CS162
Operating Systems and Systems Programming
Lecture 4

Abstractions 2: Processes and Files and I/O
A quick programmer's viewpoint

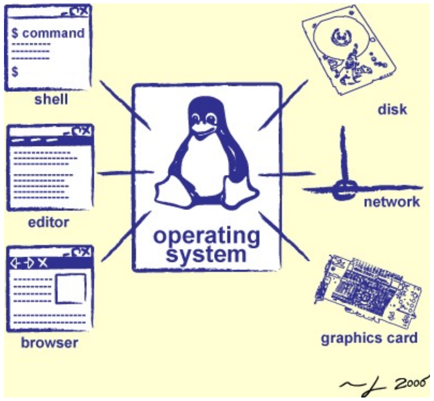
September 7th, 2021

Prof. Ion Stoica

http://cs162.eecs.Berkeley.edu

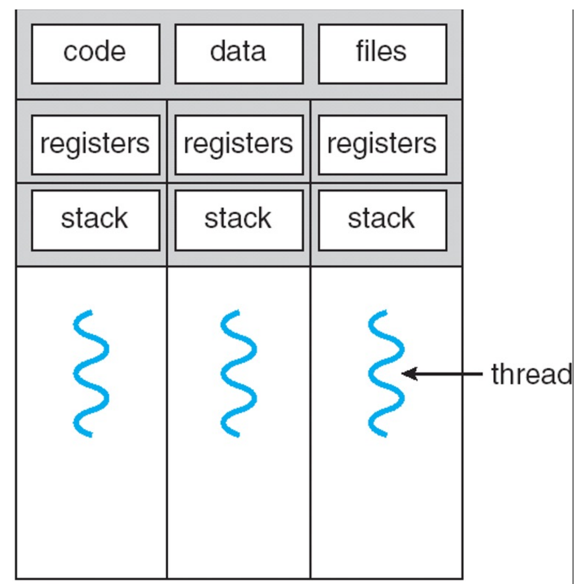
Goals for Today: The File Abstraction

- Finish discussion of process management
- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors



Thread State

- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)
- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack
- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing



multithreaded process

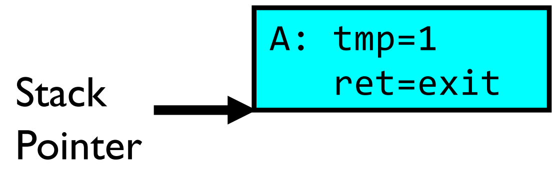
Execution Stack Example

```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

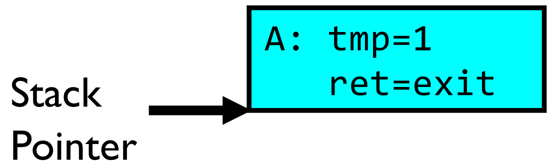
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

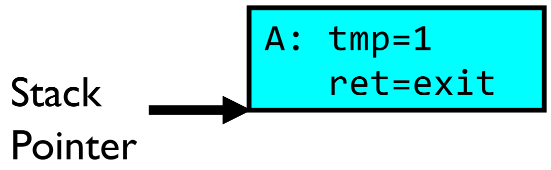
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

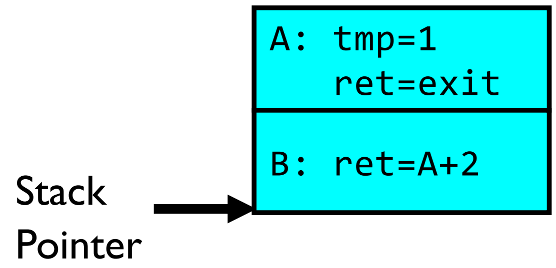
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

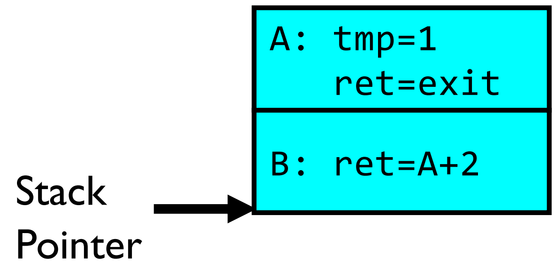
```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

