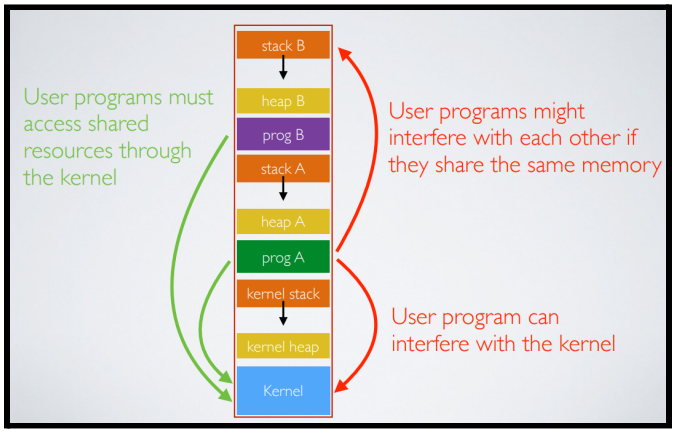
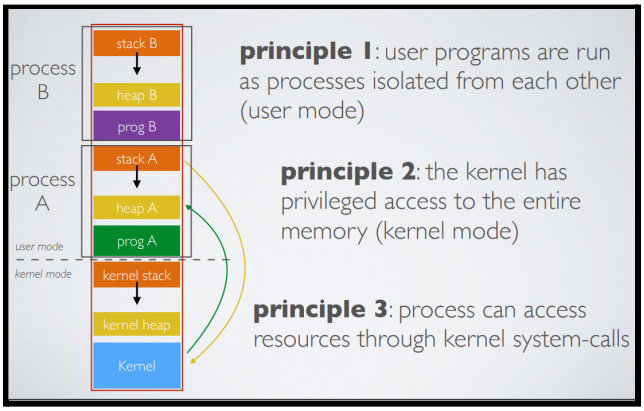
Lecture Notes:

- **The need for protection:**



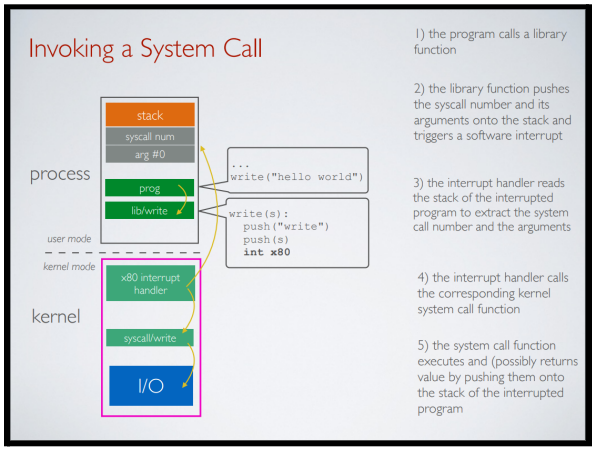
- **User mode:** User programs are run as processes isolated from each other.
- **Kernel Mode:** The kernel has privileged access to the entire memory.
- Processes can access resources through kernel system-calls.
- I.e.



- We can use virtual memory to isolate processes and kernel memory spaces. In a nutshell, user programs do not directly access the memory but the virtual memory that is somehow mapped onto the real memory and the kernel manages the virtual memory for all processes.
- **System Call:**
- **System calls** provide user programs with an API to use the services of the operating system. Think of it as an API. System calls look like some sort of "kernel API".
- There are 6 categories of system calls:
    1. Process control
    2. File management
    3. Device management
    4. Information/maintenance (system configuration)
    5. Communication (IPC)
    6. Protection

- E.g. Suppose you are writing a Bash-like program that reads keyboard inputs from the user.

  How do you know which keyboard to listen to? There's the default keyboard, there's wireless keyboard/bluetooth keyboards, and others.
  User programs do not operate I/O devices directly. The OS abstracts those functionalities and manages access through system calls.
- To invoke a system call, you have to use **software interrupts**/**syscall trap**. This is because user programs don't have access to the kernel's memory.
- E.g.

