# Systemy operacyjne

Laboratorium 1

Mateusz Małek
10 marca 2017

# Jak mnie złapać

- **(inż.) Mateusz Małek**
  - *Cześć/Dzień dobry/Szanowny Panie*…
  - Na pewno nie *Szanowny Panie Doktorze* - przynajmniej na razie…
- **Pokój:** D-17, 2.48 (“Laboratorium robotów”)
- **Telefon:** (+48) 12 32 83 432
  - (Na imię mi admin bo) Jest nas wielu w tym pokoju
- **E-mail:** mateusz.malek@iisg.agh.edu.pl
  - **Preferowana** forma kontaktu
- **Konsultacje:** tak
  - W razie potrzeby - indywidualnie ustalony termin
- **“Grasuję” też na forum i czacie kierunku:** mkwm

# Wymagania i zasady zaliczenia

# Zasady zaliczenia

- Wszystkie szczegóły w kursie “Systemy Operacyjne 2016/2017” na UPeL
  - **Adres kursu:** http://upel.agh.edu.pl/wiet/course/view.php?id=344
  - **Hasło dla naszej grupy:** MM_SysOpy_1617
- Ocena z laboratorium jest średnią ważoną
  - Zestawy zadań i aktywność (waga 0.5)
  - Kolokwia (waga 0.25)
  - Projekt (waga 0.25)
- Oceny są wyznaczane zgodnie ze skalą ocen z regulaminu studiów na AGH
  - 5.0 gdy >= 91%
  - 4.5 gdy >= 81%
  - 4.0 gdy >= 71%
  - 3.5 gdy >= 61%
  - 3.0 gdy >= 50%
  - 2.0 gdy < 50%

# Zasady zaliczenia - zestawy zadań i aktywność

- Obowiązuje język C
- Czas na oddanie każdego zestawu to (zwykle) tydzień
  - Dokładniej: do 23:59 w czwartek poprzedzający kolejne zajęcia
- Przekroczenie terminu o <= tydzień: obniżenie finalnej oceny zestawów o 0.2
  - Za każdy spóźniony zestaw!
- Przekroczenie terminu o > tydzień: brak zaliczenia w pierwszym terminie
- Do zaliczenia wymagana pozytywna ocena (z każdego z osobna i łącznie)
- Ocena na podstawie 3-4 odpowiedzi ustnych z prezentacją zadania

- Mogą pojawić się kartkówki z materiału poprzedniego laboratorium :(
- Praca nad bieżącym zestawem już na zajęciach będzie nagradzana :)
- Każda ocena z aktywności to [-0,2; +0,2] do oceny końcowej ze zestawów

# Zasady zaliczenia - kolokwia

- Planowane 2 kolokwia w semestrze
- Do zaliczenia wymagana średnia z obu kolokwiów >= 50%
- Forma testu wielokrotnego wyboru w systemie UPeL
  - Bez korzystania z dokumentacji
  - Bez korzystania ze źródeł zewnętrznych
- Być może odbędzie się kolokwium praktyczne
  - Zadania postaci “uzupełnij implementację funkcji która X”
  - Z dostępem do dokumentacji
  - Bez korzystania ze źródeł zewnętrznych

# Zasady zaliczenia - projekt

- Oddanie projektu nie jest obowiązkowe
  - Brak projektu lub niezaliczony projekt ogranicza ocenę z laboratorium do maksymalnie 4.0
- Tematy i wymagania dla projektów zostaną podane około połowy semestru
- W ramach grupy wybrane tematy projektów nie mogą się powtarzać
- Można zaproponować własny temat projektu
  - Musi być przekrojowy
  - Musi być związany z systemami operacyjnymi
  - Trzeba przekonać prowadzącego :)

# Laboratorium 1

opcje kompilatora gcc/g++; narzędzia Make i GDB; zarządzanie pamięcią; biblioteki statyczne i ładowane dynamicznie; pomiar czasu

# Opcje kompilatora gcc/g++

- Kontrolowanie “jak bardzo” zbudować program
  - `-E` - zakończ przetwarzanie na preprocesorze
    - Wszystkie #include, #define i #if zostały rozwinięte w pełny kod
  - `-S` - zakończ przetwarzanie na kompilacji
    - Kod został przekształcony do postaci instrukcji asemblera (domyślnie do pliku *.s)
    - Można tutaj obserwować działanie różnych optymalizacji
  - `-c` - zakończ przetwarzanie na asemblowaniu
    - Mamy gotowy kod maszynowy, ale program jeszcze nie nadaje się do uruchomienia (np. “obiecaliśmy” że funkcja x istnieje, ale jeszcze nie wiadomo gdzie ona jest)

