L2.Z1. Wywołania systemowe `fork`, `waitpid`, `lseek`, `rename`, `mmap` mogą zakończyć się błędem. W jaki sposób system operacyjny komunikuje programiście przyczynę niepowodzenia wywołania? Sporządź listę niepowtarzających się kodów błędów tak, by dla każdego wywołania w zrozumiały sposób wyjaśnić słuchaczom co najmniej dwie przyczyny niepowodzenia.
Wskazówka: Zapoznaj się z sekcją ERRORS odpowiednich stron rozdziału 2 podręcznika systemowego - użyj polecenia `man`.

wywołanie systemowe:

- mechanizm wydawania wywołań systemowych jest w dużym stopniu zależny od maszyny i często musi być wyrażony w kodzie asemblera. Z tego powodu trzeba korzystać z biblioteki procedur, co pozwala na wydawanie wywsysów z poziomu programów w języku C
- każdy komputer z pojedynczym procesorem jest zdolny do uruchamiania tylko jednej instrukcji naraz. Co się dzieje, gdy proces uruchamia program użytkownika w user mode i wymaga usługi systemowej (np. czytania danych z pliku):
    - wykonanie rozkazu pułapki w celu przekazania sterowania do sysopka
    - sysopek dowiaduje się, czego chce proces wywołujący poprzez inspekcję parametrów
    - sysopek realizuje wywołane systemowe i zwraca sterowanie do następnej instrukcji za wywsysem

W jaki sposób system operacyjny komunikuje programiście przyczynę niepowodzenia wywołania?
Jeśli nie można zrealizować wywsysa, do zmiennej globalnej errno zapisywany jest numer kodu błędu.

`fork` - tworzy nowy proces, duplikując proces wywołujący

```
pid_t fork(void);
```

- EAGAIN - niepowodzenie alokacji wystarczającej ilości pamięci do skopiowania tablicy stron pamięci procesu-rodzica i alokacji struktury zadań dla procesu-dziecka
- ENOMEM - fork nie potrafi zaalokować niezbędnych struktur jądra z powodu niedostatecznej ilości pamięci
- ENOSYS - fork nie jest wspierany na tej platformie (np. Hardware bez MMU)

`waitpid` - zatrzymuje wykonywanie bieżącego procesu aż do zmiany stanu procesu potomka lub do dostarczenia sygnału kończącego bieżący proces

```
pid_t waitpid(pid_t pid, int *status, int options);
```

pid – process ID dziecka
status – waitpid zapisze informację o statusie w miejscu, na które status wskazuje
options - na jakie zachowanie dziecka zareagować

- ECHILD - proces o zadanym pid nie istnieje lub nie jest potomkiem procesu wywołującego
- EINTR – WNOHANG nie był ustawiony i przechwycono niezablokowany sygnał lub SIGCHLD (kiedy dziecko się zatrzymuje lub terminuje, SIGCHLD jest wysyłany do rodzica, defaultowa odpowiedź to zignorowanie)
- EINVAL – options jest invalid

`lseek` - zmiana pozycji „kursora” (offsetu) w pliku read/write

```
off_t lseek(int fd, off_t offset, int whence);
```

fd – deskryptor pliku skojarzony z otwartym plikiem
(Hola hola, czym jest **deskryptor pliku**? To identyfikator pliku w tablicy deskryptorów plików w jądrze, która przechowuje informacje o wszystkich otwartych plikach. Deskryptor może odnosić się do pliku, gniazda, katalogu, kolejki FIFO, strumienia czy dowiązania symbolicznego).
offset – o tyle jest przesuwany „kursor”
whence – polityka zmiany pozycji (czy przesunąć względem początku pliku czy względem aktualnej pozycji)
SEEK_SET – przesunięcie jest ustawione na offset bajtów
SEEK_CUR – aktualna pozycja plus offset bajtów
SEEK_END – rozmiar pliku plus offset bajtów

- EBADF – fd nie jest otwartym deskryptorem pliku
- EINVAL – whence nie jest valid lub wynikowe przesunięcie jest ujemne/poza końcem pliku

- EOVERFLOW – wynikowego offsetu nie da się zapisać do zmiennej off_t
- ESPIPE – fd jest związany z potokiem, gniazdem lub FIFO (a powinien być zwykłym plikiem)
- ENXIO – wartość whence to „SEEK_DATA” lub „SEEK_HOLE” i offset jest za końcem pliku

rename - zmienia nazwę lub lokalizację pliku

```
int rename(const char *oldpath, const char *newpath);
```