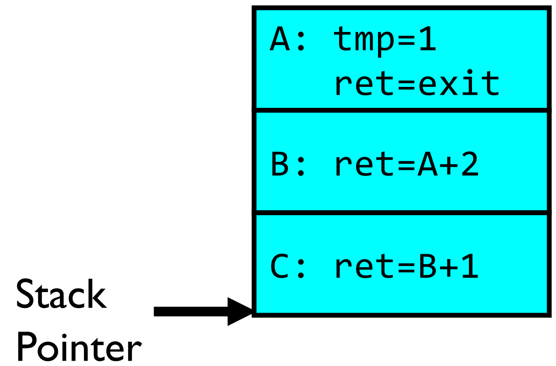
```
           A(int tmp) {
   A:        if (tmp<2)
 A+1:          B();
 A+2:        printf(tmp);
           }
           B() {
   B:        C();
 B+1:      }
           C() {
   C:        A(2);
 C+1:      }
           A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

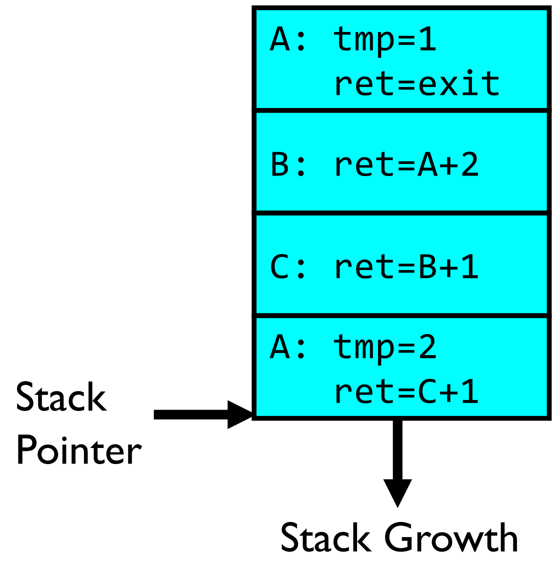
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

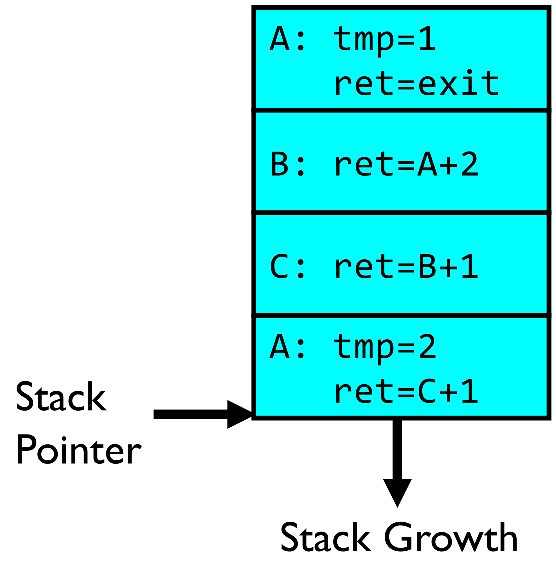
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```

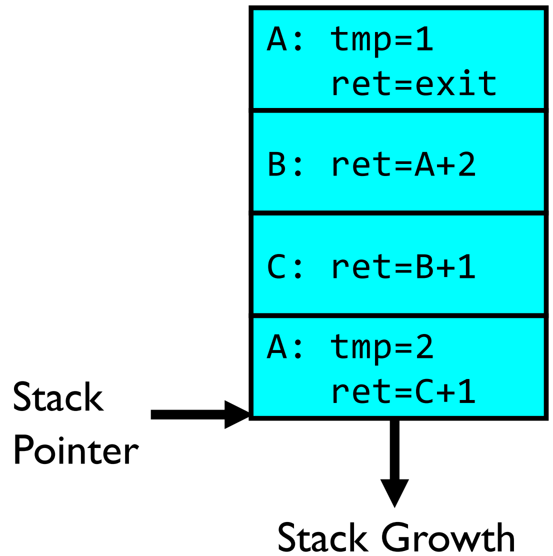


Output: >2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```

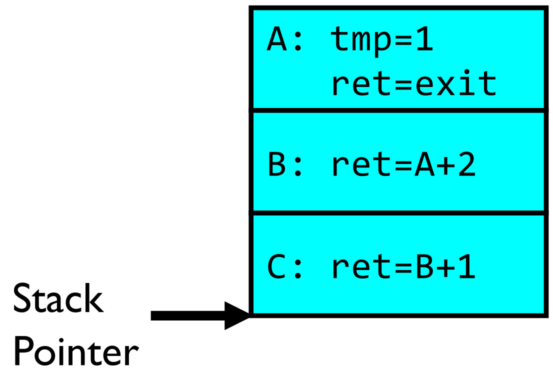


Output: >2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
       A(int tmp) {
A:       if (tmp<2)
A+1:       B();
A+2:     printf(tmp);
       }
       B() {
B:       C();
B+1:   }
       C() {
C:       A(2);
C+1:   }
       A(1);
exit:
```

