Processes:
sleep()

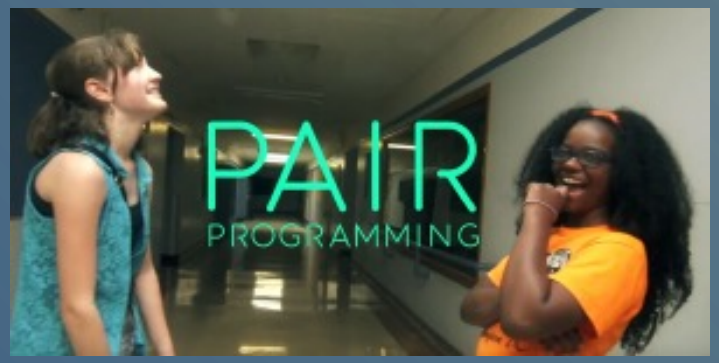
Instructor: Linyi Li
Slides adapted from Dr. B. Fraser

Topics

1) What specifically is a running program?
2) Writing C code to call a syscall: sleep()
3) Using man pages.
4) Fun with some C pointers.

Pair Programming

- In lecture, we'll do lots of programming activities!
  - You and a partner will use
    .. one computer to write code
  - Show: Pair Programming (by Code.org)



- Suggestion
  - Driver typing the code
  - Navigator look up the man page
  - Both are creating the code!

- See ordinary pair programming session (show 30s)

Process

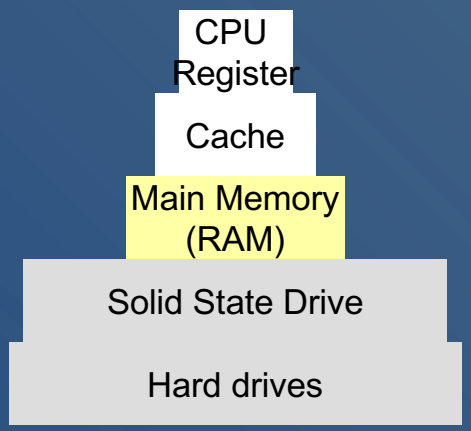
Process

- What is a program?
  - Basically a.. compiled executable file.
  - But unless you run it, it's just a file!

- What is a process?
  - Basically a.. running program.

  (not quite that simple; we'll learn more)

Program in Memory

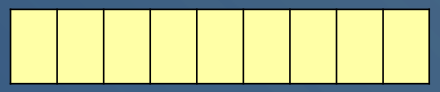


Memory Hierarchy

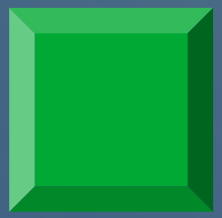
- ..CPU can execute instructions from memory.
- Program (the executable) stored on disk.
  - Slow data access (fetch) speed due to distance, spinning drive, etc.
  - CPU cannot access bytes without loading them into memory.
  - So, a program must be in memory to run.

Slow storage

Data loaded into main memory



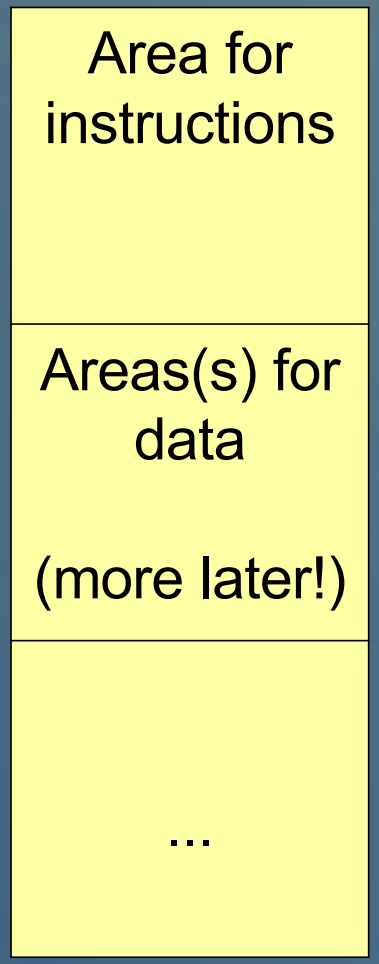
Bytes in Memory: Fast CPU access



CPU

Start Execution

- To start executing a program, the OS will:
  - .. create a memory space in RAM for the program to run
  - load the machine code from the program's file on disk into memory.
  - make part of memory space for data (variables, ...). More later!
  - start executing the program from memory (makes it a process!)



