Announcements

• Homework 0 is available
  – Due on September 8th
  – All you need to do is implement "Hello World"; the real goal is to try out the VM and the submission process
  – If you encounter any problems, please let one of us know right away!

• How is the VM image working for you?

• Homework 1 will be available soon
  – Handout will be released later this week – probably on Thursday
Plan for the next two lectures

• We are going to start by reviewing some of your undergraduate material:
  – Processes
  – Threads
  – System calls
  – Kernel basics

• The idea is to get everyone on the same page

• If you've seen this material recently and still remember all of it, please bear with me!
Plan for today

• Processes
  – From cradle to grave: The life and times of a process
  – Inter-process communication
  – Context switching

• Threads
  – Threads vs. processes
  – Thread pools
  – User-level threads
What is a process?

• A “program in execution”
  – with associated (data and execution) context
  – Process execution must progress in sequential fashion

• Not the same as “program” or “application”
  – a given program may be running 0, 1, or >1 times

• Each instance of a program is a separate process
  – with its own address space and other context
Why run processes?

• An operating system executes a variety of programs:
  – Batch system – jobs
  – Time-shared systems – user programs or tasks

• Many processes can run concurrently:
  – Singer-user system: several programs (word processor, browser, email) running at the same time
  – OS internal programmed activities, such as memory management
What is in a process?

- Processes live in **address spaces**, which contain:
  - **Code**: The code that your program executes
    - **program counter (PC)** points to the next instruction to execute
    - Includes libraries that the program is linked against
  - **Data**: Any static data, such as global variables
  - **Heap**: Dynamically allocated memory
    - malloc/free, new/delete
  - **Stack**: Temporary data
    - Function arguments, return addresses, local variables, ...
    - Grows and shrinks dynamically;
      stack pointer (SP) points to current 'top'

- The actual locations can vary
  - Even between runs of the same program!
  - Example: Address-space layout randomization (ASLR)
A closer look

Try it out in your VM!

- /proc/[processID]/maps shows you the memory map of a process
- You can find the process ID, e.g., with 'ps fax'
The process control block

• To keep track of running processes, the kernel maintains a process table
  – This table contains a process control block for each process
  – Each process has a process ID, which can be thought of (or, in some cases, is actually) as an index into this table

• The process control block contains:
  – The state of the process
  – Information about any child processes
  – Pending signals and messages
  – Any special privileges the process has
  – The owner of the process (user ID)
  – Processor state (when the process is not running)
  – Accounting and scheduling information
  – (and some other things)

see the following slides
Process states

- **new**: The process is being created
- **ready**: The process is waiting to be assigned to a processor
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **terminated**: The process has finished execution
Parents and children; PIDs

• Where do processes come from?
  – A Unix process comes into being when another process creates it
  – The new process is the 'child'; the existing process is the 'parent'
  – Every process has exactly one parent

• Then where does the first process come from?
  – The first process (init/systemd) is created by the kernel during boot

• Processes have a unique process ID (PID) number
  – Internally, the OS has a table of process contexts
  – The PID is simply an index into that table
Example: Unix process hierarchy

- This tree shows (some of) the processes in your VM
  - To view, run `ps -ejh | less` in a terminal
Creating a process: fork()

• The fork() system call creates a new process
  – The child process is an (almost) exact clone of the parent (with its own copy of the parent’s address space)
  – The child starts running as soon as fork() returns
  – Both the child and the parent now running simultaneously
  – Parent and child share the same resources (open files, etc.)

• What good is this?
  – Write code to behave differently if you’re the child!