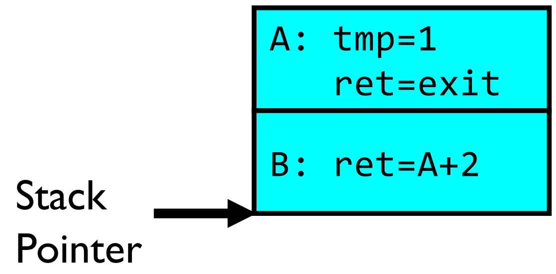


Output: >2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



Output: >2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
       A(int tmp) {
A:       if (tmp<2)
A+1:       B();
A+2:     printf(tmp);
       }
       B() {
B:       C();
B+1:   }
       C() {
C:       A(2);
C+1:   }
       A(1);
exit:
```

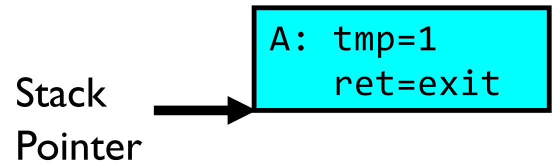
Stack Pointer → `A: tmp=1 ret=exit`

Output: `>2 1`

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



Output: >2 1

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

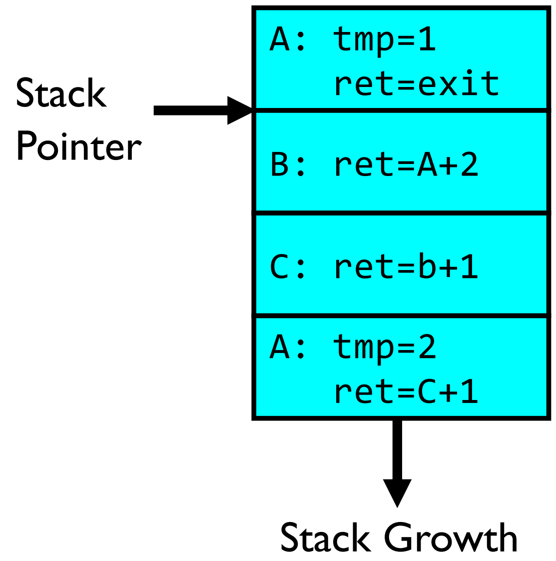
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```

Output: `>2 1`

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

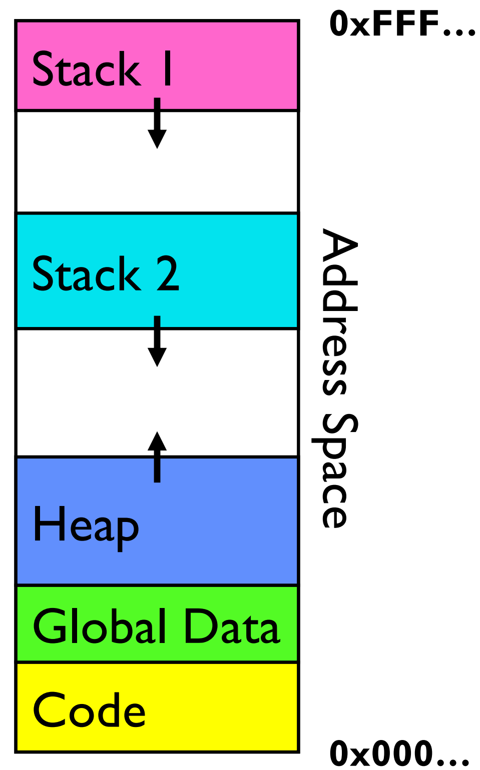
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

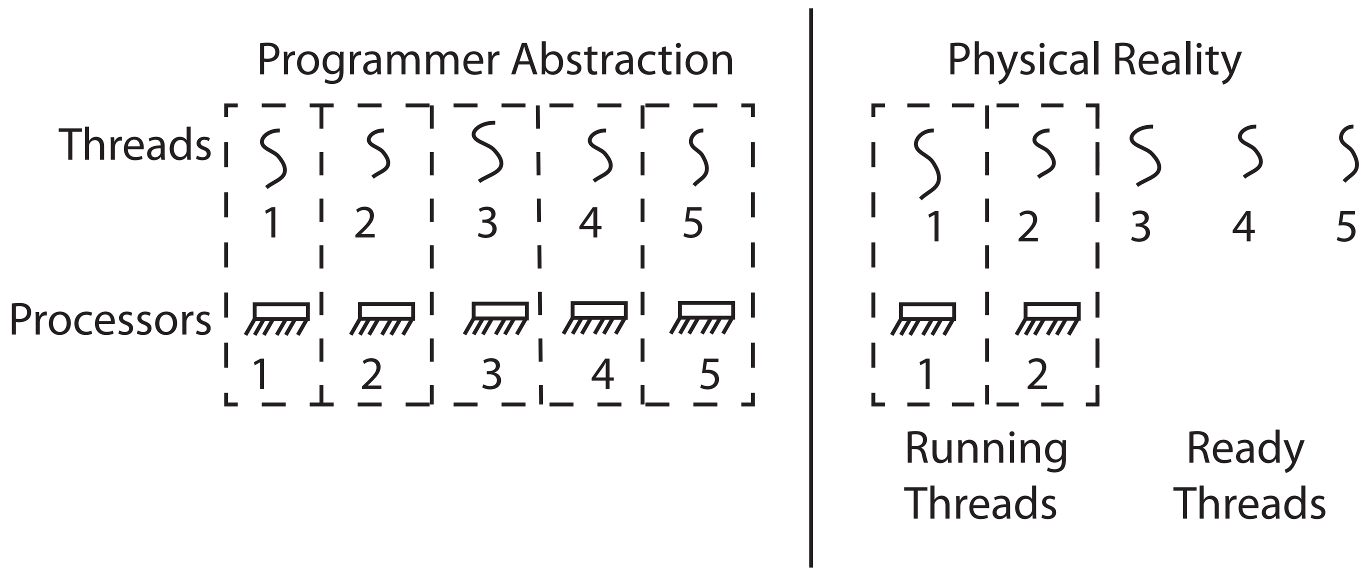
Memory Layout with Two Threads

- Two sets of CPU registers
- Two sets of Stacks
- Issues:
  – How do we position stacks relative to each other?
  – What maximum size should we choose for the stacks?
  – What happens if threads violate this?
  – How might you catch violations?



INTERLEAVING AND NONDETERMINISM
(The beginning of a long discussion!)

Thread Abstraction

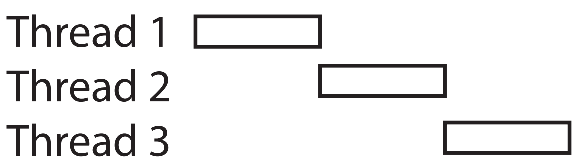


- Illusion: Infinite number of processors
- Reality: Threads execute with variable "speed"
  - Programs must be designed to work with any schedule

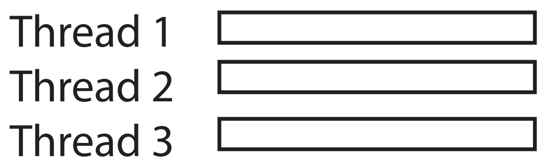
Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1	x = x + 1
y = y + x;	y = y + x;	....	y = y + x
z = x +5y;	z = x + 5y;	thread is suspended	....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	....	thread is resumed
		y = y + x	....
		z = x + 5y	z = x + 5y

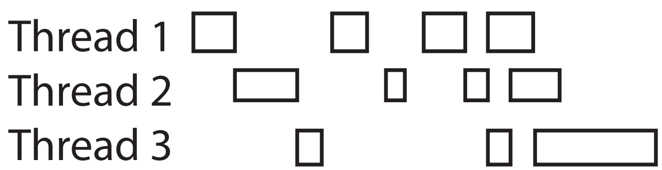
Possible Executions



a) One execution



b) Another execution



c) Another execution

Correctness with Concurrent Threads

- Non-determinism:
  - Scheduler can run threads in **any order**
  - Scheduler can switch threads **at any time**
  - This can make testing very difficult
- *Independent Threads*
  - No state shared with other threads
  - Deterministic, reproducible conditions
- *Cooperating Threads*
  - Shared state between multiple threads
- **Goal: Correctness by Design**

Race Conditions

- Initially `x == 0 and y == 0`