- EACCES – wymagany write permission lub search permission dla katalogu zawierającego oldpath lub newpath
- EBUSY – oldpath lub newpath są w katalogu używanym przez jakieś procesy
- EDQUOT – przydział bloków dysku użytkownika w systemie plików został wyczerpany
- EFAULT – oldpath lub newpath wskazują na miejsce poza dostępną przestrzenią adresową
- EISDIR – directory newpath istnieje, oldpath nie

mmap - tworzy odwzorowanie danej części pliku w przestrzeni adresowej procesu. Operacja ta powoduje, że do obszaru pliku można odnosić się jak do zwykłej tablicy bajtów w pamięci, eliminując potrzebę korzystania z dodatkowych wywołań systemowych typu read lub write. Z tego powodu często używa się tej operacji do przyspieszenia działania na dużych plikach.

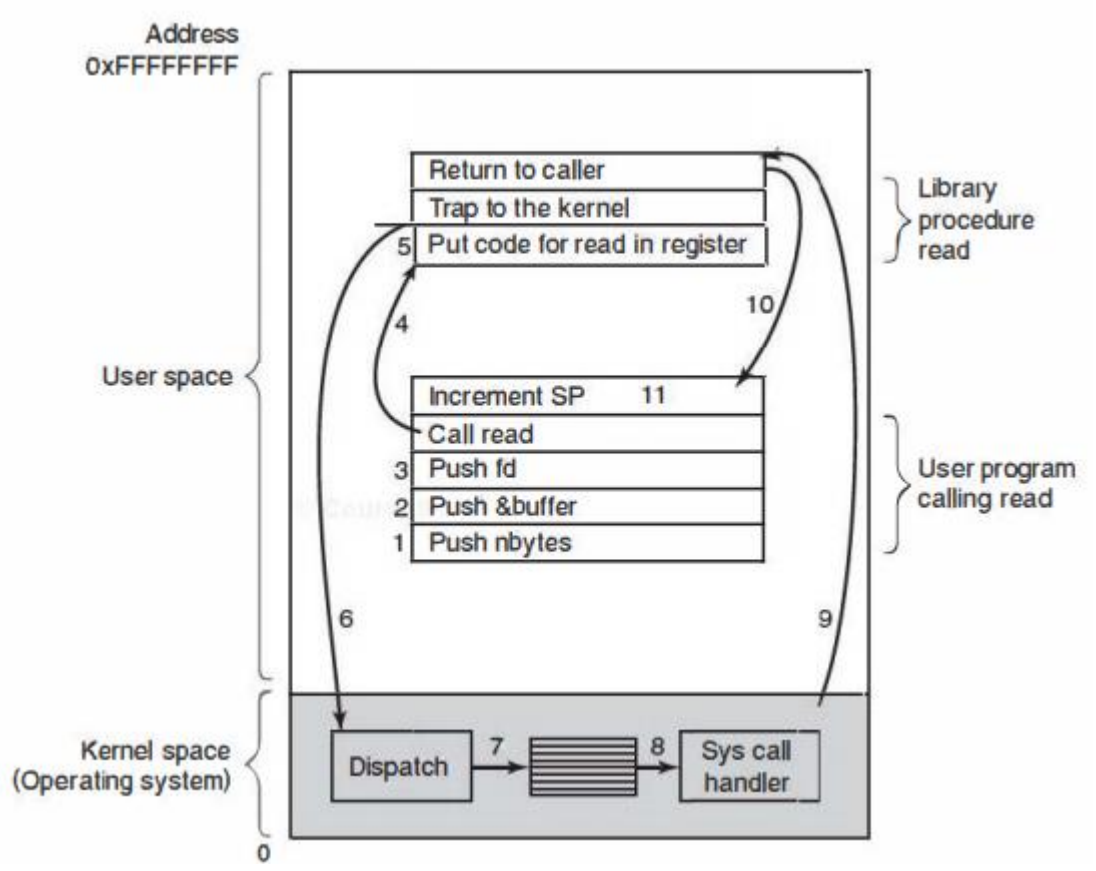
```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

addr – adres odwzorowania pliku
length – liczba bajtów, jaką chcemy odwzorować w pamięci
prot - flagi określające uprawnienia jakie chcemy nadać obszarowi pamięci, np. tylko do odczytu, etc.
flags - dodatkowe flagi określające sposób działania mmap
fd – deskryptor pliku, który chcemy odwzorować w pamięci
offset – od którego miejsca w pliku chcemy rozpocząć odwzorowywanie

- EACCES, EAGAIN, EBADF, EINVAL, ENOMEM
- ENFILE – osiągnięto systemowy limit otwartych naraz plików
- ENODEV – system plików tego pliku nie wspiera mapowania

Przebieg wywołania procedury bibliotecznej `read`, która faktycznie realizuje wywołanie systemowe `read`:

- 1-3: program wywołujący umieszcza parametry procedury na stosie (kompilatory C i C++ z powodów historycznych umieszczają parametry na stosie w odwróconej kolejności)
- 4: wywołanie procedury bibliotecznej