• How can you tell if you’re the child or the parent?
  – for the child, fork() return value is 0
  – for the parent, fork() return value is the PID of child
Replacing a process: exec()

• The exec() system call replaces the program that a process is running
  – exec() doesn’t create any new processes!
  – The new program is specified by the name of the file containing its executable code
  – If there are no errors, exec() never returns – the old code stops running immediately

• What good is this?
  – usually run after fork()

• How do parameters get passed to the program?
  – There are several variants of exec() that can do this
  – Example: execv() can pass arguments, execve() can additionally pass environment variables
Creating a child process

- Typically uses both fork() and exec()
  - fork() creates a new process, which then immediately calls exec()
  - The above example uses execv() to tell the 'ls' process which directory to list

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

int main(void)
{
    pid_t pid;
    char *const parmList[] = {"/bin/ls", "-l", "/home/cis505", NULL};
    if ((pid = fork()) == -1)
        perror("fork error");
    else if (pid == 0)
    {
        execv("/bin/ls", parmList);
        perror("execv error");
    }
}
```
Terminating a process: exit()

• If a process wants to terminate, it can call exit()
  – Effect: Process is terminated immediately
  – Returning from main() implicitly calls exit()

• exit() takes a status code as its argument
  – This code becomes available to the parent process (see next slide)
  – Often used to indicate whether the process succeeded, or whether there was some problem

• The status code can be read by the parent

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

int main(void)
{
    printf("Exiting with code 3\n");
    exit(3);
}
```

cis505@vm:~$ g++ foo.c -o foo
 cis505@vm:~$ ./foo
 Exiting with code 3
 cis505@vm:~$ echo $?
 3
 cis505@vm:~$
Waiting for termination: wait()

• How can the parent tell whether a child has exited?
  – There is a special system call for this: wait()

• wait() blocks until a child process terminates
  – It returns the PID of the child that terminated, as well as the exit code that child gave to exit()
  – Syntax: wait(&status)

• What if a process has multiple children?
  – There is a variant called waitpid() that can wait for a specific child (identified by its PID)
  – waitpid() also takes options, e.g., to control whether the call should block until the child exits, or simply check and then return right away
  – Syntax: waitpid(pid, &status, options)
Zombie processes

• Once a program is terminated, the kernel releases any resources it has acquired
  – Example: Allocated memory is freed, open files are closed, ...

• However, the process control block is not released immediately
  – Why not?
  – The PCB still contains information that the parent process may be interested in, e.g., the exit code
  – During this time, the process is called a 'zombie'

• The PCB is freed once the parent calls wait()
Example: The Unix shell

- What happens when you enter a shell command?
  - Step 1: The shell calls `fork()` to create a child process
  - Step 2: The child process calls `exec()`, which causes it to stop executing the shell program and start executing your command
  - Step 3: The parent (the shell process) calls `wait()` to wait for the child
  - Step 4: The child process terminates and calls `exit()`
  - Step 5: The `wait()` call in the parent returns
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    pid_t pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork failed!
"), exit(1);
    } else if (pid == 0) {
        execlp("/bin/ls", "ls", NULL);
        /* Never reached, unless execlp() fails */
    } else {
        int status;
        wait(&status);
        printf("Child returned (%d)\n", status);
    }
    return 0;
}
Plan for today

• Processes
  – From cradle to grave: The life and times of a process
  – Inter-process communication
  – Context switching

• Threads
  – Threads vs. processes
  – Thread pools
  – User-level threads
Why interprocess communication?

- Suppose a process forks a child to do some work
  - How does the work get back to the parent?

- Idea: Shared variables
  - What will happen in the program on the right? Why?

- Problem: Child's address space is a duplicate of the parent's
  - If the child changes something in its copy, the parent won't notice (and vice versa)
  - This is a good thing! (Address spaces are protection domains!)

