Lista 10

tags: SO

Zadanie 1

Zadanie 1. Czym różni się **przetwarzanie równoległe** (ang. *parallel*) o
(ang. *concurrent*)? Czym charakteryzują się **procedury wielobieżne** (
procedury w języku C (a) wielobieżnej, ale nie **wątkowo-bezpiecznej** (ang.
w jednowątkowym procesie uniksowym może wystąpić współbieżność?

Wskazówka: Właściwości *thread-safe* i *async-signal-safe* są opisane również w POSIX S

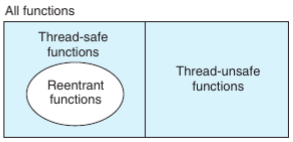
> A system is said to be *concurrent* if it can support two or more
> *progress* at the same time. A system is said to be *parallel* if it c
> two or more actions executing simultaneously.

::: info

- **wątkowo-bezpieczna** – procedura zachowująca się poprawnie w środowisku wielowątkowym
- **wielobieżna** – procedura, która może być bezpiecznie wywołana, gdy jej wcześniejsza instancja jest w trakcie wykonywania. :::

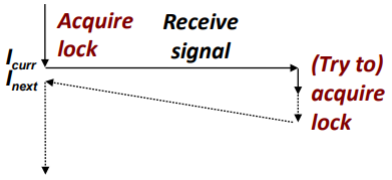
a) Procedura wielobieżna, ale nie wątkowo-bezpieczna

Figure 12.39
Relationships between the sets of reentrant, thread-safe, and thread-unsafe functions.



Nie istnieje? ツ

b) Procedura wątkowo-bezpieczna, ale nie wielobieżna



Ogólnie każda procedura korzystająca z blokad, np. `printf`

Kiedy w jednowątkowym procesie może wystąpić współbieżność?

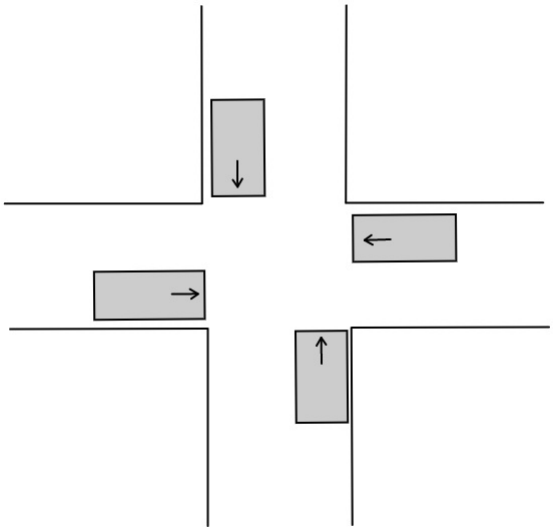
Przy obsłudze sygnału

Zadanie 2

Zadanie 2. Wybierz odpowiedni scenariusz zachowania wątków, w któryı
i na tej podstawie precyzyjnie opisz zjawisko **zakleszczenia** (ang. *dead*
oraz **głodzenia** (ang. *starvation*). Dalej rozważmy wyłącznie ruch uliczny
żowaniach może powstać każde z wymienionych zjawisk? Zaproponuj me
zakleszczeń (b) **zapobiegania** zakleszczeniom. Rozważ dwa warianty: w p
jącego ruchem policjanta, w drugim kierowcy mogą przekazywać sobie zn
zagłodzić pewnych kierowców?

- A deadlock is a situation that occurs in OS when any process enters in a waiting state because the demanded resource is being held by another waiting process.
- A livelock, on the other hand, is almost similar to a deadlock, except that the states of the processes which are involved in a livelock always keep on changing to one another, none progressing.