- 5: procedura biblioteczna, która może być napisana w języku asm, zwykle umieszcza numer wywsysa w miejscu, w którym sysopek się go spodziewa, np. w rejestrze
- 6: wykonanie instrukcji TRAP (pułapki) w celu przełączenia procesora z user mode do kernel mode i rozpoczęcia uruchamiania kodu od wskazanego adresu w jądrze
    - instrukcja występująca za TRAP jest pobierana z lokalizacji zdalnej, a adres powrotu jest zapisywany na stosie do późniejszego wykorzystania
    - instrukcja TRAP nie ma możliwości skoku pod dowolny adres, dlatego nie można przekazać do niej adresu, pod którym jest umieszczona procedura
- 7: kod jądra, który zaczyna działać za instrukcją TRAP, sprawdza numer wywsysa, a następnie przesyła go do właściwej procedury obsługi wywsysów (za pośrednictwem tabeli wskaźników do procedur obsługi wywsysów poindeksowanej wg numeru wywsysa)
- 8: uruchomienie procedury obsługi wywsysów
- 9: gdy procedura zakończy pracę, sterowanie może być zwrócone do procedury bibliotecznej przestrzeni usera – do następnej instrukcji za TRAP-em
- 10: procedura ta zwraca sterowanie do programu usera w sposób, w jaki standardowo następuje powrót sterownia z wywołań procedur
- 11: w celu zakończenia zadania program usera musi wyczyścić stos tak, jak po każdym wywołaniu procedury (usunięcie parametrów odłożonych na stos przed wywołaniem read)



Źródła: Tanenbaum 1.6, man (strony wywołań systemowych)

L2.Z2. W jakich przypadkach wywołanie **stat(2)**[1] mogłoby zakończyć się awarią systemu, gdyby jądro nie używało **funkcji kopiujących**[2] i nie sprawdzało poprawności argumentów? Czemu w trakcie przetwarzania wywołania systemowego należy skopiować dane pomiędzy **przestrzenią użytkownika** a **przestrzenią jądra**?

stat – zwraca informacje o pliku pod ścieżką path i zapisuje w strukturze buf lub o systemie plików, na którym dany plik się znajduje

