CompSci131

Parallel and Distributed Systems

Prof. A. Veidenbaum

Today's topics

- **Processes and threads in OS and DS**
  - OS and processes
  - Threads
  - Pthreads

- **Reading assignment:**
  - Today: 3.1
  - Next time: Online lecture 7

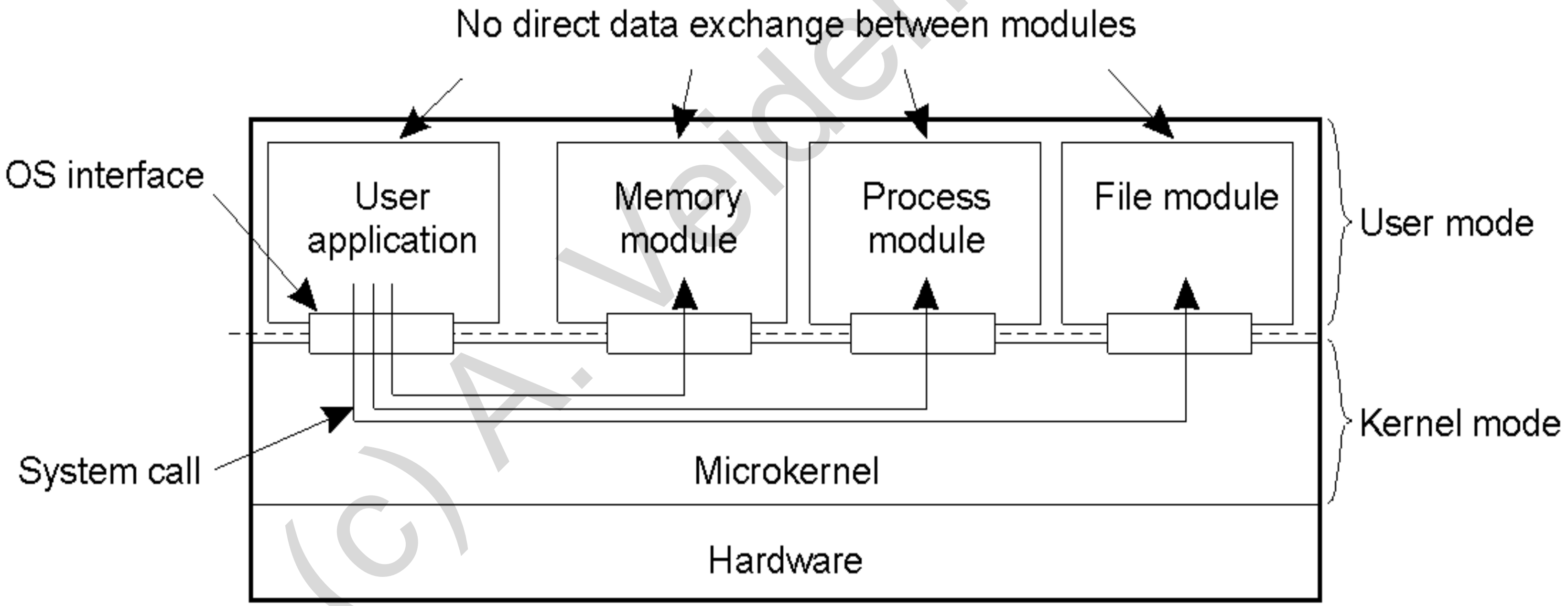
- **Lab 1 will be out today**

Last Lecture Covered

- **Middleware organization**
- **System Architectures**
  - **Centralized and Decentralized**

Uniprocessor OS

- **Presents a user with a virtual machine, syscall interface**
- **A microkernel isolates hrdw details, exports services**
- **Provides protection in sharing resources**



Processes

- A major Operating Systems (OS) concept
  - An instance of a program in execution
  - Can be in a running, blocked, ready states
    - » Describes a program code, data, state, resources
      - A process can be restarted from this description
  - Created by Unix fork()

- The OS manages and schedules processes
  - They <u>share</u> the system in a multi-programmed OS

- Processes may execute “concurrently” and are <u>protected</u> from each other
  - Each executes as if on its own processor
    - » As if it is the only program running…

- **The OS is responsible for "fair sharing" of system resources**
  - Including protection from other processes
- **A process executes until**
  - A hardware interrupt occurs OR
  - Its time slice is over (timer interrupt) OR
  - It calls the OS for a service OR
  - It ends or terminates
- **In all these cases a <u>context switch</u> occurs**
  - Another ready process will start executing
    - » first the OS itself, then (possibly) another user
  - Another process = another program

Multiprocessor OS (1)

- **Extends the OS to support multiple processors**
  - **Another resource to manage**
  - **Memory is SHARED**
    - » **all cores see the same *physical* address space**
    - » **But need to redo VM management**