# Opcje kompilatora gcc/g++

- Różne poziomy optymalizacji
  - `-O0` - wyłączenie optymalizowania kodu, przydatne przy debugowaniu
  - `-O` lub `-O1` - podstawowy poziom optymalizacji
  - **`-O2` - zalecany poziom optymalizacji, zawiera wszystko z `-O1` + kilka dodatkowych**
  - `-Os` - wariant `-O2`, w którym wyłączone są optymalizacje mogące zwiększyć rozmiar kodu
  - `-O3` - zawiera wszystko z `-O2` + dodatkowe, agresywne optymalizacje
  - `-Ofast` - wariant `-O3`, w którym aktywowane są dodatkowe optymalizacje mogące uczynić kod niezgodnym ze standardem języka
- Wykorzystanie różnych zestawów instrukcji procesora
  - `-march=…` - nazwa konkretnej rodziny procesorów lub opcje takie jak `generic` (“uniwersalny”) lub `native` (dokładnie pod ten komputer, na którym kompilujemy źródła)
- Ok… Ale co tak właściwie znaczy?
  - `gcc --help=optimizers`
  - `gcc -Q -O1 --help=optimizers`
  - `gcc --help=targets`
  - `gcc -Q -march=native --help=targets`
  - …

# Opcje kompilatora gcc/g++

- Opcje linkera:
  - `-l`*`biblioteka`* - dołączenie biblioteki do programu
  - `-L./sciezka` - dodanie ./sciezka do katalogów w których poszukiwane będą biblioteki
  - `-shared` - służy do skompilowania biblioteki dzielonej
  - `-static` - służy do zlinkowania programu i bibliotek w sposób statyczny
- Inne przydatne opcje:
  - `-DNAZWA=WARTOSC` - ma efekt identyczny jak umieszczenie w kodzie linijki `#define NAZWA WARTOSC`
  - `-g` - umieszcza w programie wynikowym informacje pozwalające na jego skuteczne debugowanie (np. przy użyciu GDB)
  - `-fPIC` - Position Independent Code, potrzebne przy bibiliotekach ;)
  - `-Wall` - włącza wszystkie możliwe ostrzeżenia w trakcie kompilacji
  - `-std=...` - pozwala podać wersję standardu języka, której chcemy używać (np. `-std=c11` dla standardu języka C z roku 2011)
  - `-pthread` - włączenie obsługi POSIXowych wątków
  - `-m32`, `-m64` - skompilowanie programu w wersji 32-bitowej lub 64-bitowej
    - Na 64-bitowej Fedora może być potrzebne wydanie komendy `dnf install glibc-devel.i686`

# Narzędzie Make

- Pamiętanie poleceń do kompilacji - uciążliwe
- Wyjaśnianie użytkownikowi jak ma skompilować program - jeszcze gorsze
- A gdyby tak po prostu wpisać jedną komendę, np. `make`?

- Tworzymy plik tekstowy o specjalnej strukturze z opisem budowy różnych elementów programu
- Taki plik nazywamy Makefile (lub makefile, w ostateczności GNUmakefile)
- Po wywołaniu komendy `make` plik zostaje przetworzony, a zażądana przez użytkownika operacja - wykonana

- W większych projektach Makefile tworzone są zwykle automatycznie
  - skrypty ./configure, pakiet Autotools (Autoconf, Automake, Libtool), CMake

# Narzędzie Make

- Dokumentacja pod adresem: https://www.gnu.org/software/make/manual/make.html
- Reguły postaci:

```
cele …: zależności …
	polecenia
	…
```

- “Automatic variables”:
  - `$^` - lista zależności
  - `$@` - nazwa celu
- Zmienne specjalne:
  - CC - kompilator C
  - CFLAGS - flagi kompilacji C
  - CPPFLAGS - flagi preprocesora C
  - LDFLAGS - flagi linkera
  - LDLIBS - nazwy bibliotek do zlinkowania z programem
- .PHONY - stosowane do wskazania że dany cel nie ma odzwierciedlenia jako plik (w przykładzie obok: gdybyśmy utworzyli plik clean, to `make clean` nigdy nie byłoby wykonywane - bo Make uznawałoby, że “clean jest aktualne”)

```
CFLAGS = -O2
LDLIBS = -lm

all: program

%.o: %.c
	$(CC) -c $(CFLAGS) $(CPPFLAGS) $^ -o $@

program.o: CFLAGS += -DSYSOPY

%: %.o
	$(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@

.PHONY: clean

clean:
	-rm -rf *.o program
```