Areas of program's memory space

Controlling a Process

- Controlling a process
  - Programmers use system calls (syscalls) to control processes.
- Some core process syscalls include:
  - .. fork()

    Create a new process by cloning current one.
  - .. exec()

    Replace current process with another executable.
    (family of different calls, but do the same thing).
  - .. wait()

    Wait until a created process finishes its work.

ABCD: Process

- What is the difference between a process and a program?

(a) A process is a program loaded into memory and running.

(b) A program is a process loaded into memory and running.

(c) A process is loaded from RAM to the hard drive by the OS.

(d) A program is loaded from RAM to the hard drive by the OS.

Coding &
Process Activity

Ready to Code

- Open Two Terminals (tabs or windows)
  - A terminal for Coding:
    - Launch the CMPT 201 container:
      docker start -ai cmpt201
    - Make a folder for our work
      mkdir -p ~/lecture/02-forkexecwait
  - A terminal for 'man' page:
    - connect to the already running container:
      docker exec -it cmpt201 zsh --login
    - Run
      man 3 printf

      If not yet downloaded docker image, first run:
      docker create -it --name cmpt201 ghcr.io/sfu-cmpt-201/base # if needed

Activity: Hello C World!

- Create a C program:

cd ~/lecture/02-forkexecwait/
nvim hello.c

- Compile

`clang hello.c`

–This builds executable a.out; run it:
./a.out

–Set executable's name:
clang hello.c -o hello

- (3 mins)
  You do it now!

```c
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
}
```

Activity: sleep()

- (5 mins)
  Write a program that keeps calling `sleep()` with some timeout value.
  - Check the man page for sleep():

```
$ man 3 sleep
```

(Without the 3, it will give you the Linux sleep command)

- In a 3rd terminal, run btop
  - Connect to running container using `docker exec...`
  - btop is a good tool to visualize parent/child processes

sleep() Solution

- See process information: btop
  - Use tree view (press e)
  - Each process has a parent (except init and kthreadd; not shown in containers).
  - Our container's zsh runs a.out

sleep.c > ...

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>


int main()
{
    char* message = "Hello world!\n";
    for (int i = 0; i < strlen(message); i++) {
        printf("%c", message[i]);
        fflush(stdout);
        sleep(2);
    }
    printf("\n");
    printf("DONE\n");
}
```

```
proc filter per-core reverse tree < cpu lazy >
Tree:                                  User:  MemB      Cpu%
[-]-1 systemd (init)                   root    14M       0.0
    ├─ 59774 packagekitd               root    70M       0.0
  [+]-1514 systemd                     brian  3.4G       1.7
    ├─ 59665 fwupd                     root   124M       0.0
    ├─ 675 vmtoolsd                    root    11M       0.0
```

On Linux shows init

```
proc filter per-core reverse tree < pid >
Tree:                                  User:  MemB      Cpu%
[-]-128 zsh                            cmpt+  6.8M      0.0
    └─ 694 btop                        cmpt+  6.2M      0.5
  ├─ 66 zsh                            cmpt+  6.8M      0.0
  └─ 1 zsh                             cmpt+  9.3M      0.0
```

In container, no init

ABCD: Docker

- Which command connects to
  an already running Docker container?
- Which command downloads the Docker container?
- Which command launches the Docker container?

```
(a) docker start -ai cmpt201
(b) docker exec -it cmpt201 zsh --login
(c) docker git clone github.com/sfu-cmpt-201/base
(d) docker create -it --name cmpt201 ghcr.io/sfu-cmpt-201/base
```

Reading a man page

Man Page

- Reading a man page
  - our primary way to learn functions/system calls for systems programming.
  - It takes practice to effectively read a man page!
- The command is

man <da-thing>

  - e.g., `man ls`, `man cd`
- Section Numbers

..Pick between two things of the same name.

  - Most relevant sections for CMPT 201:
  - man 1: General commands e.g., `man 1 ls`
  - man 2: System calls e.g., `man 2 fork`
  - man 3: C standard library functions e.g., `man 3 printf`

Learning a Function

- Problem
  - I know a syscall/function; how do I use it?
- Steps
  1) Is this what I want?
  2) How do I call it?
  3) What does it give me?
  4) How can it go wrong? (errno, feature test)

```
atoi(3)                Library Functions Manual                atoi(3)

NAME
       atoi, atol, atoll - convert a string to an integer

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <stdlib.h>

       int atoi(const char *nptr);
       long atol(const char *nptr);
       long long atoll(const char *nptr);

   Feature   Test   Macro   Requirements   for  glibc  (see  fea-
   ture_test_macros(7)):

       atoll():
           _ISOC99_SOURCE
               || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION
       The atoi() function converts the initial  portion  of  the
       string  pointed  to  by  nptr to int.  The behavior is the
       same as

           strtol(nptr, NULL, 10);

       except that atoi() does not detect errors.

