Context Switches, IPC, and Threads

CS 256/456

Dept. of Computer Science, University of Rochester

Last Class

- Processes
  - Process concept
  - Operations on processes
  - A process's image in a computer
- System protection and kernel mode
- System calls and the interrupt interface
- Signals
  - User-level events
- Pipes
  - Inter-process communication

Today

- Inter-process communication
  - Shared memory
- Context switches and the scheduling process
- Threads
  - Thread concept
  - Multithreading models
  - Types of threads

Interprocess Communication

- Reasons for processes to cooperate
  - Information sharing (e.g., files)
  - Computation speedup
  - Modularity and protection
  - Convenience - multitasking

Interprocess Communication: Pipes

- Conduit allowing two processes to communicate
  - Unidirectional or bidirectional
  - Full-duplex or half-duplex two-way communication
  - Is parent-child relationship required?
  - Is communication across a network allowed?

Ordinary Unix Pipes

- A unidirectional data channel that can be used for interprocess communication
- Treated as a special type of file, accessed using read() and write()
- Cannot be accessed from outside the process that created it unless inherited (by a child)
- Pipe ceases to exist once closed or when process terminates
- System calls
  - pipe (int fd[])
  - dup2

Example

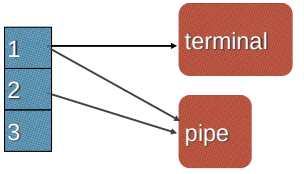
- pipe(int fd[])
  - fd[0] = read_end
  - fd[1] = write_end

fd[1] fd[0]

```
int fd[2];
pid_t pid;

pipe(fd);
pid = fork();
if (pid > 0) {
   /* Parent Process */
   close (fd[0]);

   /* Write a message to the child process */
   write (fd[1], write_msg, strlen(write_msg)+1);
   close (fd[1]);
} else {
   /* Child Process */
   close(fd[1]);

   /* Read a message from the parent process */
   read(fd[0], read_msg, BUFFER_SIZE);
   printf("read %s", read_msg);
   close(fd[0];
}
```


dup2() System Call

- Make one file descriptor point to the same file as another
- dup2 (old_fd, new_fd)
- Return value is -1 on error and new_fd on success
- dup2(1,2)



Standard In, Out, and Error

- By convention, file descriptors 0, 1, and 2 are used for:
  - Standard Input
  - Standard Output
  - Standard Error

Mechanisms for Interprocess Communication

- Shared memory
- Message passing
  - Pipes, sockets, remote procedure calls

Shared Memory: POSIX interface

- shmget – returns the identifier of a shared memory segment
- shmat – attaches the shared memory segment to the address space
- shmdt – detaches the segment located at the specified address
- shmctl – control of shared memory segments, including deletion

- Other possibilities: mmap (file sharing with preserved properties)

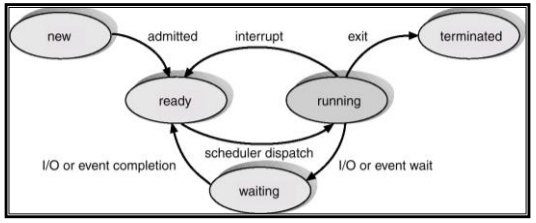
Message Passing

- Direct or indirect communication – processes or ports
- Fixed or variable size
- Send by copy or reference
- Automatic or explicit buffering
- Blocking or non-blocking (send or receive)

- Examples: client-server sockets, Mach ports, Windows 2000 local procedure call (LPC), remote procedure call (RPC, RMI)

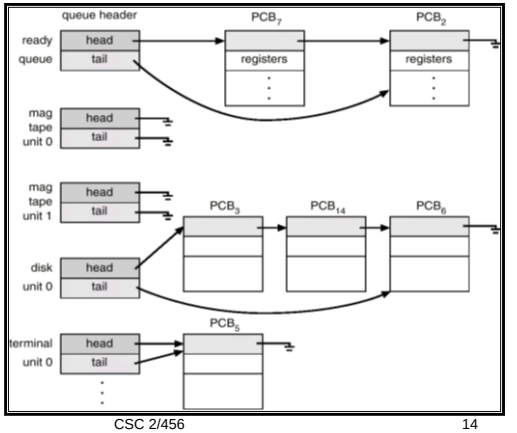
Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a process
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution



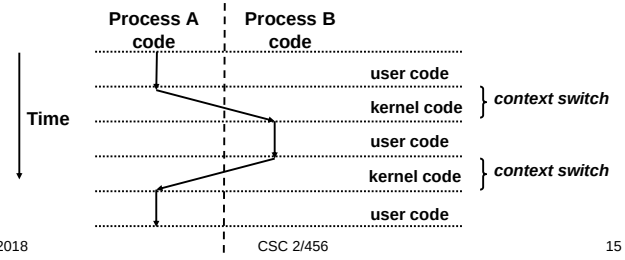
Queues for PCBs

- Ready queue – set of all processes ready for execution.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues.



