorzystuje go do synchronizacji dostępu do zasobu dzielonego ze swoim procesem potomnym.**

fd=shm\_open(”semafor”,O\_RDWR|O\_CREAT,0777);

size=ltrunc(fd,sizeof(sem\_t),SEEK\_SET);

addr=mmap(0,sizeof(sem\_t),PROT\_READ|PROT\_WRITE,

MAP\_SHARED,fd,0);

sem=(sem\_t \*)addr;

sem\_init(sem,1,0); // pshared

...

fork();

...

sem\_wait(sem);

...

sem\_destroy(sem);

munmap(addr,sizeof(sem\_t);

close(fd);

shm\_unlink(”semafor”);

...

/\* drugi proces – bez wykorzystania fork() \*/

...

fd=shm\_open(”semafor”,O\_RDWR);

addr=mmap(0,sizeof(sem\_t),PROT\_READ|PROT\_WRITE,

MAP\_SHARED,fd,0);

sem=(sem\_t \*)addr;

...

sem\_post(sem)

...

munmap(addr,sizeof(sem\_t);

close(fd);

1. Zdania charakteryzujące sieci przemysłowe
   * Muszą efektywnie przenosić krótkie wiadomości
   * Muszą efektywnie przenosić długie wiadomości
   * Sieć CAN wdraża priorytetowe szeregowanie wszystkich wiadomości nadawanych w sieci
   * Sieć Interbus jest siecią typu master-slave z wieloma węzłami typu master

* Wykonanie pomiaru i odczytanie przez węzeł nadrzędny temperatury z szybkiego czujnika dołączonego do węzła podporządkowanego wymaga jednego cyklu odpytania
* Sieć Interbus gwarantuje przekazanie wartości wszystkich zmiennych procesowych w pewnym z góry znanym czasie
* Węzły typu slave sieci Profibus nie mogą wywoływać żadnych usług warstwy liniowej
* W warstwie liniowej sieć profibus używa algorytmu dostępu z odpytywaniem
* W warstwie liniowej sieć profibus używa algorytmu dostępu znacznikowego
* Algorytm dostępu z przekazywaniem znazcnika gwarantuje deterministyczny czas nadania wszystkich komunikatów
* Długość okresu, przez który węzeł typu *master* sieci Profibus może nadawać swoje komunikaty jest ograniczona od góry przez wartość ustaloną podczas konfiguracji sieci.
* Długość okresu, przez który węzeł typu *master* sieci Profibus może nadawać swoje komunikaty jest stałą ustaloną podczas konfiguracji sieci.
* Każde wywołanie usługi Read w relacji **MSAC** sieci Profibus powoduje wysłanie żądania do węzła serwera i odebranie jego odpowiedzi
* Każde wywołanie usługi Read w relacji **MSCY** sieci Profibus powoduje wysłanie żądania do węzła serwera i odebranie jego odpowiedzi
* Każde wywołanie usługi Write w relacji **MSAC** sieci Profibus powoduje wysłanie danych do węzła serwera i odebranie jego potwierdzenia
* Warstwa aplikacyjna sieci Profibus realizuje bezpołączeniowy protokół komunikacyjny
* Warstwa aplikacyjna sieci Profibus realizuje model komunikacji klient-serwer, w którym rolę serwera pełni węzeł typu **master**.
* Warstwa aplikacyjna sieci Profibus realizuje model komunikacji klient-serwer, w którym rolę serwera pełni węzeł typu **slave**.

1. **Napisać szkielet programu cyklicznego o dokładnie określonym czasie cyklu**

Int timer\_create(struct sigevent \*evp, timer\_t\* timer)

Int timer\_delete(timer\_t\* timer)

Int timer\_settime(timer\_t \*timer, struct itimerspec \*value)

Int timer\_getoverrun(timer\_t \*timer)

Timer\_t licznik;

Struct itimerspec menspec;

Struct timespec interval, value;

Value.tv\_sec = value.tv\_nsec = 0;

Interval.tv\_sec = 1;

Interval.tv\_nsec = 0;

Menspec.it\_interval = interval;

Menspec.it\_value = value;

Timer\_create(&event,&licznik);

Timer\_settime(licznik, &menspec);

While() {

//cos…

Pause();

}

1. **Zaprojektować drajwer wielokanałowego przetwornika a/c w postaci biblioteki funkcji działającego w systemie zgodnym ze standardem POSIX. Drajwer nie powinien dopuszczać do inwersji priorytetów. (W9)**
   * **Lista funkcji**
   * **Narysować szkielety programów**

ad a) ac\_open();

ac\_read(char chan, sem\_t sem);

ac\_close(sem\_t sem)

ad b)

sem\_t ac\_open()

{

sem\_t sem;

sem\_init(&sem, NULL, 1);

return sem;

}

int ac\_read(char chan, sem\_t sem)

{

Uint8\_t stan, ah, al;

Sem\_wait(sem);

out8(BASE+1, chan); // ustawienie kanału

out8(BASE, 0); // rozpoczęcie pomiaru

do {

stan = in8(BASE);

} while(stan == 0)

ah = in8(BASE+2);

al = in8(BASE+3);

return (ah << 8) + al;

}

void ac\_close(sem\_t sem) { sem\_destroy(&sem); } **Wykład 9.**

1. **Wyjaśnić cel, mechanizm działania i ograniczenia protokołu pułapu priorytetu.**Protokół pułapu priorytetu służy do minimalizacji (lub wręcz unikniecia) zjawiska inwersji priorytetów oraz zakleszczeń.