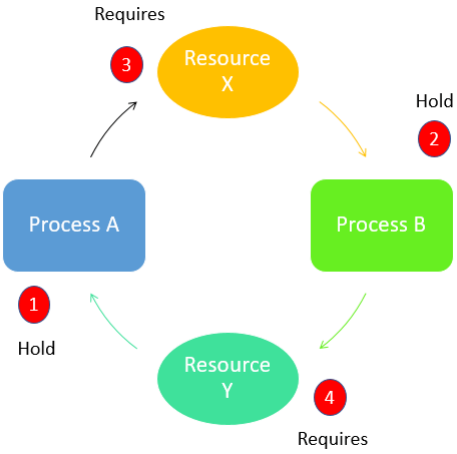
Deadlock

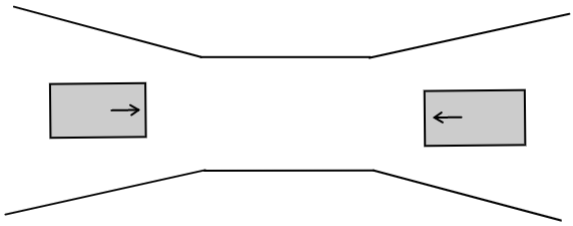


Livelock

6.7.3 Livelock

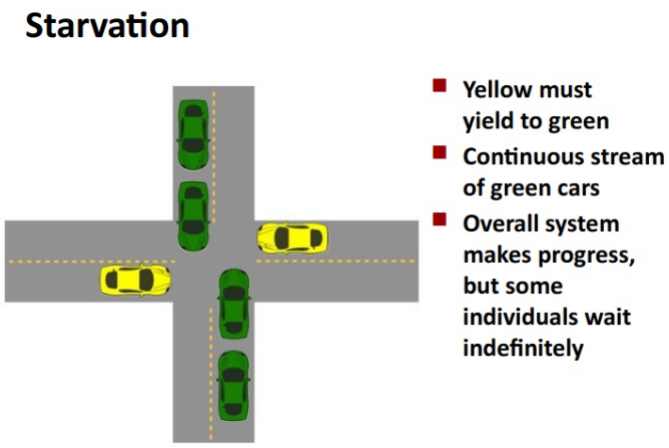
In some situations, a process tries to be polite by giving up the locks it already acquired whenever it notices that it cannot obtain the next lock it needs. Then it waits a millisecond, say, and tries again. In principle, this is good and should help to detect and avoid deadlock. However, if the other process does the same thing at exactly the same time, they will be in the situation of two people trying to pass each other on the street when both of them politely step aside, and yet no progress is possible, because they keep stepping the same way at the same time.





An easiest example of Livelock would be two cars who meet face-to-face in an alley, and both of them move aside to let the other pass. They end up moving from side to side without making any progress as they move the same way at the time. Here, they never cross each other.

Starvation



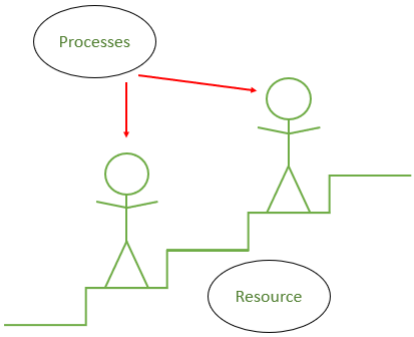
Wykrywanie i usuwanie zakleszczeń

TODO?

Możemy nadać zasobom limit czasowy, wtedy jeśli blokujący proces nie zdąży nabyć wszystkich potrzebnych mu zasobów to jest zmuszony odblokować obecnie trzymany zasób, który przypuszczalnie zostanie wtedy wykorzystany przez inny proces do wyjścia z deadlocka (kierowcy uzgadniają kolejność między sobą).

Można również próbować wykrywać i rozwiązywać cykliczność w grafie oczekiwania na zasoby (policjant nadzoruje i kieruje ruchem).