Context Switching

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a *context switch.*



Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

save all callee-saves registers on stack, including ra and fp

*current := sp

current := other

sp := *current

pop all callee-saves registers (including ra, but NOT sp!)

return (into different coroutine!)

Uniprocessor Scheduling

- Use Ready List to reschedule voluntarily (cooperative threading)

reschedule:
- t : cb := dequeue(ready_list)
- transfer(t)

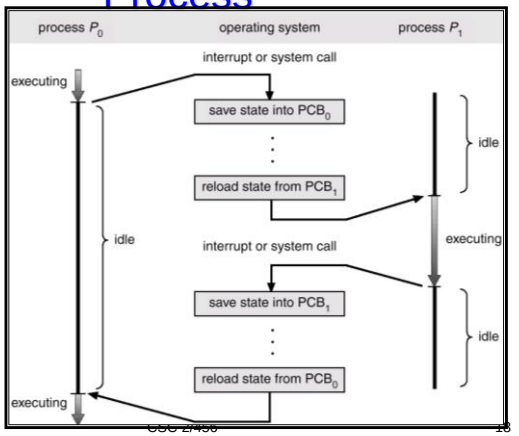
yield:
- enqueue(ready_list, current)
- reschedule

sleep_on(q):
enqueue(q, current)
reschedule

CPU Switch From Process to Process

When can the OS switch the CPU from one process to another?

Which one to switch to? - scheduling



Preemption

- Use timer interrupts or signals to trigger involuntary yields
- Protect scheduler data structures by disabling/reenabling prior to/after rescheduling

yield:
disable_signals
enqueue(ready_list, current)
reschedule
re-enable_signals

Process Termination

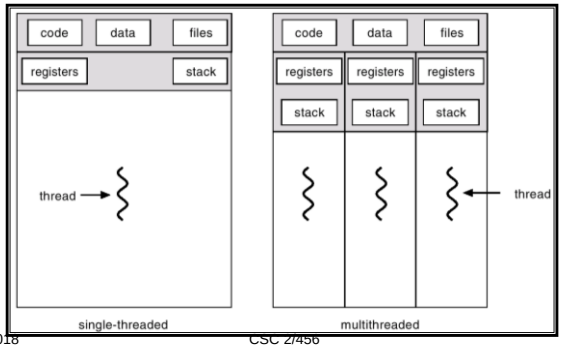
- Process executes last statement and gives the control to the OS (**exit**)
  - Notify parent if it is **wait**-ing
  - Deallocate process's resources
- The OS may forcefully terminate a process.
  - Software exceptions
  - Receiving certain signals

Processes or Threads

- A process or thread is a potentially-active execution context
- Processes/threads can come from
  - Multiple CPUs
  - Kernel-level multiplexing of single physical CPU (kernel-level threads or processes)
  - Language or library-level multiplexing of kernel-level abstraction (user-level threads)
- Threads can run
  - Truly in parallel (on multiple CPUs)
  - Unpredictably interleaved (on a single CPU)
  - Run-until-block (coroutine-style)

Processes and Threads

- Thread – a program in execution; without a dedicated address space.
- OS memory protection is only applied to processes.



Processes Vs. Threads

- Process
  - Single address space
  - Single thread of control for executing program
  - State information
    - Page tables, swap images, file descriptors, queued I/O requests, saved registers
- Threads
  - Separate notion of execution from the rest of the definition of a process
  - Other parts potentially shared with other threads
  - Program counter, stack of activation records, control block (e.g., saved registers/state info for thread management)
  - Kernel-level (lightweight process) handled by the system scheduler
  - User-level handled in user mode

Why Use Threads?