```
Thread A        Thread B
x = 1;          y = 2;
```

- What are the possible values of **x** below after all threads finish?
- Must be **1**. Thread B does not interfere

Race Conditions

- Initially `x == 0 and y == 0`

```
Thread A        Thread B
x = y + 1;      y = 2;
                y = y * 2;
```

- What are the possible values of **x** below?

- Race Condition: Thread A races against Thread B!

Relevant Definitions

- Synchronization: Coordination among threads, usually regarding shared data
- Mutual Exclusion: Ensuring only one thread does a particular thing at a time (one thread *excludes* the others)
  - Type of synchronization
- Critical Section: Code exactly one thread can execute at once
  - Result of mutual exclusion
- Lock: An object only one thread can hold at a time
  - Provides mutual exclusion

Locks

- Locks provide two **atomic** operations:
  - Lock.acquire() – wait until lock is free; then mark it as busy
    - » After this returns, we say the calling thread *holds* the lock
  - Lock.release() – mark lock as free
    - » Should only be called by a thread that currently holds the lock
    - » After this returns, the calling thread no longer holds the lock

- For now, don't worry about how to implement locks!
  - We'll cover that in substantial depth later on in the class

OS Library Locks: *pthreads*

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr)

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

You'll get a chance to use these in Homework 1

Our Example

Critical section

```
int common = 162;
pthread_mutex_t common_lock = PTHREAD_MUTEX_INITIALIZER;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  pthread_mutex_lock(&common_lock);
  int my_common = common++;
  pthread_mutex_unlock(&common_lock);