Zapobieganie zakleszczeniom



There are 4 conditions necessary for the occurrence of a deadlock. They can be understood with the help of the above illustrated example of staircase :

1. **Mutual Exclusion:**
   When two people meet in the landings, they can't just walk through because there is space only for one person. This condition to allow only one person (or process) to use the step between them (or the resource) is the first condition necessary for the occurrence of the deadlock.
2. **Hold and Wait:**
   When the 2 people refuses to retreat and hold their grounds, it is called holding. This is the next necessary condition for the the deadlock.
3. **No Preemption:**
   For resolving the deadlock one can simply cancel one of the processes for other to continue. But Operating System doesn't do so. It allocates the resources to the processors for as much time needed until the task is completed. Hence, there is no temporary reallocation of the resources. It is third condition for deadlock.
4. **Circular Wait:**
   When the two people refuses to retreat and wait for each other to retreat, so that they can complete their task, it is called circular wait. It is the last condition for the deadlock to occur.

Note:
All the 4 conditions are necessary for the deadlock to occur. If any one is prevented or resolved, the deadlock is resolved.

- Możemy zabronić wątkom rezerwowania więcej niż jednego zasobu.
- **Read only files** Shared resources such as read-only files do not lead to deadlocks. Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

- **Hold and Wait** To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this: Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later. Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it. Either of the methods described above can lead to starvation if a process requires one or more popular resources.
- **No Preemption** Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible. One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion. Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources and are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting. Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.
- **Circular Wait** One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order. In other words, in order to request resource Rj, a process must first release all Ri such that i >= j. One big challenge in this scheme is determining the relative ordering of the different resources

Zadanie 3

Zadanie 3. W poniższym programie występuje **sytuacja wyścigu** (ang. *r*
do współdzielonej zmiennej «tally». Wyznacz jej najmniejszą i największ

```
1 const int n = 50;
2 shared int tally = 0;
3
4 void total() {
5   for (int count = 1; count <= n; count++)
6     tally = tally + 1;
7 }
8
9 void main() { parbegin (total(), total()); }
```

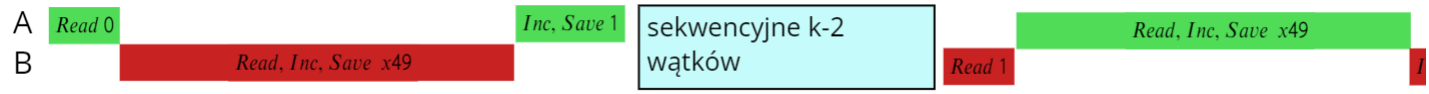
Dyrektywa «parbegin» rozpoczyna współbieżne wykonanie procesów. Ma
tyczne wyłącznie na rejestrach – tj. kompilator musi załadować wartość zm
wykonaniem dodawania. Jak zmieni się przedział możliwych wartości zmi
k procesów zamiast dwóch? Odpowiedź uzasadnij pokazując przeplot, któr

```
const int n = 50;
shared int tally = 0;

void total() {
    for (int count = 1; count <= n; count++)
        tally = tally + 1;
}

void main() { parbegin (total(), total()); }
```

Najmniejsza wartość

Jest to 2 i uzyskujemy ją następująco:

Można łatwo sprawdzić, że dla k >= 2 może być tak samo:



Metodą eliminacji dochodzimy, że jest to wartość minimalna:

- nie jest to 0, ponieważ to by oznaczało, że ostatni wątek nie wykonał dodawania
- nie jest to 1, ponieważ ostatni wątek musiałby wczytać 0 co jest niemożliwe, ponieważ co najmniej jeden wątek zakończył się przed nim i ustawił tam >=1

Największa wartość

Wystarczy, że wątki wykonają się sekwencyjnie, wtedy dla k wątków mamy wartość równą `k * n`

Zadanie 4

Zadanie 4. Podaj odpowiedniki funkcji `fork(2)`, `exit(3)`, `waitpid(2` wątków i opisz ich semantykę. Porównaj zachowanie wątków **złączalnyc** (ang. *detached*). Zauważ, że w systemie `Linux` procedura `pthread_c` reprezentacji wątku w przestrzeni użytkownika, w tym utworzenie stosu i wywołania `clone(2)`. Kto zatem odpowiada za usunięcie segmentu stosu z zakończy pracę? Pomocne może być zajrzenie do implementacji funkcji p