Wszystkie procesy, które ubiegają się o dany zasów otrzymują maxymalny priorytet przypisany do danego zasobu wraz z pułapem priorytetu. Jego wartość jest określana przez najwyższy priorytet z pośród zadań, które mogą o niego konkurować. Protokół jest przejrzysty i przewidywalny, posiada jednak pewne ograniczenia. Jeżeli dany proces ma długą sekcję krytyczną, może znacznie opóźnić wykonywanie procesów konkurujących o zasób, co z kolei może negatywnie wpłynąć na ograniczenia czasowe systemu. Należy to wziąć pod uwagę podczas projektowania.

Protokół pułapu priorytetów jest zastosowany w celu likwidacji zjawiska inwersji priorytetów. Polega to na tym, że jeżeli proces o stosunkowo niskim priorytecie zajmie jakiś zasób, na który będzie oczekiwal proces o wyższym priorytecie, to temu procesowi o niższym priorytecie przydzielany jest tymczasowo taki sam priorytet jaki ma zadanie zawieszone na zasobie. Innymi słowy – priorytet zadania zajmującego zasób jest określana przez najwyższy priorytet spośród zadań, które mogą o ten zasób konkurować. Po zwolnieniu zasobu priorytet wraca do swojej poprzedniej wartości. Ograniczeniem tego protokołu jest to, że jeżeli zadanie o niskim priorytecie zajmie zasób na bardzo długi okres czasu, to uniemożliwi wykonanie zadań o wyższych priorytetach oczekujących również na ten zasób.

1. **Wyjaśnić działanie protokołu z odpytywaniem.**

Jego cechą jest oddzielenie rytmu przekazywania od rytmu komunikacji. Master odpytuje slave, a on mu natychmiast odpowiada. Jak policzy, to slave składa odpowiedź do bufora wyjściowego i wysyła dopiero jak master go zapyta. Nie ma niebezpieczeństwa kolizji, bo tylko master wysyła żądania. Slave nic nie robi sam z siebie.

W protokole z odpytywaniem węzły dzielą się na dwie kategorie – Master i Slave. Węzły master odpytują, natomiast węzły slave tylko odpowiadają. Ponadto węzly slave posiadają dwa bufory – do wysyłania i do odczytu. W momencie gdy węzeł typu Master wyśle wiadomość do slave, ten zapisuje wiadomość w buforze odczytu. Węzeł slave odczytuje tą wiadomość, wykonuje zlecone zadanie, a wynik umieszcza w buforze wysyłania. Przy następnej turze odpytywania, gdy Master odpyta dany węzeł Slave, ten wyśle mu dane zgromadzone w buforze wysyłania. I tak w kółko **danielu**.

1. **Wyjaśnić działanie protokołu z przekazywaniem znacznika. (W10)**

Protokół ten jest niesprzeczny z protokołem odpytywania. Nie może jednak przekazywać długich wiadomości. Jest w nim TOKEN (znacznik), który wędruje po kolejnych węzłach sieci i idzie w ustalonym porządku. Jak ktoś ma znacznik to może wykonywać operacje, a pozostali czekają, dzięki czemu unikamy kolizji. Węzły równoprawne, występuje jednak narzut związany z przekazywaniem znacznika. Do wad należy możliwość utraty znacznika lub jego rozmnożenie.

Mamy sobie węzły podpięte do sieci, wszsytkie równouprawnione. Żeby uniknąć kolizji zakłada się, że nadawać komunikaty może tylko ten węzeł, który w danej chwili posiada znacznik. Znacznik jest przekazywany pomiędzy kolejnymi węzłami co określoną chwilę czasu, dzięki czemu protokół jest deterministyczny.

1. **Do czego służą funkcje mlock i mlockall standardu POSIX i w jakich okolicznościach ich użycie jest konieczne? (wykład 8)**

*Proces rezydentny… ?*

Mlock() – dla fragmentu kodu

Wymiatanie – swapping, przesyłanie procesu pomiędzy pamięcią operacyjną i dyskową

Int mlockall(int flags) – wyłączenie pamięci wirtualnej w stosunku do całej pamięci adresowej procesu.

Flagi- FUTURE/CURRENT

Wadą jest to, że wdraża niepewność czy ten fragment jest w pamięci. Dostępność funkcji jest niedeterministyczna czasowo, ponieważ np. przerwanie musi być wykonane od razu.

Funkcje mlock i mlockall gwarantują nam, że pewien fragment pamięci (mlock) lub cała pamięć (mlockall) zostaje zatrzymania w RAM-ie i nie będzie polegała wymianie. W przypadku mlockall() – funkcja wyłącza mechanizm pamięci wirtualnej. Stosuje się to między innymi, żeby uzyskać determinizm czasu działania procesu. Przykładowa sytuacja: mamy proces, który ma obsługę przerwania. Proces bardzo długo jest nieaktywny i zostaje zmieciony na dysk. Nagle nadejdzie przerwanie, system będzie chciał się odwołać do funkcji obsługi przerwania przez wektor adresów przerwań, ale adres będzie błędny (bo funkcja jest na dysku, a nie w pamięci) ☹.