  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid,
         (unsigned long) &common, my_common);
  pthread_exit(NULL);
}
```

Semaphores: A quick look

- Semaphores are a kind of *generalized lock*
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX (& Pintos)
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - `P()` or `down()`: atomic operation that waits for semaphore to become positive, then decrements it by 1
  - `V()` or `up()`: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

`P()` stands for "*proberen*" (to test) and `V()` stands for "*verhogen*" (to increment) in Dutch

Two Semaphore Patterns

- Mutual Exclusion: (like lock)
  - Called a "binary semaphore" or "mutex"

```
initial value of semaphore = 1;
semaphore.down();
      // Critical section goes here
semaphore.up();
```

- Signaling other threads, e.g. **ThreadJoin**

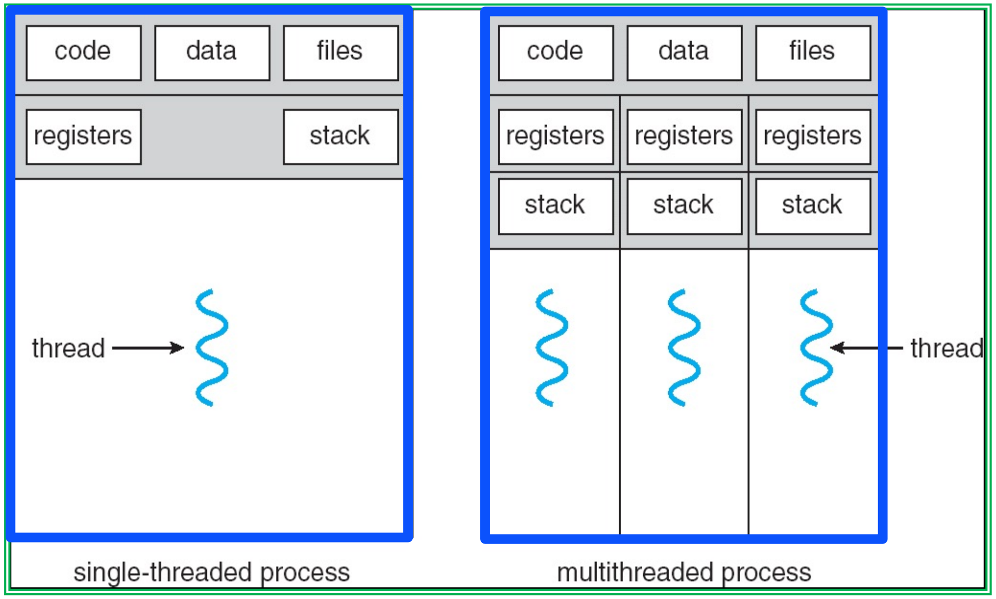
```
Initial value of semaphore = 0

ThreadJoin {
   semaphore.down();
}
```

```
ThreadFinish {
    semaphore.up();
}
```

Processes

- Definition: execution environment with restricted rights
  - One or more threads executing in a single address space
  - Owns file descriptors, network connections
- Instance of a running program
  - When you run an executable, it runs in its own process
  - Application: one or more processes working together
- Protected from each other; OS protected from them
- In modern OSes, anything that runs outside of the kernel runs in a process



Creating Processes

- `pid_t fork()` – copy the current process
  - New process has different pid
  - New process contains a single thread
- Return value from **`fork()`**: pid (like an integer)
  - When > 0:
    - » Running in (original) Parent process
    - » return value is pid of new child
  - When = 0:
    - » Running in new Child process
  - When < 0:
    - » Error! Must handle somehow
    - » Running in original process
- State of original process duplicated in *both* Parent and Child!
  - Address Space (Memory), File Descriptors (covered later), etc…

fork_race.c

