CIS520 – Operating Systems
Handout 2
Processes and Threads

- A process is an execution stream in the context of a particular process state.
  - An execution stream is a sequence of instructions.
  - Process state determines the effect of the instructions. It usually includes (but is not restricted to):
    * Registers
    * Stack
    * Memory (global variables and dynamically allocated memory)
    * Open file tables
    * Signal management information

    Key concept: processes are separated: no process can directly affect the state of another process.
- Process is a key OS abstraction that users see - the environment you interact with when you use a computer is built up out of processes.
  - The shell you type stuff into is a process.
  - When you execute a program you have just compiled, the OS generates a process to run the program.
  - Your WWW browser is a process.
- Organizing system activities around processes has proved to be a useful way of separating out different activities into coherent units.
- Two concepts: uniprogramming and multiprogramming.
  - Uniprogramming: only one process at a time. Typical example: DOS. Problem: users often wish to perform more than one activity at a time (load a remote file while editing a program, for example), and uniprogramming does not allow this. So DOS and other uniprogrammed systems put in things like memory-resident programs that invoked asynchronously, but still have separation problems. One key problem with DOS is that there is no memory protection - one program may write the memory of another program, causing weird bugs.
  - Multiprogramming: multiple processes at a time. Typical of Unix plus all currently envisioned new operating systems. Allows system to separate out activities cleanly.
- Multiprogramming introduces the resource sharing problem - which processes get to use the physical resources of the machine when? One crucial resource: CPU. Standard solution is to use preemptive multitasking - OS runs one process for a while, then takes the CPU away from that process and lets another process run. Must save and restore process state. Key issue: fairness. Must ensure that all processes get their fair share of the CPU.
- How does the OS implement these abstractions?
- How does machine implement context switch? A processor has a limited amount of physical resources. For example, it has only one register set. But every process on the machine has its own set of registers. Solution: save and restore hardware state on a context switch. Save the state in Process Control Block (PCB). What is in PCB? Depends on the hardware.

  - Registers - almost all machines save registers in PCB.
  - Processor Status Word.
  - What about memory? Most machines allow memory from multiple processes to coexist in the physical memory of the machine. Some may require Memory Management Unit (MMU) changes on a context switch. But, some early personal computers switched all of process's memory out to disk (!!!).

- Operating Systems are fundamentally event-driven systems - they wait for an event to happen, respond appropriately to the event, then wait for the next event. Examples:
  - User hits a key. The keystroke is echoed on the screen.
  - A user program issues a system call to read a file. The operating system figures out which disk blocks to bring in, and generates a request to the disk controller to read the disk blocks into memory.
  - The disk controller finishes reading in the disk block and generates and interrupt. The OS moves the read data into the user program and restarts the user program.
  - A Mosaic or Netscape user asks for a URL to be retrieved. This eventually generates requests to the OS to send request packets out over the network to a remote WWW server. The OS sends the packets.
  - The response packets come back from the WWW server, interrupting the processor. The OS figures out which process should get the packets, then routes the packets to that process.
  - Time-slice timer goes off. The OS must save the state of the current process, choose another process to run, the give the CPU to that process.

- When build an event-driven system with several distinct serial activities, threads are a key structuring mechanism of the OS.

- A thread is again an execution stream in the context of a thread state. Key difference between processes and threads is that multiple threads share parts of their state. Typically, allow multiple threads to read and write same memory. (Recall that no processes could directly access memory of another process). But, each thread still has its own registers. Also has its own stack, but other threads can read and write the stack memory.

- What is in a thread control block? Typically just registers. Don't need to do anything to the MMU when switch threads, because all threads can access same memory.

- Typically, an OS will have a separate thread for each distinct activity. In particular, the OS will have a separate thread for each process, and that thread will perform OS activities on behalf of the process. In this case we say that each user process is backed by a kernel thread.
  - When process issues a system call to read a file, the process's thread will take over, figure out which disk accesses to generate, and issue the low level instructions required to start the transfer. It then suspends until the disk finishes reading in the data.
  - When process starts up a remote TCP connection, its thread handles the low-level details of sending out network packets.

- Having a separate thread for each activity allows the programmer to program the actions associated with that activity as a single serial stream of actions and events. Programmer does not have to deal with the complexity of interleaving multiple activities on the same thread.

- Why allow threads to access same memory? Because inside OS, threads must coordinate their activities very closely.
  - If two processes issue read file system calls at close to the same time, must make sure that the OS serializes the disk requests appropriately.
  - When one process allocates memory, its thread must find some free memory and give it to the process. Must ensure that multiple threads allocate disjoint pieces of memory.

  Having threads share the same address space makes it much easier to coordinate activities - can build data structures that represent system state and have threads read and write data structures to figure out what to do when they need to process a request.

- One complication that threads must deal with: asynchrony. Asynchronous events happen arbitrarily as the thread is executing, and may interfere with the thread's activities unless the programmer does something to limit the asynchrony. Examples:
  - An interrupt occurs, transferring control away from one thread to an interrupt handler.
  - A time-slice switch occurs, transferring control from one thread to another.
  - Two threads running on different processors read and write the same memory.
- Asynchronous events, if not properly controlled, can lead to incorrect behavior. Examples:
  - Two threads need to issue disk requests. First thread starts to program disk controller (assume it is memory-mapped, and must issue multiple writes to specify a disk operation). In the meantime, the second thread runs on a different processor and also issues the memory-mapped writes to program the disk controller. The disk controller gets horribly confused and reads the wrong disk block.
  - Two threads need to write to the display. The first thread starts to build its request, but before it finishes a time-slice switch occurs and the second thread starts its request. The combination of the two threads issues a forbidden request sequence, and smoke starts pouring out of the display.
  - For accounting reasons the operating system keeps track of how much time is spent in each user program. It also keeps a running sum of the total amount of time spent in all user programs. Two threads increment their local counters for their processes, then concurrently increment the global counter. Their increments interfere, and the recorded total time spent in all user processes is less than the sum of the local times.
- So, programmers need to coordinate the activities of the multiple threads so that these bad things don't happen. Key mechanism: synchronization operations. These operations allow threads to control the timing of their events relative to events in other threads. Appropriate use allows programmers to avoid problems like the ones outlined above.