       The atol()  and  atoll()  functions  behave  the  same  as
       atoi(),  except  that  they convert the initial portion of
       the string to their return type of long or long long.

RETURN VALUE
       The converted value or 0 on error.
```

Learning a Function

1) Is this what I want?
- Read Description section
- .. Skim fast for relevant part

(You'll need this skill!)

2) How do I call it?
- Read Synopsis (prototype)
- Check header files & return type
- Check arguments (in and out)

3) What does it give me?
- Read Return Value section
- Pay attention to output parameters (pointers)!


```
atoi(3)                Library Functions Manual                atoi(3)

NAME
       atoi, atol, atoll - convert a string to an integer

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <stdlib.h>

       int atoi(const char *nptr);
       long atol(const char *nptr);
       long long atoll(const char *nptr);

   Feature   Test   Macro   Requirements   for  glibc  (see  fea-
   ture_test_macros(7)):

       atoll():
           _ISOC99_SOURCE
               || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION
       The atoi() function converts the initial  portion  of  the
       string  pointed  to  by  nptr to int.  The behavior is the
       same as

           strtol(nptr, NULL, 10);

       except that atoi() does not detect errors.

       The atol()  and  atoll()  functions  behave  the  same  as
       atoi(),  except  that  they convert the initial portion of
       the string to their return type of long or long long.

RETURN VALUE
       The converted value or 0 on error.
```

Learning a Function

4) How can it go wrong?
(errno, feature test)

- What errors possible?

Read Errors (more later)

- Do you need to a feature test?

E.g., man 3 srand
must define _POSIX_C_SOURCE

```
Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    rand_r():
        Since glibc 2.24:
            _POSIX_C_SOURCE >= 199506L
        glibc 2.23 and earlier
            _POSIX_C_SOURCE
```

ERRORS

EFAULT Problem with copying information from user space.

EINTR The pause has been interrupted by a signal that was delivered to the thread (see **signal**(7)). The remaining sleep time has been written into *rem so that the thread can easily call **nanosleep**() again and continue with the pause.

EINVAL The value in the tv_nsec field was not in the range [0, 999999999] or tv_sec was negative.

atoi(3) Library Functions Manual atoi(3)

NAME

atoi, atol, atoll - convert a string to an integer

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

int atoi(const char *nptr);
long atol(const char *nptr);
long long atoll(const char *nptr);
```

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

```
atoll():
    _ISOC99_SOURCE
        || /* glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION

The **atoi**() function converts the initial portion of the string pointed to by nptr to int. The behavior is the same as

```
strtol(nptr, NULL, 10);
```

except that **atoi**() does not detect errors.

The **atol**() and **atoll**() functions behave the same as **atoi**(), except that they convert the initial portion of the string to their return type of long or long long.

RETURN VALUE

The converted value or 0 on error.

ABCD: Review C Pointers

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

int make_abs_get_product(int *pA, int *pB)
{
    *pA = abs(*pA);
    *pB = abs(*pB);
    return *pA * *pB;
}

int main()
{
    int w = -4;
    int h = 5;
    int area = make_abs_get_product(&w, &h);
    printf("%d x %d = %d\n", w, h, area);
}
```

- What does this output?

```
(a)-4 x 5 = -20
(b) 4 x 5 =  20
(c) 4 x 5 = -20
(d)-4 x 5 =  20
```

(Formatting cleaned up)

Review C Pointers

- Note the: char** x
  - x is a.. pointer-to-a-pointer.
  - Used for output parameters
- Use of **
  - Calling code passes in.. address of their pointer
  - Function sets where that pointer points.

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

bool find_first_digit(char* data, int n, char** ppdigit)
{
    for (int i = 0; i < n; i++) {
        if (isdigit(data[i])) {
            *ppdigit = &data[i];
            return true;
        }
    }
    return false;
}

int main()
{
    char* data = "I wa5 h3r3!\n";
    char* pfirst_digit = NULL;

    if (find_first_digit(data, strlen(data), &pfirst_digit)) {
        printf("Found digit: %c\n", *pfirst_digit);
    } else {
        printf("Found no digits.\n");
    }
}
```

Summary

- Processes are programs executing from memory (RAM)
  - Each process has its own Memory Space
- C Programming
  - Use man pages to lookup functions
  - Pointers and pointers-to-pointers used as output parameters
- Development Ideas
  - Use multiple terminal tabs/windows
  - Code a little at a time
- sleep() puts function to sleep