```
int stat(const char *path, struct stat *sb);
```

funkcje kopiujące - kopiują dane z jednego adresu do drugiego (user space <-> kernel space)

`copy_from_user` – copy a block of data from user space

`copy_to_user` — copy a block of data into user space

- sprawdzają, czy dst jest accessible i writable przez dany proces (czy nie chcemy np. z poziomu usera pisać po adresach kernela)
- zwracają błąd EFAULT do userspace na wypadek, gdyby adres był nieosiągalny, zamiast crashować kernel

W jakich przypadkach wywołanie `stat`(2) mogłoby zakończyć się awarią systemu, gdyby jądro nie używało funkcji kopiujących i nie sprawdzało poprawności argumentów?

Powiedzmy, że stat nie sprawdza poprawności argumentów i podamy mu path, który w rzeczywistości nie istnieje. Ten nieistniejący adres będzie zmapowany na coś zupełnie innego (powiedzmy nawet, że coś z kernel space). Stat może nam powiedzieć, że w tym miejscu jest normalny plik, bo czyta tylko ciąg bitów bez sprawdzania poprawności. Potem my myślimy, że tam jest jakiś plik do którego chcemy potem coś nadpisać. Robimy write i nadpisujemy coś w kernel space i cały system leży.

przestrzeń użytkownika – porcja pamięci, w której żyją procesy użytkownika

przestrzeń jądra - przestrzeń niedostępna dla użytkownika (dostępna tylko przez syscalle), w której przechowywany jest kod jądra i w której kernel żyje

Czemu w trakcie przetwarzania wywołania systemowego należy skopiować dane pomiędzy przestrzenią użytkownika a przestrzenią jądra?

Należy skopiować dane z przestrzeni użytkownika do przestrzeni jądra, by nikt ich (celowo lub nie) nie zmienił w trakcie działania syscalla (co mogłoby sprawić, że np. uzyskalibyśmy informacje o pliku, do którego nie mamy dostępu, bo w ostatniej fazie działania syscalla zmieniliśmy nazwę pliku na inną).

Error detection jest zadaniem kernela, więc właśnie kiedy chcemy powiedzmy sprawdzić poprawność pliku, to musimy udać się do kernel space, nie możemy tego zrobić w user space (bo nie ma on dostępu do file system).

[1] stat() - http://netbsd.gw.com/cgi-bin/man-cgi?stat+2

[2] http://netbsd.gw.com/cgi-bin/man-cgi?copy+9

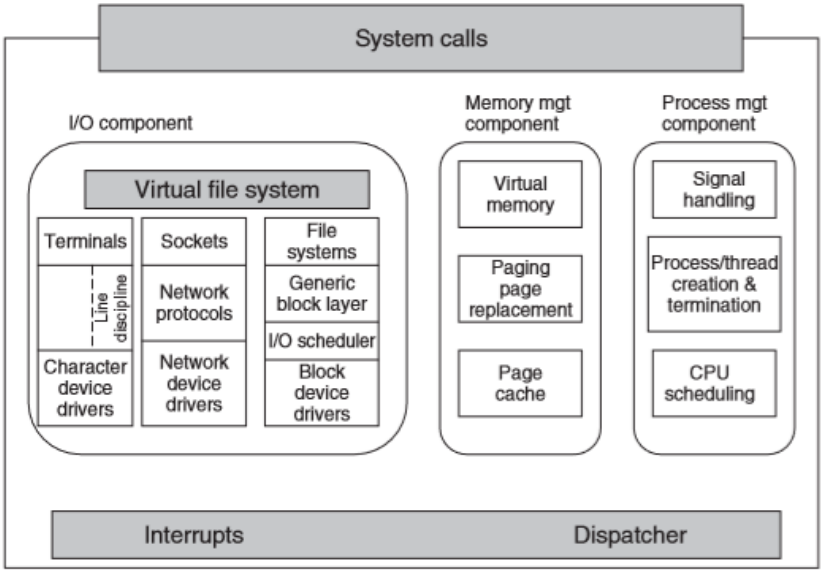
L2.Z4. Wymień główne komponenty **monolitycznego jądra** Linuksa. Czym charakteryzuje się jądro zorganizowane w **warstwy**? Wymień poważne wady architektury monolitycznej często przytaczane w literaturze. Które z nich można zniwelować przy użyciu **modułów jądra** oraz interfejsów typu FUSE?

systemy monolityczne – system operacyjny działa jako pojedynczy program w trybie jądra; najczęściej stosowane

- sysopek jest napisany jako kolekcja procedur powiązanych ze sobą w jednowarstwowy, rozbudowany binarny program wykonywalny
- każda procedura w systemie może wywołać dowolną inną, pod warunkiem, że zapewnia ona wykonanie przydatnych obliczeń
    - zaleta: gwarantuje to dużą wydajność
    - wady:
        - występowanie wielu tysięcy procedur, które bez ograniczeń wzajemnie się wywołują, często prowadzi do niezrozumiałego i trudnego do opanowania systemu
        - awaria w dowolnej z tych procedur może doprowadzić do unieruchomienia całego sysopka

Wymień główne komponenty **monolitycznego jądra** Linuksa

- procedury obsługi przerwań
- mechanizm przydziałów (ang. *dispatcher*)
- wejścia wyjścia – obejmuje wszystkie składniki jądra odpowiedzialne za interakcję z urządzeniami, operacje sieciowe i operacje IO związane z utrwalaniem danych. Na najwyższym poziomie wszystkie operacje IO są zintegrowane z warstwą wirtualnego systemu plików (ang. *Virtual File System* - VFS).