Porównanie

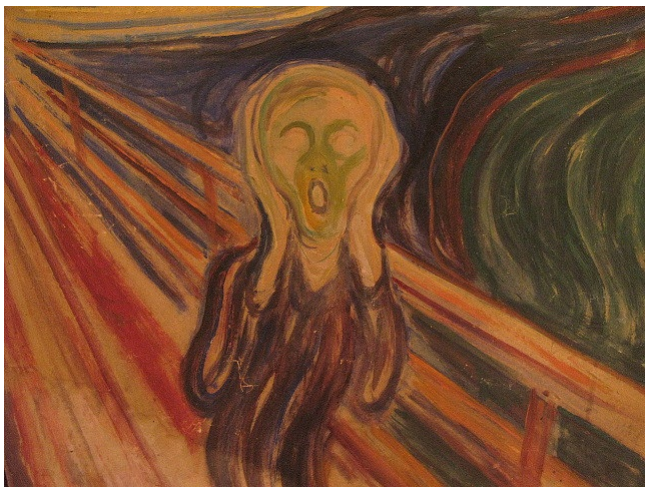
- odczepione -- odczepiony procedurą `pthread_detach` wątek jest wyczekiwany i grzebany przez system, nie mamy możliwości odczytania jego statusu wyjścia
- złączalne -- wątki, na których można wykonać `pthread_join`, wątek jest wtedy prawidłowo oczekiwany i grzebany.

Odpowiedniki

Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Figure 11.6 Comparison of process and thread primitives

Semantyka



::: spoiler Jest też https://lifecs.likai.org/2010/06/pthread-atexit.html, ale jego omówienie nie sprzyja obecnie moim interesom ツ :::

pthread_create

```
int pthread_create(
    pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void *),
    void *restrict arg);
```

::: warning

- **thread** *Before returning, a successful call to pthread_create() stores the ID of the new thread in the buffer pointed to by thread; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.*
- **attr** *The attr argument points to a pthread_attr_t structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using pthread_attr_init(3) and related functions. If attr is NULL, then the thread is created with default attributes.*
- **start_routine** *The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine()*
- **arg** *arg is passed as the sole argument of start_routine().* :::

pthread_exit

```
noreturn void pthread_exit(void *retval);
```

:::warning

- **reval** The pthread_exit() function terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls pthread_join(3). :::

pthread_join

```
 int pthread_join(pthread_t thread, void **retval);
```

::: warning

- **thread** *The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.*
- **retval** *If retval is not NULL, then pthread_join() copies the exit status of the target thread (i.e., the value that the target thread supplied to pthread_exit(3)) into the location pointed to by retval. If the target thread was canceled, then PTHREAD_CANCELED is placed in the location pointed to by retval.* ::: **pthread_cleanup_push**

```
void pthread_cleanup_push(void (*routine)(void *), void *arg);
```

::: warning *Used to push onto the calling thread's stack of thread- cancellation clean-up handlers. A clean-up handler is a function that is automatically executed when a thread is cancelled*

A cancellation clean-up handler is popped from the stack and executed in the following circumstances:

1. *When a thread is canceled, all of the stacked clean-up handlers are popped and executed in the reverse of the order in which they were pushed onto the stack.*
2. *When a thread terminates by calling pthread_exit(3), all clean-up handlers are executed as described in the preceding point. (Clean-up handlers are not called if the thread terminates by performing a return from the thread start function.)*
3. *When a thread calls pthread_cleanup_pop() with a nonzero execute argument, the top-most clean-up handler is popped and executed.*

it might be used, for example, to unlock a mutex so that it becomes available to other threads in the process.