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

int main(void)
{
    int a = 47, b = 11;
    int c = 0;
    if (fork() == 0) {
        /* Do some difficult work */
        c = a + b;
        exit(0);
    } else {
        wait(NULL);
        printf("Result: %d\n", c);
    }
    return 0;
}
```
Interprocess communication

• Processes communicate for a number of reasons
  – Information sharing, computation speedup, modularity, ...

• The kernel offers several ways to do this, e.g.:
  – Pipes
  – Signals
  – Message queues
  – Shared files
  – Shared memory

• Why so many different primitives?
  – Each has different strengths and weaknesses
  – We will see a few of them in the next couple of lectures
Pipes

• A **pipe** is a one-way FIFO channel
  – Pipes can be created by the `pipe()` system call
  – A pipe has two ends, a 'write end' and a 'read end'
  – If data is written to the write end, it becomes available for reading on the read end

• How can we use this for IPC?
  – Parent process can create a pipe before `fork()`
  – Child will inherit both ends of the pipe
  – Then the parent can close one end, and the child can close the other end
  – Result: Synchronous one-way communication channel

• What if we want two-way communication?
Example: Pipes

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

int main(void)
{
    int myPipe[2];
    pipe(myPipe);
    pid_t pid = fork();
    if (pid < 0){
        fprintf(stderr, "Fork failed!\n");
        exit(1);
    }
    else if (pid == 0){
        /* Child process: writing to the pipe */
        close(myPipe[0]);
        const char *message = "Hello world!";
        write(myPipe[1], message, strlen(message)+1);
        printf("Child process sent: %s\n", message);
        exit(0);
    } else {
        /* Parent process: reading from the pipe */
        close(myPipe[1]);
        char message[200];
        read(myPipe[0], message, sizeof(message));
        printf("Parent process received: %s\n", message);
        wait(NULL);
    }
    return 0;
}
```

Create a new pipe
Child process will inherit both ends of the pipe
The read end is not used, so close it
Write message to the write end
Close the write end
Read message from the read end
Signals

• What if we need asynchronous communication?
  – I.e., the process needs to be told about something it isn't necessarily expecting, or waiting for
  – Example: User hits Ctrl-C

• We can use signals for this
  – Some common signals: SIGTERM, SIGINT, SIGHUP, SIGSEGV, SIGSTOP/SIGCONT

• The process can define a handler for each signal
  – Essentially a function that gets called when the signal arrives
  – Example: run a shutdown handler to 'clean up' some state
  – Each signal has a default action that is taken when no handler is specified (typically either 1) do nothing or 2) terminate the process)

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

FILE *infile;

void shutdown(int arg) {
  if (infile)
    fclose(infile);
  exit(1);
}

int main(void) {
  signal(SIGINT, shutdown);
  infile = fopen("foo.txt", "r");
  while (fgets(buf, len, infile)) {
    ...
  }
  return 0;
}
```
Special signals

• What if a process doesn't terminate on its own?
  – Example: Bug (endless loop?), bad input, malicious program

• We can use signals to terminate it!
  – Processes can send signals to each other!
  – Typical approach: Use the kill(pid,signal) function to send SIGTERM, whose default behavior is to terminate the program
  – This is what the 'kill' shell command does by default

• What if the program catches SIGTERM?
  – That handler could itself contain an endless loop!
  – Does this mean that we can never get rid of such a program?
  – No! There is another signal (SIGKILL) that doesn't allow handlers
  – This is what the 'kill -9' shell command does
Restrictions on signals

• Without further restrictions, signal() is dangerous!
  – Users could (accidentally or intentionally) send signals to each other's processes

• Solution: Restrict who can send signals to whom
  – What restrictions would make sense?
  – Each process has a 'user ID' that identifies the user who started it
  – Children inherit their parent's user ID
    • But they can change it with proper authorization (e.g., the 'sudo' command)
  – Sender's user ID must match that of the recipient

• Special exception: Super-user can send to anyone
  – Why is this necessary?
Message queues

• Signals and pipes are simple but limited
  – What if we want to send richer messages, or out-of-order data?

• Message queues offer a richer API
  – Identified by a name (e.g., "/myqueue")
  – mq_open: Opens or creates a queue
  – mq_send: Add a new message to the queue
  – mq_receive: Get a message from the queue
  – mq_close: Closes the queue
  – mq_unlink: Destroys the queue

• Messages have an associated priority
  – The highest priority message is always delivered first!
Example: Message queues

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

int main(void)
{
    if (fork() != 0) {
        /* Parent: Receive the message */
        struct mq_attr attr;
        attr.mq_flags = 0; attr.mq_maxmsg = 10;
        attr.mq_msgsize = 100; attr.mq_curmsgs = 0;
        mqd_t queue = mq_open("/myqueue", O_RDONLY|O_CREAT, 0644, &attr);
        char message[200];
        mq_receive(queue, message, sizeof(message), NULL);
        printf("Received: %s\n", message);
        mq_close(queue);
    } else {
        /* Child: Send the message */
        sleep(1);
        mqd_t queue = mq_open("/myqueue", O_WRONLY);
        const char *message = "Hello world!";
        mq_send(queue, message, strlen(message)+1, 1);
        mq_close(queue);
    }

    return 0;
}
```
Plan for today