- **Major difference: multiple programs running at once**
  - **Need to protect resources in concurrent use**
    - » **Allocation**
    - » **Sharing**
  - **Need to synchronize access to the key OS data structures**
    - » **How to do this**
  - **A process may be blocked if a resource it needs is busy**

Mutual Exclusion (1)

- **Use *semaphores***
  - **Signals whether access is open or blocked:**
    - » **1 - blocked, 0 - open**
  - **Protects access to guarantee single process access**
    - » **Hard to program with**

```
/* Count initialized to 0 */
if count = 0
then count = count + 1;
else    "repeat"; /*busy-wait" */
```

- **Can this be done without hardware support?**
  - **Let's look at pseudo-assembly on 2 processors**

- **Process creation is very expensive**
  - Need to allocate resources
  - Initialize data structures
    - Memory maps, process table, etc
    - User program, data space including zeroing memory
  - It takes many thousands of instructions

- **A context switch is also quite expensive**
  - Save state, change some protected registers, MMU, etc
  - May require swapping one of the processes out
    - moving a process state – including memory - to disk

- **Process abstraction may not be well suited for DS**
  - Something “lighter” may be better!

Threads

- Similar to a process, but of finer granularity
  - Less state information and thus cheaper to create, switch
    - » Typically only the CPU state, private data
- Threads usually run *within* a process!
  - Share its Virtual Addr Space, but also have private space
  - No data access protection between threads is provided
    - » Thus a bit harder to program: *can clobber shared data*
- Context is the CPU context + private data
  - plus thread info, primarily for thread scheduling
- There are a number of benefits from *multithreading*
  - A major one: a process may continue execution if one of its threads blocks
    - » A blocked process is context switched

Other Thread Advantages

- Threads can be managed in user space
  - No context switch to the OS for scheduling
    - » Some implementations may still use the OS for this
      - But it is a "lighter" context switch
- Allows to exploit parallelism in a program
  - If multiple processors are available
    - » Memory is shared
- Allows parallelism in complex applications
  - These are typically done as a collection of processes
    - » They communicate using the *IPC* mechanism
      - Pipes, shared memory segments
    - » This is expensive!
      - IPC is through the OS kernel
  - Threads communication is cheaper - through shared memory

How does it work?

- Thread state is described in a Thread Table
  - Access protected by mutex locks
- A thread package has direct OS support
- When a thread blocks on thread synchronization a thread scheduler is called
  - Thread Table is updated
  - Another thread is selected for execution
  - The OS starts executing the new thread

Threads in DS - how to use?

- "A thread can be blocked instead of the entire process"
  - Good for expressing communication
    - » Can maintain multiple connections at once
- Consider multi-threaded clients
  - Example: a browser with multiple connections
    - » Can set up multiple connections to a server
      - E.g. to load multiple images
    - » Even better, to multiple servers
      - Works well because servers are typically *replicated*!
- Now let's consider a multithreaded server
  - Even more beneficial then multithreading a client!
  - Example: file server
    - » Waits for request, starts an operation, waits, replies

Multi-threaded (or concurrent) Servers

- A *concurrent* server does not handle requests
  - Passes a request on to a separate thread/process
    - » is basically a dispatcher
- How do clients contact a server?
  - At *endpoints* or *ports* on the server system
    - » How do they know a port name?
      - Global names as in the Internet ftp, http, etc
      - Dynamically created ports
        - Requires a name lookup
- Two possible implementations
  - A process for each server (most of UNIX servers)
    - » Idle most of the time
  - A super-server: a process listening to a number of ports
    - » Forks a process for each request. Process exits when finished
      - UNIX inetd listens to many ports for internet services