- **routine** *The pthread_cleanup_push() function pushes routine onto the top of the stack of clean-up handlers.*
- **arg** *When routine is later invoked, it will be given arg as its argument. :::*

pthread_cancel

```
int pthread_cancel(pthread_t thread);
```

::: warning

- **thread** *The pthread_cancel() function sends a cancellation request to the thread thread. Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability state and type. :::*

Zadanie 5

Zadanie 5. Implementacja wątków POSIX skomplikowała semantykę niel omawialiśmy do tej pory. Co nieoczekiwanego może się wydarzyć w wielow

- jeden z wątków zawoła funkcję `fork(2)` lub `execve(2)` lub `exit_`
- proces zadeklarował procedurę obsługi sygnału «SIGINT», sterow «SIGINT» – w kontekście którego wątek zostanie obsłużony sygnał
- określono domyślną dyspozycję sygnału «SIGPIPE», a jeden z wątków której drugi koniec został zamknięty?
- czytamy w wielu wątkach ze pliku zwykłego korzystając z tego sam

Co nieoczekiwanego może się wydarzyć, gdy

- **jeden z wątków zawoła `fork`, `execve` lub `exit_group` ?**
  - fork -- dziecko składa się tylko z wątku wołającego (ale możemy zachować blokady na zasobach mimo, że ich wątków już nie ma)
  - execve -- wszystkie wątki poza wątkiem wołającym ulegają terminacji (tutaj grozi nam wyciek deskryptorów, należy posłużyć się flagą `FD_CLOEXEC`)
  - exit_group -- wychodzi we wszytkich wątkach (*`This system call is equivalent to _exit(2) except that it terminates not only the calling thread, but all threads in the calling process's thread group.`*)
- **proces zadeklarował procedurę obsługi SIGINT, sterownik terminala wysyła do procesu SIGINT -- w kontekście którego wątku zostanie obsłużony sygnał?** Sygnał kierowany do procesu trafia do arbitralnie wybranego wątku, który tego sygnału nie blokuje.
- **określono domyślną dysozycję sygnału SIGPIPE, a jeden z wątków próbuje pisać do rury, której drugi koniec został zamknięty?** Wątek otrzymuje SIGPIPE i według domyślnej obsługi tego sygnału jest terminowany. *`Signal dispositions are process-wide; all threads in a process share the same disposition for each signal. If one thread uses sigaction() to establish a handler for, say, SIGINT, then that handler may be invoked from any thread to which the SIGINT is delivered. A signal may be directed to either the process as a whole or to a specific thread. A signal is thread-directed if it is generated as the direct result of the execution of a specific hardware instruction within the context of the thread (SIGBUS, SIGFPE, SIGILL, and SIGSEGV)`*
- **czytamy w wielu wątkach z pliku zwykłego korzystając z tego samego fd?** *`If a file is opened by a threaded program (or a task which shares its file descriptors with another, more generally), the file pointer is also shared, so you need to use another method to access the file to avoid race conditions causing chaos - normally pread, pwrite, or the scatter/gather functions readv and writev.`*

Zadanie 6

Zadanie 6. Na podstawie [2, 14.4] opowiedz jaka motywacja stała za w [3, 63.2.2] do systemów uniksowych? Czemu lepiej konstruować oprog o odpytywanie deskryptorów albo powiadamianie o zdarzeniach bazując plikach można oczekiwać na zdarzenia przy pomocy `poll(2)` [3, 63.2.3]? kazywane do `poll` powinno skonfigurować się do pracy w trybie nieblokując `connect(2)`, `accept(2)`, `read(2)` i `write(2)` na gnieździe sieciowym zamiast blokować się w jądrze [7, 16]? Chcemy by po wywołaniu `poll(2` nionych wywołań systemowych nie zwróciła «EWOULDBLOCK». Jaka wart do pola «`revents`» struktury «`pollfd`» dla danego deskryptora pliku, że

Jaka motywacja stała za wprowadzeniem `poll`?

Początkowo chcieliśmy wykorzystać dwa procesy do czytania i pisania z dwóch deskryptorów, w ten sposób nie nastawialibyśmy się na ryzyko, że dane pojawią się w jednym deskryptorze, gdy realizujemy blokujące czytanie w drugim.