• **Processes**
  - From cradle to grave: The life and times of a process
  - Inter-process communication
  - Context switching

• **Threads**
  - Threads vs. processes
  - Thread pools
  - User-level threads
Why context switching?

• If a node has multiple processes, which one should be run, and when?
  – Idea: Let each process run until it finishes ("batch processing")
  – Is this a good idea? What would this mean for user experience?

• Better: Interleave the processes somehow
  – Each process gets to run for a bit ("time quantum") and then the kernel stops it and switches to another process, etc.
Preemption and scheduling

• Who does the switching?
  – Cooperative multitasking: The processes themselves yield the CPU periodically
    • Used in older versions of Windows (pre-WIn95) and older versions of OS X
  – Preemptive multitasking: The kernel preempts processes periodically
    • Used in most of the newer desktop OSes
  – Pros and cons?

• When a switch happens, who gets to run next?
  – Many possible choices!
  – Examples: Round-robin, priority-based, EDF, ...
  – Pros and cons?
The mechanics of context switching

• What does the kernel have to do to switch from one process to another?
  – Save the state of the old process (CPU registers, program counter, stack pointer, ...) in the PCB of the old process
  – Switch to the address space of the new process
  – Load the state from the PCB of the new process

• This is (somewhat) expensive!
  – Depending on the hardware, certain caches (TLB!) have to be flushed - and there are other overheads
  – Pure overhead: The node doesn't do any 'useful' work during a switch
Recap: Processes

- **Process**: Address space + thread of execution
  - Kernel stores state in a process control block

- **Key functions**: fork(), exec(), exit(), wait()
  - Forking creates a 'child' process that is a clone of the parent

- **Processes can communicate via IPC**
  - Several methods (e.g., signals, pipes, message queues) that each have their pros and cons

- **Kernel can switch between processes**
  - Store old context, switch address space, load new context
Plan for today

• Processes
  – From cradle to grave: The life and times of a process
  – Inter-process communication
  – Context switching

• Threads
  – Threads vs. processes
  – Thread pools
  – User-level threads
Why threads?

• A process has two key elements
  – A collection of resources (address space, open files, etc.)
  – A thread of execution (program counter, registers, stack pointer, etc.)

• Having multiple threads of execution is often useful
  – Example: Keeping multiple CPU cores busy by working in parallel
  – Example: Handling lots of concurrent activities (e.g., many clients)

• But having separate resources can be a headache!
  – Communication has to be via IPC
  – Potentially lots of redundancy (every thread has its own AS, etc.)

• Idea: Let's have processes with multiple threads!
Threads vs. processes

- Threads share the same address space
  - ... but they have separate stacks, and separate contexts (SP, PC, ...)
  - So a process could simply be seen as 1 thread + 1 address space!
Threads vs. processes

• Some of the advantages of threads:
  – Creating a thread is cheaper than creating an entire process
  – Context switching within a process is much (!) cheaper
  – Threads within the same process can easily and cheaply communicate by accessing the same memory

• Some of the disadvantages of threads:
  – Same protection domain: If one thread runs amok (e.g., due to a bug), the entire process can crash
  – More opportunities for interesting bugs
    • E.g., unsynchronized access to shared data structures (more about this later!)
Thread context switching