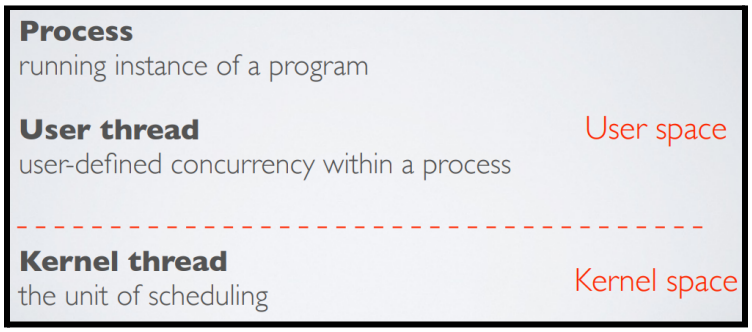


- **Process:**
- The **PCB (Process Control Block)** is a data structure to records process information:
  - Pid (process id) and ppid (parent process)
  - User (Optional)
  - Address space
  - Open files
  - Others
- A process is created by another process.
- The kernel creates the root process as part of the booting.
  E.g shell program for a simple OS
  E.g Window Manager for a GUI OS
- The root process has a pid of 0.
- **Process Creation on Unix:**
- The system call fork() creates a new process:
  1. Creates and initializes a new PCB.
  2. Creates a new address space.
  3. Initializes the address space with a copy of the entire contents of the address space of the parent, with one exception - the PCB.
  4. Initializes the kernel resources to point to the resources used by the parent process such as open files.
  5. Creates a kernel thread associated with this process and places that thread onto the ready queue.
- fork() is very useful when the child is cooperating with the parent and relies upon the parent's data to accomplish its task.
- Another very important system call is exec().
- **Note:** exec() does not create a new process, but rather, it runs the specified process.

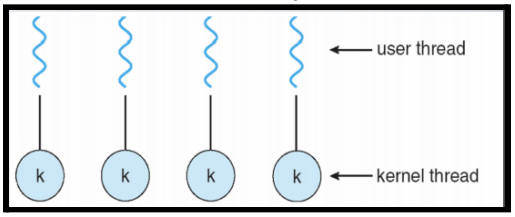
- int exec(char *prog, char *argv[])
    1. Stops the current process
    2. Loads the program “prog” into the process’ address space
    3. Initializes hardware context and args for the new program
    4. Places the PCB onto the ready queue
- You use fork() and exec() together. fork() creates a new process while exec() runs the new process. Most calls to fork are followed by exec. Using fork() and exec() in combination is called **spawning**. This is actually what happens when you run Unix commands. After you type a command and hit enter, a new process will be created via fork() and then exec will run the command you typed.
- In Unix, the wait() system call makes a parent process wait for at least 1 child process to terminate.
- In Unix, the exit() system terminates a process.
- **Process Creation on Window:**
- Windows doesn’t use fork() and exec() due to security concerns. It has its own function, BOOL CreateProcess(char *prog, char *args).
- CreateProcess:
    1. Creates and initializes a new PCB
    2. Creates and initializes a new address space
    3. Loads the program specified by “prog” into the address space
    4. Copies “args” into memory allocated in address space
    5. Initializes the saved hardware context to start execution at main (or wherever specified in the file)
    6. Places the PCB on the ready queue
- WaitForSingleObject() is the Windows equivalent of wait().
- ExitProcess(int status) is the Windows equivalent of exit().
- **User thread:**
- Threads are important because:
    1. Creating a process is costly (space and time)
    2. Context switching is costly (time)
    3. Inter-process communication is costly (time)

  If you want something more lightweight, you can use **user threads**.
- Modern OSes separate the concepts of processes and threads.
- The thread defines a sequential execution stream within a process.
- The process defines the address space and general process attributes.
- A thread is bound to a single process but a process can have multiple threads.
- Benefits of threads:
    1. Responsiveness:
       An application can continue running while it waits for some events in the background.
    2. Resource sharing:
       Threads can collaborate by reading and writing the same data in memory instead of asking the OS to pass data around.
    3. Economy of time and space:
       No need to create a new PCB and switch the entire context (only the registers and the stack).
    4. Scalability in multi-processor architecture:
       The same application can run on multiple cores.