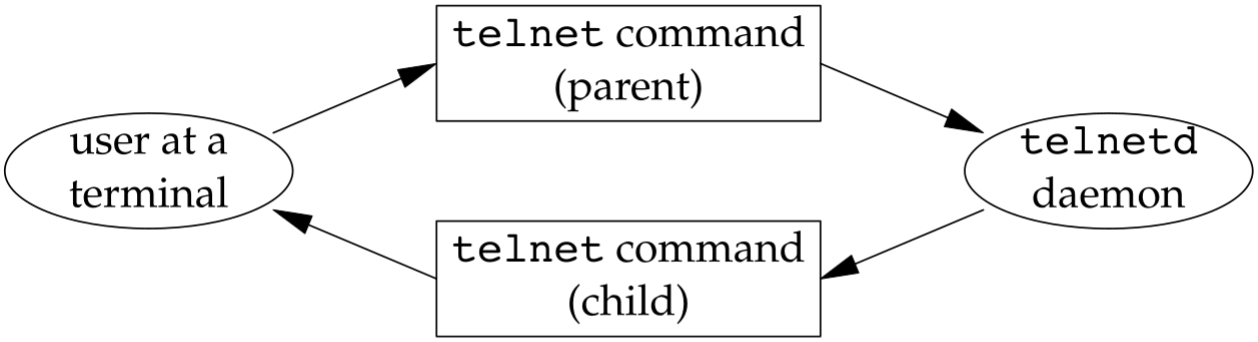


Figure 14.14 The `telnet` program using two processes

Pojawia się tutaj problem zakończenia tej komunikacji. Widzimy, że jeśli `telnetd` wyśle EOF to dziecko ulega terminacji, a ojciec otrzymuje SIGCHLD. Niestety sytuacja odwrotna jest kłopotliwa: jeśli EOF przyjdzie od strony terminala to nie mamy prostego sposobu by przekazać tę informację do dziecka (pewnym rozwiązaniem byłoby ustawienie zachowania dla SIGUSR1, ale zdaniem autora jest to mało eleganckie).

Tutaj pojawia się `poll` (i przyjaciele), który pozwala nam zaczekać, aż chociaż jeden deskryptor z podanej puli jest gotowy do obsłużenia.

Czemu lepiej konstruować oprogramowanie w oparciu o `poll`, niż odpytywanie deskryptorów albo powiadamianie o zdarzeniach bazujące na sygnale *SIGIO*?

Przy podejściu tzw. asynchronicznym polegamy na powiadomieniach systemu o gotowości do odczytu danego deskryptora. Problemem jest to, że sygnały niosą za mało informacji, by takie rozwiązania były wydajne, gdy mamy doczynienia z licznymi deskryptorami, tj. mamy za mało sygnałów by pokryć wszystkie deskryptory, czyli musielibyśmy przejrzeć je wszystkie za każdym razem, gdy otrzymujemy sygnał.

Na jakich plikach można oczekiwać na zdarzenia przy pomocy `poll`?

- Regular files
- Terminals and pseudoterminals
- Pipes and FIFOs
- Sockets

Czemu wszystkie deskryptory przekazywane do `poll` powinno skonfigurować się do pracy w trybie nieblokującym?

Celem korzystania z `poll` jest właśnie unikanie zachowań blokujących. Sam `poll` nie gwarantuje jednak, że przy obsłudze potencjalnie blokujących deskryptorów takie zachowanie nie wystąpi ...

> Using `poll()` or `select()` with a non-blocking file descriptor gives you two advantages:
>
> - You can set a timeout to block for;
> - You can wait for any of a *set* of file descriptors to become useable.
>
> If you only have a single file descriptor (socket) to wait for, and you don't mind waiting indefinitely on it, then yes; you can just use a blocking call.
>
> The second advantage is really the killer use case for `select()` and friends. It means that you can handle multiple socket connections, as well as standard input and standard output and possibly file I/O, all with a single thread of control.