• Recall: Process context switching:
  – Save the state of the old process (CPU registers, program counter, stack pointer, ...) in the PCB of the old process
  – Switch to the address space of the new process
  – Load the state from the PCB of the new process

• Thread context switches are (almost) the same
  – ... except that there is no need to switch to a new address space!
  – Register state is still saved to one PCB and loaded from another

• Result: Much lower overhead!
Plan for today

• Processes
  – From cradle to grave: The life and times of a process
  – Inter-process communication
  – Context switching

• Threads
  – Threads vs. processes
  – Thread pools
  – User-level threads
Why thread pools?

• Suppose you have a server that runs the pseudocode on the right
  – Examples: Web server; HW2 servers

• What if there is only one thread?
  – Suppose my grandmother connects to the server with her 1,200bps modem and downloads a Windows 10 install image (~3GB)
  – What will happen to the other clients?

• Solution: Multiple threads!
  – If some of the threads have to block, the other threads can still get work done
  – Result: I/O latencies are hidden

```
openServerSocket()
while (true) {
    conn=acceptConnection()
    readRequest(conn);
    sendResponse(conn);
    close(conn);
}
```
How thread pools work

• One dispatcher thread, many worker threads
  – Dispatcher accepts new connections and puts them into a queue
  – Workers get connections from the queue and handle them

• What if there are...
  – ... more connections than workers?
  – ... more workers than connections?

• How many threads should be in the pool?
POSIX threads

• Several functions for working with threads:
  – `pthread_create`: Starts a new thread
    • Can specify a function to run, and a single argument (what if you have more?)
  – `pthread_exit`: Terminates the current thread
    • Similar to `exit()` for processes
  – `pthread_join`: Blocks the caller until specified thread terminates
    • Similar to `wait()` for processes
  – `pthread_cancel`: Terminates another thread
    • Similar to `kill()` for processes
  – `pthread_self()`: Returns the ID of the current thread

• To compile a program with threads, most platforms require a special library (-lpthread)
Example: POSIX threads

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

void *threadFunc(void *arg)
{
    printf("This is worker #%d\n", *(int*)arg);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t threads[10];
    int state[10];
    for (int i=0; i<10; i++) {
        state[i] = i+1;
        pthread_create(&threads[i], NULL, threadFunc, &state[i]);
    }

    void *status;
    for (int i=0; i<10; i++)
        pthread_join(threads[i], &status);
}
```
Plan for today

• Processes
  – From cradle to grave: The life and times of a process
  – Inter-process communication
  – Context switching

• Threads
  – Threads vs. processes
  – Thread pools
  – User-level threads
User-level threads

• So far, we have discussed kernel threads
  – Each thread had its own process control block, ID, resources, ...
  – The threads just happen to live in the same address space

• But this is not the only way to implement threads!
  – The 'context switch' for threads doesn't have to involve any privileged operations – just saving and loading registers!
  – We can implement a small library that does this
  – Most of the other functionality can be implemented at user level
    • Example: Periodic timer can be used to preempt threads & context-switch

• Result: User-level threads
  – From the kernel's perspective, there "are no threads" – it's just a normal process with one 'real' thread
User-level vs. kernel-level threads

- **Some advantages** of user-level threads:
  - Creating a thread is cheap: Only requires allocating some memory for a stack and a thread control block
  - Context switching is very fast (just a few CPU instructions)
  - Process can pick its own scheduling algorithm
- **Some disadvantages** of user-level threads:
  - If one thread blocks, all the other threads are blocked too!
  - Signal handling is somewhat complicated
Recap: Threads

• Multiple threads can share a single address space
  – Result: Communication is cheap and easy
  – Lower overhead, but also less protection

• Example use case: Thread pools
  – One way for a server to handle multiple clients concurrently
  – Increases utilization: If one thread is blocked, others can still work

• Can be implemented at user level or in the kernel
  – User level is cheaper, but has some important limitations