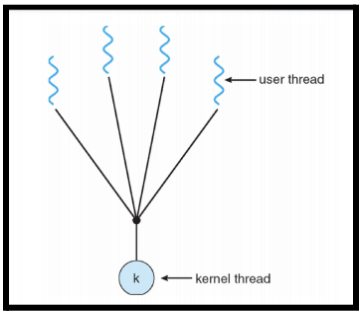
- **Multithreading Model:**
- Process vs Threads:



- POSIX Thread APIs:
  1. Create a new thread
     - Run this function: **tid thread_create (void (*fn) (void *), void *);**
     - When you run the function, it will:
       - Allocate Thread Control Block (TCB)
       - Allocate stack
       - Put func, args on stack
       - Put thread on ready list
  2. Destroy current thread
     - Run this function: **void thread_exit ();**
  3. Wait for thread thread to exit
     - Run this function: **void thread_join (tid thread);**
- There are 3 types of multithreading models:
  1. **One-to-one model**
     - Kernel-level threads/native threads
     - Each kernel thread only has 1 user thread.



     - The kernel manages and schedules threads. Furthermore, all thread operations are managed by the kernel.
     - This way is good for scheduling but bad for speed.
  2. **Many-to-one model**
     - User-level threads/green threads
     - Each process has only 1 kernel thread, and that kernel thread can have multiple user threads.



- Thread management and scheduling is delegated to a library, meaning that the kernel is not involved.
- Because the kernel isn't involved, this method is very lightweight and fast but all threads can be blocked if one thread is waiting for an event and cannot be scheduled on multiple cores.

3. **Many-to-many model**
    - Hybrid threads/n:m threading
    - This model maps some N number of user threads onto some M number of kernel threads. This is a compromise between the one-to-one model and many-to-one model.
    - In general, this model is more complex to implement than the other two, because changes to both kernel and user-space code are required.
    - Here, the threading library is responsible for scheduling user threads on the available schedulable kernel threads. This makes context switching of threads very fast, as it avoids system calls.

Textbook Notes:

- **Process API:**
- The fork() System Call:
- The fork() system call is used to create a new process.
- If successful, fork() will return the pid (process id) of the child process to the parent process and 0 to the newly created child process.
- If unsuccessful, fork() will return a negative number, usually -1.
- When the child process is created, there are now two active processes in the system that we care about: the parent and the child. Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point. Hence, with fork(), you might get a race condition.
- The CPU scheduler determines which process runs at a given moment in time. Because the scheduler is complex, we cannot usually make strong assumptions about what it will choose to do, and hence which process will run first. This nondeterminism, as it turns out, leads to some interesting problems, particularly in multi-threaded programs.
- The wait() System Call:
- A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent process **continues** its execution.
- The exec() System Call:
- A final and important piece of the process creation API is the exec() system call.
- This system call is useful when you want to run a program that is different from the calling program.
- The exec family of functions replaces the current running process with a new process.
- Motivating The API:
- The separation of fork() and exec() is essential in building a UNIX shell, because it lets the shell run code after the call to fork() but before the call to exec(); this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.
- The shell is just a user program. It shows you a prompt and then waits for you to type something into it. You then type a command (i.e., the name of an executable program, plus any arguments) into it; in most cases, the shell then figures out where in the file system the executable resides, calls fork() to create a new child process to run the command, calls some variant of exec() to run the command, and then waits for the command to complete by calling wait(). When the child completes, the shell returns from wait() and prints out a prompt again, ready for your next command.