Jak zapewnić, żeby wywołania *connect, accept, read, write* na gnieździe sieciowym zawsze zwróciły *EWOULDBLOCK* zamiast blokować się w jądrze?

```
int fcntl(int fd, int cmd, ... /* arg */ )
```

::: warning

- **cmd** *fcntl() performs one of the operations described below on the open file descriptor fd. The operation is determined by cmd.*
- **F_GETFL** *Return (as the function result) the file descriptor flags; arg is ignored.*
- **F_SETFL** *Set the file descriptor flags to the value specified by arg.* :::

Wystarczy zawołać na gnieździe *socketfd*:

```
fcntl(socketfd, F_SETFL, fcntl(socketfd, F_GETFL, 0) | O_NONBLOCK)
```

Można również od razu utworzyć gniazdo nieblokujące:

```
socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0)
```

Chcemy by po wywołaniu `poll` pierwsza instancja wyżej wymienionych wywołań systemowych nie zwróciła *EWOULDBLOCK*. Jaka wartośc musi być wpisana przez jądro do pola *revents* struktury *pollfd* dla danego deskryptora pliku, żeby mieć tą pewność?

```
struct pollfd {
   int   fd;         /* file descriptor */
   short events;     /* requested events */
   short revents;    /* returned events */
};
```

As its function result, *poll()* returns one of the following:

- A return value of −1 indicates that an error occurred. One possibl
  `EINTR`, indicating that the call was interrupted by a signal handler. (As
  Section 21.5, *poll()* is never automatically restarted if interrupted b
  handler.)
- A return of 0 means that the call timed out before any file descripto
  ready.
- A positive return value indicates that one or more file descriptors a
  The returned value is the number of *pollfd* structures in the *fds* array t
  nonzero *revents* field.

Bit	Input in *events*?	Returned in *revents*?	Description
`POLLIN`	•	•	Data other than high-priority data can be
`POLLRDNORM`	•	•	Equivalent to `POLLIN`
`POLLRDBAND`	•	•	Priority data can be read (unused on Linu
`POLLPRI`	•	•	High-priority data can be read
`POLLRDHUP`	•	•	Shutdown on peer socket
`POLLOUT`	•	•	Normal data can be written
`POLLWRNORM`	•	•	Equivalent to `POLLOUT`
`POLLWRBAND`	•	•	Priority data can be written
`POLLERR`		•	An error has occurred
`POLLHUP`		•	A hangup has occurred
`POLLNVAL`		•	File descriptor is not open
`POLLMSG`			Unused on Linux (and unspecified in SUS

::: warning

- **revents** *The field revents is an output parameter, filled by the kernel with the events that actually occurred. The bits returned in revents can include any of those specified in events, or one of the values POLLERR, POLLHUP, or POLLNVAL. (These three bits are meaningless in the events field, and will be set in the revents field whenever the corresponding condition is true.)*
- **POLLIN** *There is data to read.*
- **POLLOUT** *Writing is now possible, though a write larger than the available space in a socket or pipe will still block (unless O_NONBLOCK is set).* :::

Zadanie 7

Ściągnij ze strony przedmiotu archiwum «so21_lista_10.tar.gz», następnie rozpaku
UWAGA! Można modyfikować tylko te fragmenty programów, które zostały oznaczo

Zadanie 7. Program «echoclient-thread» wczytuje plik tekstowy zad
linie zakończone znakiem '\n'. Następnie startuje podaną liczbę wątków,
nawiązanie połączenia, wysłanie «ITERMAX» losowych linii wczytanego pl
robią to tak długo, aż do programu nie przyjdzie sygnał «SIGINT».

Twoim zadaniem jest tak uzupełnić kod, by program uruchamił «nthr
zakończenie. Czemu nie można go łatwo przerobić w taki sposób, żeby gł
pozostałych wątków tak, jak robi się to dla procesów przy pomocy wait

```
./echoserver-select 3000 &
./echoclient-thread echoserver-poll.c 10 127.0.0.1 3000
```

Zadanie 8

Zadanie 8. Program «echoserver-poll» implementuje serwer usługi
poll(2), ale bez użycia wątków i podprocesów. W kodzie wykorzystano
(ang. *event loop*) do współbieżnej obsługi wielu połączeń sieciowych. Pro
to samo, ale przy pomocy wywołania select(2).

Twoim zadaniem jest uzupełnienie pliku źródłowego «echoserver-poll.c
połączenia przychodzące, nadejście nowych danych na otwartych połą
Do przetestowania serwera użyj programu «echoclient-thread» z popr