- Multithreading is used for parallelism/concurrency. But why not multiple processes?
  - Memory sharing.
  - Efficient synchronization between threads
  - Less context switch overhead

User/Kernel Threads

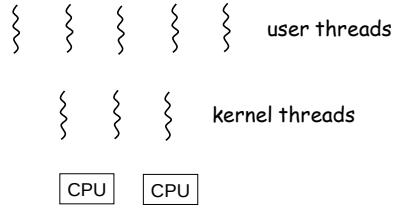
- User threads
  - Thread data structure is in user-mode memory
  - scheduling/switching done at user mode
- Kernel threads
  - Thread data structure is in kernel memory
  - scheduling/switching done by the OS kernel

User/Kernel Threads (cont.)

- Benefits of user threads
  - lightweight – less context switching overhead
  - more efficient synchronization??
  - flexibility – allow application-controlled scheduling
- Problems of user threads
  - can't use more than one processor
  - oblivious to kernel events, e.g., all threads in a process are put to wait when only one of them does I/O

Mixed User/Kernel Threads

- M user threads run on N kernel threads (M≥N)
  - N=1: pure user threads
  - M=N: pure kernel threads
  - M>N>1: mixed model



Solaris/Linux Threads

- Solaris
  - supports mixed model
- Linux
  - No standard user threads on Linux
  - Processes and threads treated in a similar manner (both called tasks)
  - Processes are tasks with exclusive address space
  - Tasks can also share the address space, open files, …

Pthreads

- Each OS has its own thread package with different Application Programming Interfaces ⇒ poor portability.
- Pthreads
  - A POSIX standard API for thread management and synchronization.
  - API specifies behavior of the thread library, not the implementation.
  - Commonly supported in UNIX operating systems.

Thread Creation

```
int pthread_create
  (pthread_t *new_id,
  const pthread_attr_t *attr,
  void *(*func) (void *),
  void *arg)
```

- new_id: thread's unique identifier
- attr: ignore for now
- func: function to be run in parallel
- arg: arguments for function func

Example of Thread Creation

```
void *func(void *arg) {
  int       *I=arg;
  …..
}

void main()
{
  int X;    pthread_t      id;
  ….
  pthread_create(&id, NULL, func, &X);
  …
}
```


Pthread Termination

```
void pthread_exit(void *status)
```

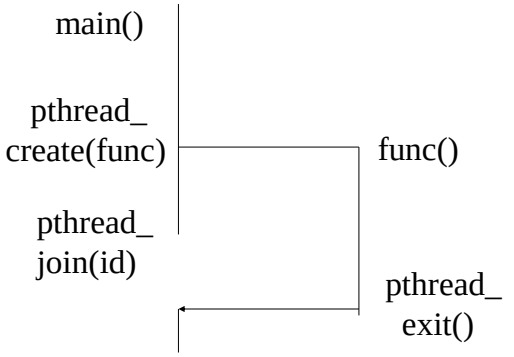
- Terminates the currently running thread.
- Is implicit when the function called in pthread_create returns.

Thread Joining

int pthread_join(
pthread_t new_id,
void **status)

- Waits for the thread with identifier new_id to terminate, either by returning or by calling pthread_exit().
- Status receives the return value or the value given as argument to pthread_exit().

Example of Thread Creation



Contention Scope

- Process contention scope – thread library schedules user threads onto light-weight processes (kernel-level thread)
  - Use priority as defined by user – no preemption of threads with same priority
- System contention scope – compete with all tasks and schedule kernel thread on a physical CPU
- pthreads: PTHREAD_SCOPE_PROCESS, PTHREAD_SCOPE_SYSTEM
  - pthread_attr_setscope
  - pthread_attr_getscope

Pthread Attributes

- Pthread_attr_init(pthread_attr_t *attr), destroy – initializes attr to default value
  - Scope – pthread_attr_setscope (&attr, SCOPE)
  - Stack size – pthread_attr_getstacksize, pthread_attr_setstacksize
  - Priority
  - Joinable or detached

Issues with the Threading Model

- Thread-local storage – what about globals?
- Stack management
- Interaction with fork and exec system calls
  - Two versions of fork?
- Signal handling – which thread should the signal be delivered to?
  - Synchronous
  - All
  - Assigned thread
  - Unix: could assign a specific thread to handle signals
  - Windows: asynchronous procedure calls, which are thread-specific