# Słowo wstępne do bibliotek...

- Żeby korzystanie z bibliotek miało sens, należy oddzielić deklaracje funkcji od ich definicji
  - Deklaracje funkcji (nazwa, argumenty, zwraca wartość) i struktur - w pliku nagłówkowym *.h
  - Definicje (kod implementacji) - w pliku *.c biblioteki
- W kodzie który będzie korzystał z biblioteki include'ujemy plik nagłówkowy *.h
- Plik nagłówkowy powinien zawierać tzw. "header guard" - żeby uniknąć błędów "already declared":

```
#ifndef _NAGLOWEK_H
#define _NAGLOWEK_H
// tutaj deklaracje
#endif
```

- Jeśli bibliotekę tworzymy w C++ i chcemy z niej korzystać w C, musimy deklaracje umieścić w bloku extern "C":

```
extern "C" {
    // tutaj deklaracje
}
```

Jest to związane ze sposobem w jaki linker "radzi sobie" z przeciążaniem funkcji w C++ - symbol funkcji musi być unikatowy, w związku z czym nazwy funkcji zostają przekształcone tak, by zawierały pewną informację o ich argumentach.

# Biblioteki statyczne i ładowane dynamicznie

- Biblioteka statyczna:
  - Zwykle w postaci archiwum *.a
  - Archiwum zawiera zasemblowane pliki *.o
  - Dołączenie do programu biblioteki statycznej powoduje włączenie jej skompilowanego i zasemblowanego kodu w skład programu
  - Program zlinkowany z biblioteką statycznie zajmuje więcej miejsca na dysku, ale nie wymaga dostarczania tej biblioteki dodatkowo (bo zawiera ją “w sobie”)
- Biblioteka dzielona/ładowana dynamicznie
  - W postaci pliku *.so
  - Można dołączyć ją do programu na dwa sposoby:
    - W trakcie linkowania - w programie zostaną umieszczone wskazania na symbole pochodzące z biblioteki (zamiast jej kodu)
    - W trakcie działania programu - odwoływać się do niej z wykorzystaniem funkcji z nagłówka dlfcn.h i biblioteki dl
  - Jeśli z biblioteki korzysta wiele programów, oszczędzamy miejsce i ułatwiamy sobie jej aktualizowanie
  - Potencjalny koszt: zaktualizowana biblioteka przestaje być zgodna z programami które z niej korzystały

# Biblioteki statyczne

- Tworzymy zasemblowany kod biblioteki:
  `gcc -c biblioteka.c -o biblioteka.o`
- Pakujemy bibliotekę do archiwum:
  `ar rcs **lib**bilioteka**.a** biblioteka.o`
- W trakcie kompilacji naszego programu:
  `gcc program.c -L. -lbiblioteka -o program`
  - `-L.` - oznacza szukanie biblioteki/archiwum z nią w bieżącym katalogu
  - przedrostek lib i rozszerzenie .a (lub .so) zostaną automatycznie uwzględnione w trakcie poszukiwań biblioteki
- Jeśli istnieje zarówno statyczna jak i dzielona wersja biblioteki, to domyślnie zostanie użyta biblioteka dzielona; możemy wymusić statyczne linkowanie całego programu:
  `gcc program.c -static -L. -lbiblioteka -o program`
  - **Żeby taka kompilacja się powiodła, w systemie musi być obecna również statyczna wersja biblioteki standardowej C!** (W Fedorze można ją doinstalować komendą `dnf install glibc-static`)

# Biblioteki dzielone

- Tworzymy bibliotekę:

  `gcc -fPIC -shared biblioteka.c -o libbiblioteka.so` (możemy też użyć biblioteka.o zamiast biblioteka.c)
  - `-fPIC` - w różnych programach biblioteka może trafić pod różne adresy w pamięci, więc instrukcje skoku muszą być względne, a nie do konkretnych adresów (Position Independent Code)
- W trakcie kompilacji naszego programu:

  `gcc program.c -L. -Wl,-rpath=. -lbiblioteka -o program`
  - `-L.` - oznacza szukanie biblioteki w bieżącym katalogu na etapie linkowania (w celu odczytania tablicy symboli)
  - `-Wl,-rpath=.` - oznacza szukanie biblioteki w bieżącym katalogu w momencie uruchomienia programu
- Użycie `-Wl,-rpath=.` przydaje się raczej w eksperymentach; na co dzień zwykle stosuje się jedno z następujących podejść:
  - umieszczenie biblioteki w odpowiednich lokalizacjach systemowych (np. /usr/lib64)
  - dodanie katalogu z biblioteką do konfiguracji dynamicznego loadera (/etc/ld.so.conf) i wywołanie ldconfig
  - wskazanie w zmiennej środowiskowej LD_LIBRARY_PATH dodatkowych katalogów do uwzględnienia w trakcie poszukiwań biblioteki
- Komenda `ldd` pozwala nam podejrzeć z jakimi bibliotekami został dynamicznie zlinkowany nasz program

# Biblioteki dzielone ładowane przez program

- Korzystamy z tego samego pliku *.so z biblioteką co w przypadku linkowania dynamicznego
- W trakcie kompilacji naszego programu nie odwołujemy się do naszej biblioteki; zamiast tego dołączamy bibliotekę dynamicznego loadera (dl):

  ```
  gcc program.c -ldl -o program
  ```
- Zmienia się konstrukcja naszego programu:
  - Dołączamy plik nagłówkowy dlfcn.h
  - Otwieramy plik z biblioteką funkcją void ***dlopen**(const char *filename, int flags)
  - Pozyskujemy z otwartej biblioteki uchwyt do funkcji z użyciem void ***dlsym**(void *handle, const char *symbol)
    - Konieczne wykonanie akrobacji z ustawieniem prawidłowego typu zmiennej-wskaźnika do funkcji :(
  - Wywołujemy funkcję, korzystając z zapisanego chwilę wcześniej wskaźnika
  - Zamykamy bibliotekę funkcją int **dlclose**(void *handle)
  - Manual: man 3 dlsym
- Po co tak kombinować?
  - Różnego rodzaju wtyczki i rozszerzenia, które wskazujemy w konfiguracji oprogramowania

# Przykład - użycie funkcji cosinus z biblioteki matematycznej C

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(void) {
   void *handle;
   // Zadeklarowanie zmiennej wskaźnikowej cosine
   double (*cosine)(double);
   char *error;

   // Otwarcie biblioteki
   handle = dlopen("/lib64/libm.so.6", RTLD_LAZY);
   if (!handle) {
       fprintf(stderr, "%s\n", dlerror());
       return 1;
   }

   // Uzyskanie uchwytu do funkcji cos w bibliotece
   cosine = (double (*)(double)) dlsym(handle, "cos");

   // Obsługa ewentualnego błędu
   error = dlerror();
   if (error != NULL) {
       fprintf(stderr, "%s\n", error);
       return 1;
   }

   // Użycie funkcji - zauważ operator wyłuskania *
   printf("%f\n", (*cosine)(2.0));

   // Zamknięcie biblioteki
   dlclose(handle);

   return 0;
}
```

# Zarządzanie pamięcią

- nagłówek stdlib.h
- void *malloc(size_t size)
  - Zaalokowanie size bajtów pamięci bez jej inicjalizowania
  - Uwaga na overflow iloczynu jeśli robimy coś w stylu `obszar = malloc(nmemb * size);`!
- void *calloc(size_t nmemb, size_t size)
  - Zaalokowanie nmemb * size (bez obaw o overflow :) ) bajtów pamięci + zainicjalizowanie zerami
  - Prawie jak malloc + memset… Prawie, bo używa “oszustwa” w postaci Copy-on-Write ;)
    - Patrz: https://vorpus.org/blog/why-does-calloc-exist/
- void *realloc(void *ptr, size_t size)
  - Zmiana rozmiaru zaalokowanego dynamicznie obszaru pamięci
  - Jeśli nie dało się po prostu go rozszerzyć, przekopiuje dane w nowe miejsce i zwolni stary
- void free(void *ptr)
  - Zwolnienie zaalokowanego dynamicznie obszaru pamięci
  - Próba ponownego zwolnienia już zwolnionego obszaru jest błędem!
- Manual: man 3 malloc

# Pomiar czasu

- Zewnętrznie (w środowisku):
  - Wbudowana funkcja powłoki bash:

```
[mkwm:~] $ time /usr/bin/sleep 3
real 0m3.004s
user 0m0.000s
sys  0m0.003s
```

  - Polecenie time (wymagana podanie pełnej ścieżki, żeby nie wykorzystać funkcji powłoki!):