```c
int i;
pid_t cpid = fork();
if (cpid > 0) {
  for (i = 0; i < 10; i++) {
    printf("Parent: %d\n", i);
    // sleep(1);
  }
} else if (cpid == 0) {
  for (i = 0; i > -10; i--) {
    printf("Child: %d\n", i);
    // sleep(1);
  }
} else { /* ERROR! */ }
```

Parent Process Runs HERE!

Child Process Runs HERE!

- What does this print?
- Would adding the calls to `sleep()` matter?

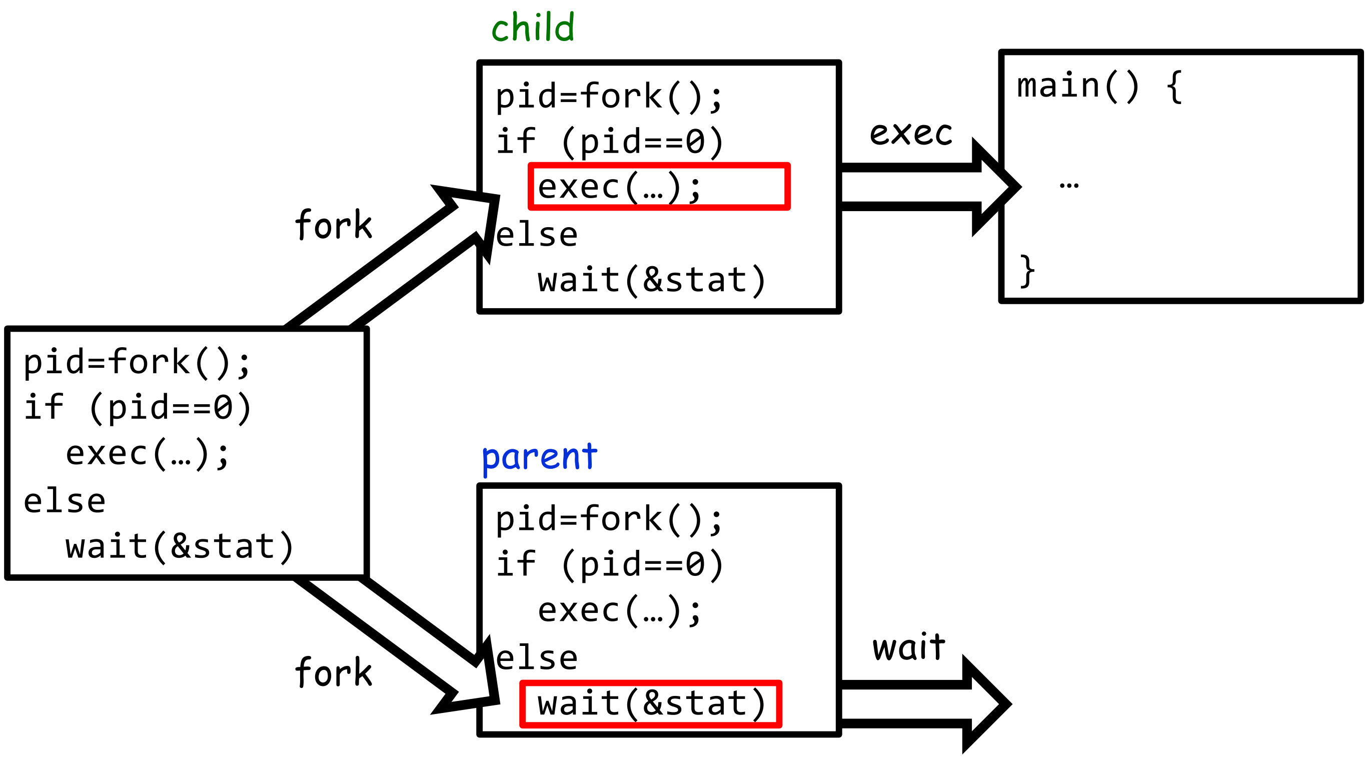
Start new Program with exec

```
…
cpid = fork();
if (cpid > 0) {                /* Parent Process */
  tcpid = wait(&status);
} else if (cpid == 0) {       /* Child Process */
  char *args[] = {“ls”, “-l”, NULL};
  execv(“/bin/ls”, args);

  /* execv doesn’t return when it works.
     So, if we got here, it failed! */

  perror(“execv”);
  exit(1);
}
…
```

Starting New Program (for instance in Shell)



Finishing up: Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

fork2.c – parent waits for child to finish

```
int status;
pid_t tcpid;
…
cpid = fork();
if (cpid > 0) {                  /* Parent Process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
} else if (cpid == 0) {      /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
  exit(42);
}
…
```

Finishing up: Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

inf_loop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
  printf("Caught signal!\n");
  exit(1);
}
int main() {
  struct sigaction sa;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = signal_callback_handler;
  sigaction(SIGINT, &sa, NULL);
  while (1) {}
}
```

Q: What would happen if the process receives a SIGINT signal, but does not register a signal handler?
A: The process dies!

For each signal, there is a default handler defined by the system

Common POSIX Signals

- `SIGINT` – control-C
- `SIGTERM` – default for **`kill`** shell command
- `SIGSTP` – control-Z (default action: stop process)

- `SIGKILL`, `SIGSTOP` – terminate/stop process
  - Can't be changed with **`sigaction`**
  - Why?

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task

- You will build your own shell in Homework 2…
  - … using `fork` and `exec` system calls to create new processes…
  - … and the File I/O system calls we'll see next to link them together

Process vs. Thread APIs

- Why have `fork()` and `exec()` system calls for processes, but just a `pthread_create()` function for threads?
  - Convenient to `fork` without `exec`: put code for parent and child in one executable instead of multiple
  - It will allow us to programmatically control child process' state
    - » By executing code before calling `exec()` in the child
  - We'll see this in the case of File I/O later
- Windows uses `CreateProcess()` instead of `fork()`
  - Also works, but a more complicated interface

Threads vs. Processes

- If we have two tasks to run concurrently, do we run them in separate threads, or do we run them in separate processes?

