Today's plan

- project 1 (review)
- scripting languages
- processes and threads
- process table
- schedulers
- interrupts and context switch
- layered operating system
- seL4

Project 1

- create a simple shell that can:
  - execute commands with arbitrary arguments (but no need for quoting, etc)
  - support input and output redirection to files
  - support input and output redirection to pipes
  - allow programs to run in the background (which is actually easier than waiting for programs to terminate)
- it is my hope that if you write your own shell,
  - you will no longer consider shells a mystical component of the operating system
  - you will in the future feel more comfortable writing programs that execute other programs

Shell scripts

- a shell given input from a file is executing a shell script
- some shells have elaborate command interpreters with variables, loops, and conditionals
- as a result, it is possible to write programs in the shell scripting languages
- such a program can be interpreted, easily modified (without recompilation), and is often more compact (but slower) than a comparable C or Java program

Unix Shell scripts

- when execv tries to execute a file, it checks to see whether it begins with the characters

  #!

  If it does, the remainder of the first line is assumed to be a path for an interpreter for the remaining lines of this file
- this allows interpretation of arbitrary scripting languages, e.g. perl, python
- because this is so convenient, for applications that are not performance critical or overly complicated it is very advantageous to use scripts

A design principle

- giving users (and programmers) flexibility leads to the creation of multiple efficient and effective ways of doing things
- this is good for expert users, though not necessarily for users that don't want to learn to program

Processes

- a program execution is a **process**
- a given program may be executed more than once, even simultaneously
- for a program to execute, it must have its own variables, return address, etc, so all these are part of a process
- however, some programs need to create multiple processes to run at the same time, e.g. a producer and a consumer of information

Processes and Threads

- multiple parts of a program running at the same time still need independent local variables and return addresses, but could:
  - share global variables and other global resources, e.g. file descriptors: **threads** within a process
  - not share global variables/resources: separate processes
- switching among threads within a process takes less work than switching among processes, so threads are also thought of as **lightweight processes**
- **user-space threads** don't rely on the kernel to schedule threads, and the thread switch may be extremely fast
- on the other hand, if a **kernel-space thread** blocks doing I/O, the kernel can wake up another thread. This is difficult with user-space threads

Programming Posix Threads

pthread_create calls a function and immediately returns

```
#include <stdio.h>
#include <unistd.h>  /* sleep */
#include <stdlib.h>  /* random */
#include <pthread.h>

static int global = 0;

static void * do_thread (void * arg) {
  char * string = (char *) arg;
  for (int i = 0; i < 5; i++) {
    printf ("%s: %d, %d\n", string, i, global++);
    sleep (1);
    if (random () % 2) {
      sleep (1);
    }
  }
}

int main (int argc, char **argv) {
  pthread_t ignore;
  for (int i = 1; i < argc; i++) {
    pthread_create (&ignore, NULL, do_thread, argv [i]);
  }
  sleep (10);
  printf ("main thread exiting\n");
}
```

- sample run:

```
hello: 0, 0
world: 0, 1
world: 1, 2
hello: 1, 3
world: 2, 4
hello: 2, 5
world: 3, 6
hello: 3, 7
world: 4, 8
hello: 4, 9
main thread exiting
```

Synchronization of Posix Threads

- declare a variable:

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
```

- acquire the lock (acquire the mutex):

```
pthread_mutex_lock(&lock1);
```

- once the _lock call completes, nobody else can be holding the lock, so now is a good time to access and modify any global variables associated with this mutex
- release the lock:

```
pthread_mutex_unlock(&lock1);
```

  now, anyone can acquire the lock again

- the association between the mutex and any global variables is entirely in the programmer's understanding

Synchronization of Java Threads

- a block of statements beginning with

  `synchronized(anyObjectReference)`

  can only be executed in one thread at a time, for that given object
  - i.e. two threads can execute the block at the same time, if referring to two different objects
- synchronized methods within the same class automatically synchronize with each other for a given object
  - i.e. two threads calling synchronized methods for the same object will be ordered so that one call goes first, and the other call goes after
  - this is true for calls to the same method, and also for calls to different methods
  - calls to methods of different objects of the same class can proceed concurrently
- see also: synchronized statements and synchronized methods

Process Table

- with multiple processes (or threads), the operating system must switch from one to the other
- to do this, the OS maintains a **process table** containing values relevant to each process, including stack pointer and register values needed to restart the process
- each table entry also stores the **state** of each process, typically one of:
  - *running*, the currently running process
  - *ready*, a process that could be selected for running
  - *blocked*, a process that cannot run until some external condition is met, e.g. data is available from the disk or the network (different blocked states can be used for different conditions)
  - *exited*, a process whose data is still in the process table but which is no longer executable (also known as a **zombie**)
- other data in the process table may include accounting (time/memory used), process ID, current directory, saved registers, virtual memory tables, etc

Schedulers

- once a process blocks, another must be started
- to minimize latency of execution, different processes should be given the CPU on a quick rotation even if they do not block
- a timer interrupt fires on a regular basis so the operating system can make the current process *ready* and select another ready process to be *current*
- kernel code called the **scheduler** select a new ready process
  - if there are no ready processes, the scheduler must run the **null** process (an infinite loop), or go to a low-power mode
  - if there is only one ready process, the scheduler selects it
  - if there are multiple ready processes, the scheduler must select one of them

Scheduler Operation

- when a timer interrupt fires, the hardware saves a few registers (including the PC) onto the current stack and loads the PC with a fixed address. This address may depend on which device interrupted, in which case a number associated with the device is used as an index into the **interrupt vector** (**interrupt table**)
- the software in the **interrupt handler** must first save the remaining registers (including the stack pointer), so they can be restored when resuming the program. This must be done in assembler
- the software in the interrupt handler can now handle the interrupt
- after doing the device driver operations needed to handle the interrupt, the interrupt handler can now call the scheduler to figure out which process to restart
- the scheduler must reload all the registers (stored in the process table), and execute a return-from-interrupt instruction (or a return instruction) to start the new current process

A Layered Operating System

- book, figure 2-3 – (*n* sequential processes layered on top of a scheduler)
- lowest layer contains scheduler and interrupt handlers that do the least possible to handle devices, as well as a mechanism to allow communication among processes
- all other services, including file systems, networking protocols, etc are separate processes
  - in Minix, these are called servers
- to handle a disk interrupt, the disk interrupt handler builds a message and "sends" it to the file system manager
- if the file system manager was waiting for a message, it now becomes ready
- if the file system manager has high priority, the scheduler now makes it the current process (unless another process has more priority)
- the file system manager therefore executes quickly after the interrupt, perhaps issuing another message to a process that had performed a read system call
- a system like this is a **microkernel** system
- a famous early microkernel was Mach 3.0
- L4 is a currently popular microkernel, you can run a modified Linux on top of it

Operating System Structures

- a **monolithic** kernel is a single program. No message passing is needed, procedure calls are enough. Linux follows this model
- a **microkernel** has a "small-as-possible" kernel, and all other services are implemented as processes. Minix follows this model
- a **virtual machine** provides multiple computing environments equivalent to actual computers by responding appropriately when a "user program" performs a privileged operation. The instruction set of the virtual computer may be the same as of the actual computer (most virtual machines), or independent (Bochs)

L4 Kernel

- inspired by the Mach microkernel
- designed to be as fast as possible (originally written in assembly)
- machine dependent

seL4 Kernel

https://sel4.systems/Info/Docs/seL4-manual-latest.pdf

- create and manage virtual address spaces
- create and manage threads
- inter-process communication (IPC)
  - IPC used to send interrupts to unprivileged device drivers
- capabilities define access rights
- small implementation of 8700 lines of C
- proof of integrity and confidentiality
- analysis of worst-case execution time

seL4 Capabilities

- in Unix, a file descriptor is an integer that identifies an open file in the kernel
- a capability is likewise a reference to data stored in the kernel, that identifies something that this process may do
- capabilities may be sent via IPC
- new capabilities may be created by limiting some of the rights of existing capabilities
- capabilities can be revoked

seL4 API

- three basic system calls: send, receive, yield
  - call combines send+receive, reply responds to call
- the destination of a send is identified by a capability
  - this destination may be a thread, or an operation inside the kernel
- send and receive can both block
  - rendezvous: transmission only happens when both ready
  - non-blocking versions are available
- seL4 starts an initial thread with all the possible capabilities, including to all unused memory

seL4 Objects

- objects are allocated by the kernel within memory to which a thread has a capability
  - the thread loses access to that part of the memory
- capability nodes (**CNodes**) for capabilities
- thread control blocks, **TCB**s, one for each thread
- **endpoints** for IPC, and **notifications** and **interrupts** for interrupt handling
- **memory**: untyped memory and device memory
- **virtual address space** objects
- all objects are managed from user-space