Issues with the Threading Model

- Thread-local storage – what about globals?
- Stack management
- Interaction with fork and exec system calls
  - Two versions of fork?
- Signal handling – which thread should the signal be delivered to?
  - Synchronous
  - All
  - Assigned thread
  - Unix: could assign a specific thread to handle signals
  - Windows: asynchronous procedure calls, which are thread-specific

Multiprocessor Scheduling

- Disabling signals not sufficient
- Acquire scheduler lock when accessing any scheduler data structure, e.g.,

```
yield:
    disable_signals
    acquire(scheduler_lock) // spin lock
    enqueue(ready_list, current)
    reschedule
    release(scheduler_lock)
    re-enable_signals
```

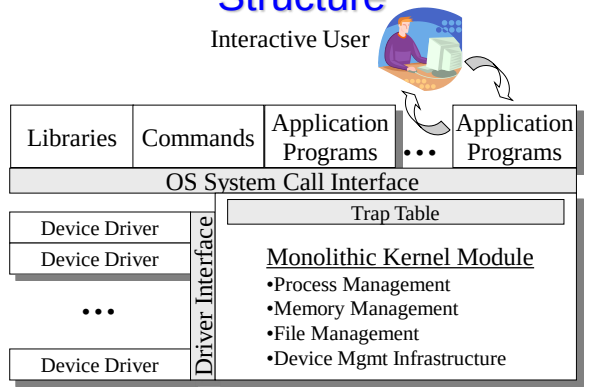

Ready List Management

- Various options are possible with varying levels of concurrency -
  - Single lock for all data structures
  - Multiple locks, one per data structure
  - Local freelists for control blocks and stacks (in the case of threads), single shared locked ready list
  - Queue of idle processors with preallocated control block and stack waiting for work
  - Local ready list per processor, each with its own lock

Operation System Architectures

- Monolithic architecture
- Microkernel architecture
- Layered architecture
- Virtual machines

OS Architecture: Monolithic Structure



Most modern OSes fall into this category!

Microkernel System Architecture

- Microkernel architecture:
  - Moves as much from the kernel into "*user*" space (still protected from normal users).
  - Communication takes place between user modules using message passing.
- What must be in the kernel and what can be in user space?
  - Mechanisms determine how to do something.
  - Policies decide what will be done.
- Benefits:
  - More reliable (less code is running in kernel mode)
  - More secure (less code is running in kernel mode)
- Disadvantage:
  - More overhead in inter-domain communications

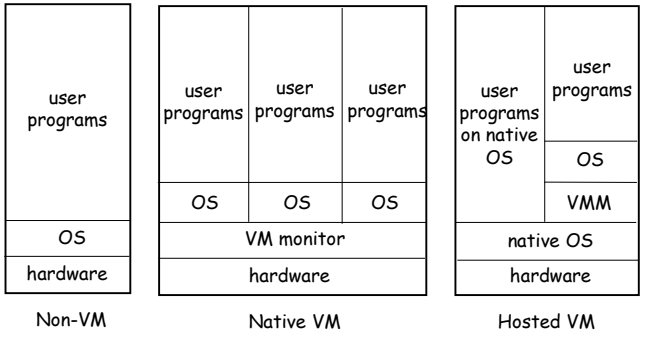
Layered Structure

- Layered structure
  - The operating system is divided into a number of layers (levels), each built on top of lower layers.
  - The bottom layer (layer 0), is the hardware.
  - The highest (layer N) is the user interface.
  - Decreased privileges for higher layers.
- Benefits:
  - more reliable
  - more secure
- Disadvantage:
  - Weak integration results in performance penalty (similar to the microkernel structure).

Virtual Machines

- Virtual machine architecture
  - Virtualization: A piece of software that provides an interface *identical* to the underlying bare hardware.
    - the upper-layer software has the illusion of running directly on hardware
    - the virtualization software is called virtual machine monitor
  - Multiplexing: It may provide several virtualized machines on top of a single piece of hardware.
    - resources of physical computer are shared among the virtual machines
    - each VM has the illusion of owning a complete machine
- Trust and privilege
  - the VM monitor does not trust VMs
  - only the VM monitor runs in full privilege
- Compared to an operating system
  - VM monitor is a resource manager, but not an extended machine

Virtual Machine Architecture



Disclaimer

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Dave O'Hallaron, Randal Bryant, Kai Shen, and John Criswell. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).