```
[mkwm:~] $ /usr/bin/time /usr/bin/sleep 3
0.00user 0.00system 0:03.00elapsed 0%CPU (0avgtext+0avgdata 2020maxresident)k
0inputs+0outputs (0major+66minor)pagefaults 0swaps
```

    - Manual: man 1 time (zawiera m.in. sposób zmiany formatu wyświetlania wyniku)

# Pomiar czasu

- Wewnętrznie (w programie):
  - clock_t times(struct tms *buf) z sys/times.h
    - Przez podany wskaźnik aktualizuje strukturę ze zużyciem czasu procesora i użytkownika w “tickach”
    - Ilość “ticków” w sekundzie zwróci wywołanie: `sysconf(_SC_CLK_TCK);`
    - “To measure changes in elapsed time, use clock_gettime(2) instead” - co pewien czas jest overflow
    - Manual: man 2 times, man 3 times
  - int clock_gettime(clockid_t clk_id, struct timespec *tp) z time.h
    - Interesujące z naszego punktu widzenia są zegary CLOCK_REALTIME i/lub CLOCK_MONOTONIC
    - Przez podany wskaźnik aktualizuje strukturę z sekundami i nanosekundami dla wybranego zegara
    - Manual: man 3 clock_gettime
  - int getrusage(int who, struct rusage *usage) z sys/resource.h
    - Pozwala odczytać zużycie zasobów dla “wołającego” procesu lub wątku oraz dla procesów potomnych
    - Przez podany wskaźnik aktualizuje strukturę która zawiera dużo więcej niż tylko czas procesora i czas użytkownika
    - Manual: man 2 get_rusage

# Narzędzie GDB

- Debugger (“odpluskwiacz”)
- Pozwala na:
  - Uruchomienie nowego procesu lub podłączenie się do istniejącego i śledzenie ich wykonania
  - Analizę “post mortem” przyczyn awarii programu w sytuacji kiedy dostępny jest jego zrzut pamięci z momentu zdarzenia (“core dump”)
- Żeby z GDB można było skutecznie korzystać:
  - Program powinien być skompilowany z opcją `-g` (lub powinny być dostępne jego symbole debugowania - takie podejście jest typowe dla paczek z repozytoriów dystrybucji Linuksa)
  - Jeśli chcemy analizować pliki core, to musimy zmienić limit ich wielkości na niezerowy:
    `ulimit -c unlimited`
    (Obowiązuje - w przybliżeniu - w konkretnym oknie terminala)

# Jak rozpocząć pracę z GDB?

- Uruchomienie programu pod kontrolą GDB:

```
[mkwm:~] $ gdb ./program
…
(gdb)
```

- Podłączenie się do już uruchomionego procesu danego programu:

```
[mkwm:~] $ gdb ./program
…
(gdb) attach NR_PID_PROCESU
(gdb)
```

- Analizowanie pliku core.PID:

```
[mkwm:~] $ gdb -c core.PID ./program
…
(gdb)
```

# Co możemy robić w shellu GDB?

Kiedy widzimy prompt `(gdb)`, możemy używać licznych komend - oto niektóre:

- **run** - uruchomienie programu :)
- **list; list A,B** - wyświetlenie źródła lub jego fragmentu od linii A do linii B
- **break N** - ustawienie breakpointa w linii N
- **disable/enable** X - włączenie lub wyłączenie breakpointa X
- **watch X** - zatrzymywanie programu przy zmianach wartości zmiennej X
- **step** - wykonanie pojedynczego kroku po zatrzymaniu programu na breakpoince
- **finish** - przeskoczenie do końca pętli, funkcji…
- **continue** - kontynuowanie wykonania programu do następnego breakpointa
- **set variable X = WARTOSC** - zmiana wartości zmiennej X na wartosc
- **print X** - wyświetlenie wartości zmiennej X
- **info locals** - wyświetlenie informacji o wartościach zmiennych lokalnych
- **info args** - wyświetlenie informacji o argumentach funkcji
- **info break** - wyświetlenie informacji o aktualnie ustawionych breakpointach
- **info signals** - wyświetlenie informacji o aktualnym sposobie obsługi sygnałów
- **info sharedlibrary** - wyświetlenie informacji o załadowanych bibliotekach dzielonych
- **info threads** - wyświetlenie informacji o aktulnie działających wątkach
- **backtrace; bt** - wyświetlenie jak znaleźliśmy się w danym miejscu programu (ramki stosu)

Dziękuję za uwagę