  (Hola hola, czym jest **wirtualny system plików**? Gdy programy z przestrzeni użytkownika chcą wykonać jakąkolwiek operację na pliku, odwołują się do funkcji VFS (np. open, read itp.). VFS przechwytuje wywołania systemowe i do realizacji operacji na pliku wywołuje funkcję konkretnego systemu plików (np. dla ext2 ext2_open, ext2_read itp.). Dzięki takiemu rozwiązaniu programy mogą korzystać z plików niezależnie od tego, jaki system plików został użyty do ich przechowywania.)
- zarządzania pamięcią - utrzymywanie odwzorowań pamięci wirtualnej w pamięć fizyczną, buforowanie ostatnio wykorzystywanych stron, przenoszenie na żądanie do pamięci nowych stron z niezbędnym kodem i danymi
- zarządzania procesami
    - tworzenie i kończenie procesów
    - obsługa sygnałów
    - szeregowanie czasu procesora – mechanizm koordynujący, szeregujący (ang. *scheduler*), który wybiera wątek wykonywany w pierwszej kolejności



systemy warstwowe – komponenty są podzielone na warstwy
Struktura współczesnego, wielowarstwowego systemu operacyjnego:

1. Ukrywanie najbardziej niezrozumiałych aspektów funkcjonowania sprzętu
2. Obsługa przerwań, przełączanie kontekstu i działania jednostki MMU, aby kod w wyższych warstwach był niezależny od sprzętu
3. Mechanizmy zarządzania wątkami (szeregowania i synchronizacji wątków)

4. Sterowniki urządzeń, z których każdy ma postać odrębnego wątku z własnym stanem, licznikiem programu, rejestrami itp. Sterowniki mogą, ale nie muszą działać w przestrzeni adresowej jądra.
5. Pamięć wirtualna
6. Jeden lub wiele systemów plików
7. Mechanizm obsługi wywołań systemowych

Czym charakteryzuje się jądro zorganizowane w warstwy?

- Każda kolejna warstwa nie musi się przejmować kolejnymi rzeczami. Np. pierwsza warstwa ukrywa sprzęt pod abstrakcjami. Kolejna dodaje obsługę przerwań i context switching. Od tego momentu warstwy nie przejmują się sprzętem.
- Warstwa n może wywoływać tylko warstwy n-1 (requesting services)i n+1 (answering services). Każda kolejna warstwa daje większe abstrakcje, unifikuje podobne schematy, urządzenia. Warstwy powinny mieć dobrze wyspecyfikowane zadania.

moduły jądra – kawałki kodu, które rozszerzają funkcjonalność jądra bez potrzeby rebootowania systemu.

- przykład: sterownik urządzenia, który daje dostęp kernelowi do hardware'u podłączonego do systemu
- bez nich jądro jest monolityczne i nowe funkcjonalności trzeba dodawać prosto do kodu kernela
- błąd w module nie prowadzi do awarii całego systemu, tylko danego komponentu

Wymień poważne wady architektury monolitycznej często przytaczane w literaturze.

- awaria jednego komponentu (np. sterownika) powoduje awarię całego systemu
- mamy dostęp do wszystkich funkcji w całym jądrze, zmieniając jedno, możemy popsuć inne
- takie jądro to jeden wielki program – duża liczba linii, co statystycznie daje dużą szansę na sporą liczbę błędów

Które z tych wad można zniwelować dzięki użyciu modułów jądra oraz interfejsów typu FUSE?

Filesystem in Userspace (FUSE) – interfejs dla uniksowych sysopków pozwalający nieuprzywilejowanym userom tworzyć ich własne systemy plików bez konieczności edytowania kodu jądra. Jeśli obsługa systemu plików będzie w user space to w kernelu to nic nie popsuje. Wtedy błędy w systemie plików to błędy w user space.

Źródła: Tanenbaum 1.7, 10.2.5,

L2.Z5 Jądro systemu WinNT jest łatwo przenośne (ang. portable) dzięki implementacji warstwy abstrakcji sprzętu (ang. hardware abstraction layer). Jakie zadania pełni HAL (§11.3)? Jakie korzyści daje HAL programistom implementującym sterowniki urządzeń?

Hardware Abstraction Layer - najniższa warstwa kodu jądra, zapewnia abstrakcję szczegółów hardware (rejestry, DMA). HAL daje zunifikowany interfejs dla wyższych warstw abstrakcji. Dla każdego sprzętu implementacja się różni, ale nterfejs jest ten sam.