- Process Control And Users:
- Beyond fork(), exec(), and wait(), there are a lot of other interfaces for interacting with processes in UNIX systems.
- For example, the kill() system call is used to send signals to a process, including directives to pause, die, and other useful imperatives.
- The man pages contain a lot of helpful details and information.
- **Thread API:**
- Thread Creation:
- The first thing you have to be able to do to write a multi-threaded program is to create new threads, and thus some kind of thread creation interface must exist.
- Here's how to do it in POSIX:

```
#include <pthread.h>
int
pthread_create(pthread_t       *thread,
          const pthread_attr_t *attr,
                void           *(*start_routine)(void*),
                void           *arg);
```

There are four arguments: thread, attr, start routine, and arg.
The first, thread, is a pointer to a structure of type pthread t; we'll use this structure to interact with this thread, and thus we need to pass it to pthread_create() in order to initialize it.
The second argument, attr, is used to specify any attributes this thread might have.
The third argument, start_routine, is a function pointer. The function start_routine takes in a void pointer.
Finally, the fourth argument, arg, is the argument to be passed to the function where the thread begins execution. You might ask: why do we need these void pointers? Well, the answer is quite simple: having a void pointer as an argument to the function start_routine allows us to pass in any type of argument; having it as a return value allows the thread to return any type of result.

- Thread Completion:
- For thread completion, you must call the function pthread_join().
  int pthread_join(pthread_t thread, void **value_ptr);
- This routine takes two arguments.
- The first is of type pthread_t, and is used to specify which thread to wait for. This variable is initialized by the thread creation routine (when you pass a pointer to it as an argument to pthread_create()). If you keep it around, you can use it to wait for that thread to terminate.
- The second argument is a pointer to the return value you expect to get back. Because the routine can return anything, it is defined to return a pointer to void; because the pthread_join() routine changes the value of the passed in argument, you need to pass in a pointer to that value, not just the value itself.

- Locks:
- Beyond thread creation and join, probably the next most useful set of functions provided by the POSIX threads library are those for providing mutual exclusion to a critical section via locks. The most basic pair of routines to use for this purpose is provided by the following:

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

- When you have a region of code that is a critical section, and thus needs to be protected to ensure correct operation, locks are quite useful.
- Here's an example:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

The intent of the code is as follows: if no other thread holds the lock when pthread_mutex_lock() is called, the thread will acquire the lock and enter the critical section. If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock (implying that the thread holding the lock has released it via the unlock call). Of course, many threads may be stuck waiting inside the lock acquisition function at a given time; only the thread with the lock acquired, however, should call unlock.

Unfortunately, this code is broken, in two important ways.
The first problem is a lack of proper initialization. All locks must be properly initialized in order to guarantee that they have the correct values to begin with and thus work as desired when lock and unlock are called. With POSIX threads, there are two ways to initialize locks. One way to do this is to use PTHREAD_MUTEX_INITIALIZER, as follows:
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
Doing so sets the lock to the default values and thus makes the lock usable.
The second problem with the code above is that it fails to check error codes when calling lock and unlock. Just like virtually any library routine you call in a UNIX system, these routines can also fail! If your code doesn't properly check error codes, the failure will happen silently, which in this case could allow multiple threads into a critical section.

- Condition Variables:
- The other major component of any threads library, and certainly the case with POSIX threads, is the presence of a condition variable.
- Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue.
- Two primary routines are used by programs wishing to interact in this way:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- To use a condition variable, one has to have a lock that is associated with this condition. When calling either of the above routines, this lock should be held.
- The first routine, pthread_cond_wait(), puts the calling thread to sleep, and thus waits for some other thread to signal it, usually when something in the program has changed that the now-sleeping thread might care about.