- Depends on how much isolation we want
  – Threads are lighter weight [why?]
  – Processes are more strongly isolated

Administrivia

- Project 0 due Thursday (9/9)!
  - To be done on your own – like a homework!
- Group assignments will be released by Wednesday, EOD
- Discussion section attendance is mandatory (with cameras on if remote).
- Start Planning on how your group will collaborate on projects!
  - Virtual Coffee Hours with your group (with camera)
  - Regular Brainstorming meetings?
  - Try to meet multiple times a week

The File Abstraction

- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors
- *How* and *Why* of High-Level File I/O
- Process State for File Descriptors
- Common Pitfalls with OS Abstractions [if time]

} Next lecture

Unix/POSIX Idea: Everything is a "File"

- Identical interface for:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)
- Based on the system calls **`open()`**, **`read()`**, **`write()`**, and **`close()`**
- Additional: **`ioctl()`** for custom configuration that doesn't quite fit
- Note that the "Everything is a File" idea was a radical idea when proposed
  - Dennis Ritchie and Ken Thompson described this idea in their seminal paper on UNIX called "The UNIX Time-Sharing System" from 1974
  - I posted this on the resources page if you are curious

Note: What does POSIX stand for?

- POSIX: Portable Operating System Interface (for uniX?)
  - Interface for application programmers (mostly)
  - Defines the term "Unix," derived from AT&T Unix
  - Created to bring order to many Unix-derived OSes, so applications are portable
    - » Partially available on non-Unix OSes, like Windows
  - Requires standard system call interface

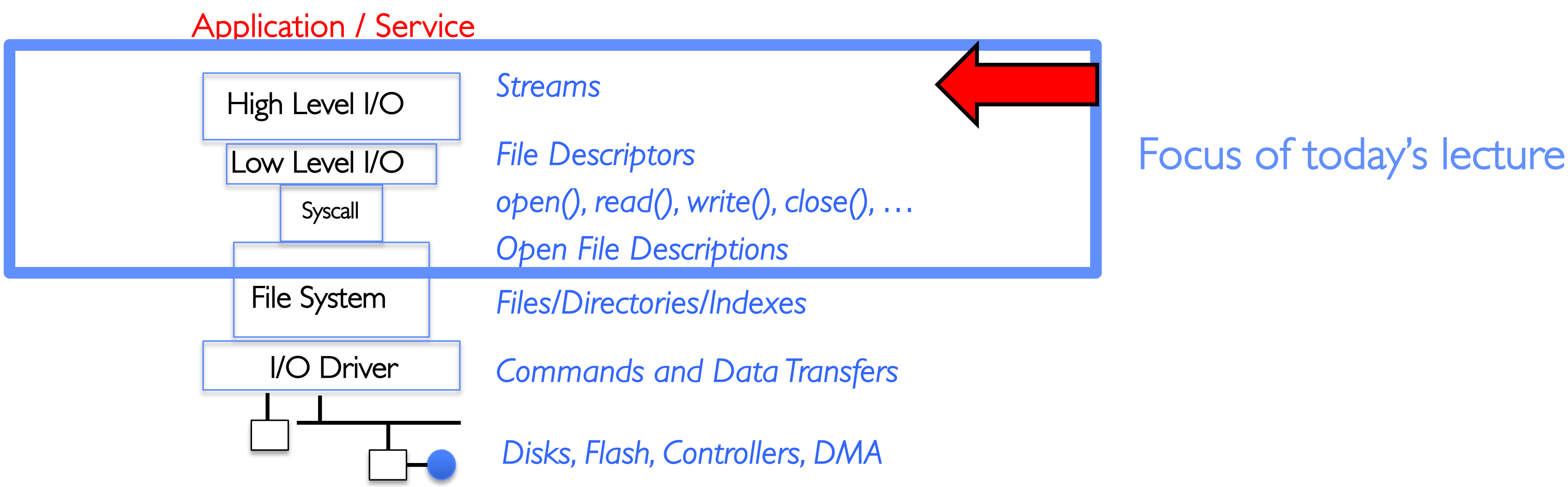
The File System Abstraction

- File
  - Named collection of data in a file system
  - POSIX File data: sequence of bytes
    - » Could be text, binary, serialized objects, …
  - File Metadata: information about the file
    - » Size, Modification Time, Owner, Security info, Access control
- Directory
  - "Folder" containing files & directories
  - Hierachical (graphical) naming
    - » Path through the directory graph
    - » Uniquely identifies a file or directory
      - /home/ff/cs162/public_html/fa14/index.html
  - Links and Volumes (later)

Connecting Processes, File Systems, and Users

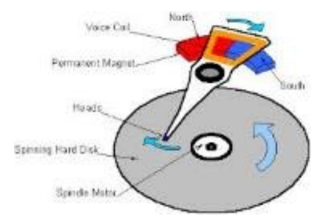
- Every process has a *current working directory* (CWD)
  - Can be set with system call:
    `int chdir(const char *path); //change CWD`
- Absolute paths ignore CWD
  - /home/john/cs162
- Relative paths are relative to CWD
  - index.html, ./index.html
    - » Refers to index.html in current working directory
  - ../index.html
    - » Refers to index.html in parent of current working directory
  - ~/index.html, ~cs162/index.html
    - » Refers to index.html in the home directory