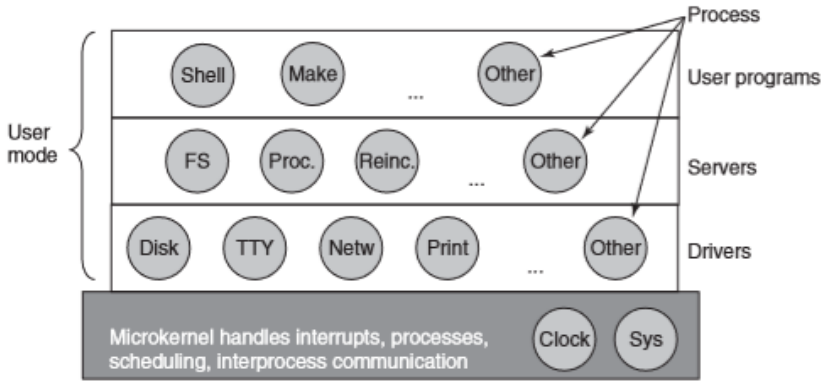
Portable Operating System - system jest przenośny, jeśli jest w stanie działać na wielu platformach hardware. Idealnie byłoby po prostu skompilować SO na inną platformę i ma też działać. Nie da się tego jednak czasem zrobić, bo powiedzmy CPU jest 32 albo 64 bitowy.

Sterownik urządzenia – program lub frangment programu odpowiadający za dane urządzenie i pośredniczący pomiędzy nim a resztą systemu komputerowego

Sterowniki korzystają z HAL jako interfejsu. Dzięki temu nie trzeba pisać sterowników osobno na różne urządzenia, wystarczy żeby urządzenie korzystało z tego samego HAL. Wtedy nie ma znaczenia jakie powiedzmy rejestry są w CPU, bo HAL i tak zapewnia jednakową abstrakcję dla różnych urządzeń.

HAL zapewnia abstrakcję dla następujących elementów: Rejestry urządzeń, przerwania, DMA, timery i zegary, Firmware (BIOS, Basic Input/Output System).

L2.Z6. Podaj motywacje stojące za wprowadzeniem systemów operacyjnych opartych na **mikrojądrach**.
Ze względu na sposób komunikacji międzyprocesowej (ang. *Interprocess Communication*) w takich systemach każdy proces może pełnić rolę **klienta** lub **serwera**. Czy systemy z mikrojądrem to naturalni kandydaci na **rozproszone systemy operacyjne**?



mikrojądro:

- motywacje - dążenie do osiągnięcia wysokiej niezawodności poprzez dzielenie sysopka na niewielkie, dobrze zdefiniowane moduły, z których tylko jeden – mikrojądro – działa w kernel mode, natomiast pozostałe działają jako zwykłe procesy użytkownika o relatywnie małych możliwościach
    - jeśli w trybie jądra będzie jak najmniej funkcji, to mniejsza szansa, że dojdzie do awarii całego systemu na skutek błędu w jądrze
    - błąd w sterowniku spowoduje awarię komponentu, nie całego systemu
    - procesy użytkownika można skonfigurować tak, aby miały mniejsze możliwości, dzięki temu występujące w nich błędy nie musza być krytyczne
- poza jądrem system ma strukturę trzech warstw procesów
    - wszystkie działają w user mode
    - najniższa zawiera sterowniki urządzeń, ponieważ działają one w user mode, nie mają fizycznego dostępu do przestrzeni IO, zamiast tego jądro może sprawdzić, czy sterownik zapisuje lub odczytuje dane z IO device, z którego ma prawo korzystać (zapobiega to sytuacji, w której błąd sterownika implikuje awarię systemu)
    - nad sterownikami znajduje się warstwa zawierająca serwery, które realizują większość usług sysopka
        - zarządzanie systemem plików
        - zarządzanie procesami (tworzenie, niszczenie)
- kandydatura - tak. Komunikacja jak w sieci. Klient wysyła wiadomość z prośbą o coś. Serwer przetwarza i odsyła wynik. Nie jest ważne, czy żądania między procesami przechodzą przez komunikację wewnątrz komputera, czy przez sieć.

model klient-serwer:

- dwie klasy procesów:
    - **serwery** – każdy udostępnia pewne usługi
    - **klienci** – korzystają z tych usług
- komunikacja odbywa się poprzez przekazywanie komunikatów. Uzyskanie usługi: klient tworzy komunikat z informacją o tym, czego chce, i przesyła go do odpowiedniej usługi. Usługa wykonuje pracę i przesyła odpowiedź.
- uogólnieniem tej koncepcji jest uruchomienie klientów i serwerów na różnych komputerach połączonych ze sobą lokalną lub rozległą siecią, ponieważ klienci nie muszą wiedzieć, czy komunikaty są obsługiwane lokalnie na ich własnych maszynach, czy też są one przesyłane w sieci do serwerów na zdalnej maszynie

rozproszone systemy operacyjne – celem sysopka rozproszonego jest przekształcenie luźno powiązanego zbioru komputerów w spójny system bazujący na jednej koncepcji

Źródło: Tanenbaum 1.7.3, 1.7.4