I/O and Storage Layers










C High-Level File API – Streams

- Operates on "streams" – unformatted sequences of bytes (wither text or binary data), with a position:

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

Mode Text	Binary	Descriptions
r	rb	Open existing file for reading
w	wb	Open for writing; created if does not exist
a	ab	Open for appending; created if does not exist
r+	rb+	Open existing file for reading & writing.
w+	wb+	Open for reading & writing; truncated to zero if exists, create otherwise
a+	ab+	Open for reading & writing. Created if does not exist. Read from beginning, write as append

- Open stream represented by pointer to a FILE data structure
  - Error reported by returning a NULL pointer

C API Standard Streams – `stdio.h`

- Three predefined streams are opened implicitly when the program is executed.
  - `FILE *stdin` – normal source of input, can be redirected
  - `FILE *stdout` – normal source of output, can too
  - `FILE *stderr` – diagnostics and errors
- STDIN / STDOUT enable composition in Unix
- All can be redirected
  - `cat hello.txt | grep "World!"`
  - **`cat`**'s **`stdout`** goes to **`grep`**'s **`stdin`**

C High-Level File API

```
// character oriented
int fputc( int c, FILE *fp );              // rtn c or EOF on err
int fputs( const char *s, FILE *fp );      // rtn > 0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ... );
```

C Streams: Char-by-Char I/O

```
int main(void) {
  FILE* input = fopen("input.txt", "r");
  FILE* output = fopen("output.txt", "w");
  int c;

  c = fgetc(input);
  while (c != EOF) {
    fputc(output, c);
    c = fgetc(input);
  }
  fclose(input);
  fclose(output);
}
```

C High-Level File API

```
// character oriented
int fputc( int c, FILE *fp );       // rtn c or EOF on err
int fputs( const char *s, FILE *fp );       // rtn > 0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ... );
```

C Streams: Block-by-Block I/O

```
#define BUFFER_SIZE 1024
int main(void) {
  FILE* input = fopen("input.txt", "r");
  FILE* output = fopen("output.txt", "w");
  char buffer[BUFFER_SIZE];
  size_t length;
  length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  while (length > 0) {
    fwrite(buffer, length, sizeof(char), output);
    length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  }
  fclose(input);
  fclose(output);
}
```

Aside: System Programming

- Systems programmers should always be paranoid!
  – Otherwise you get intermittently buggy code
- We should really be writing things like:

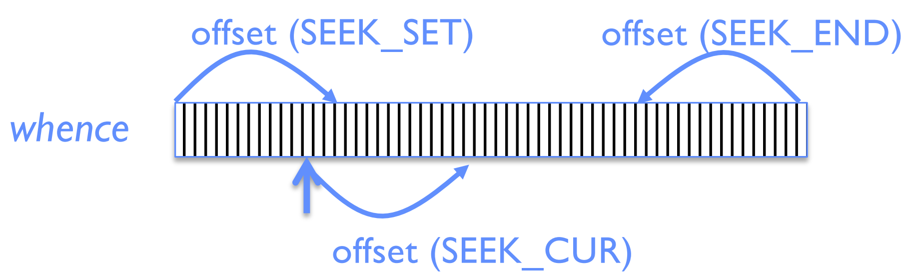
```
FILE* input = fopen(“input.txt”, “r”);
if (input == NULL) {
  // Prints our string and error msg.
  perror(“Failed to open input file”)
}
```

- Be **thorough about checking return values!**
  – Want failures to be systematically caught and dealt with
- I may be a bit loose with error checking for examples in class (to keep short)
  – Do as I say, not as I show in class!

C High-Level File API: Positioning The Pointer

```
int fseek(FILE *stream, long int offset, int whence);
long int ftell (FILE *stream)
void rewind (FILE *stream)
```

- For `fseek()`, the `offset` is interpreted based on the `whence` argument (constants in `stdio.h`):
  - `SEEK_SET`: Then offset interpreted from beginning (position 0)
  - `SEEK_END`: Then offset interpreted backwards from end of file
  - `SEEK_CUR`: Then offset interpreted from current position



- Overall preserves high-level abstraction of a uniform stream of objects

Conclusion

- Threads are the OS unit of concurrency
  - Abstraction of a virtual CPU core
  - Can use pthread_create, etc., to manage threads within a process
  - They share data → need synchronization to avoid data races
- Processes consist of one or more threads in an address space
  - Abstraction of the machine: execution environment for a program
  - Can use fork, exec, etc. to manage threads within a process
- POSIX idea: "everything is a file"