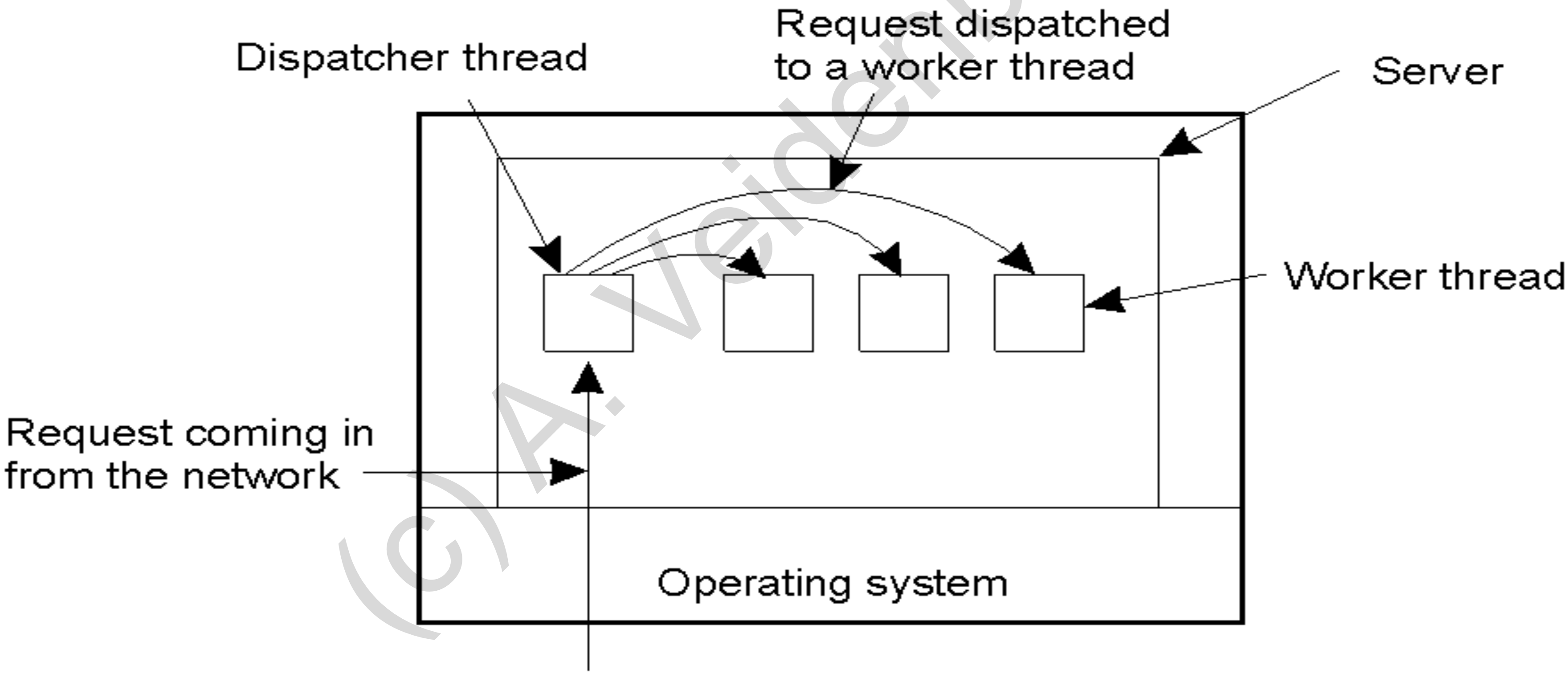
- **Can a server be interrupted?**
  1. **Kill the connection – after client dies or is killed**
  2. **Signal via a separate channel**
     » **high-priority communication, e.g. TCP urgent data transmit**
       - **Leads to a server interrupt**

- **One more issue:**
  – **Is the server stateless**
    » **Does not keep a client state**
    » **can change its own state without informing clients**
      - **Web servers are done this way**

  – **or statefull?**
    » **Keeps a client state**
      - **File server that allows a client to copy a file and read/write it**
      - **Needs to be able to recover from crashes**

One popular server model

- **Organized in a dispatcher/worker model**
- **Worker thread can block until file operation completed**
  - **Other threads will be scheduled after it blocks**



Pthreads

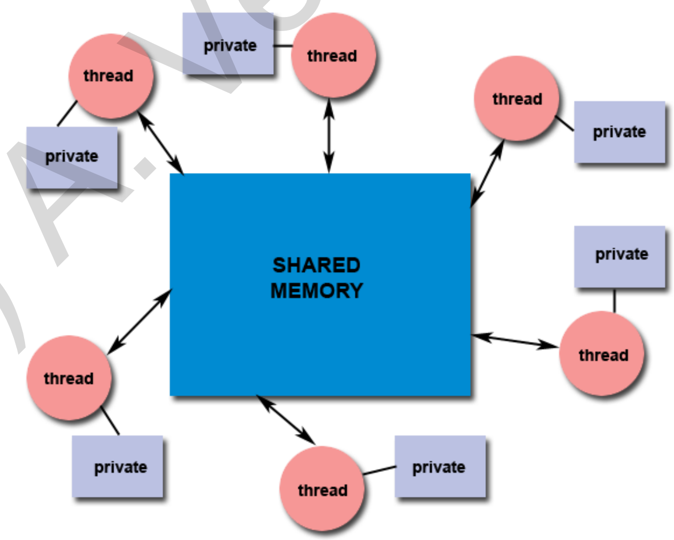
- **Multiple pthreads can be created in a program**
  - Threads may create other threads
    - » rc = pthread_create(thread_id, NULL, function, (void *) arg)
- **Basic thread behaviour**
  - OS creates a thread sharing virtual address space with parent
    - » Linux threads are actually child processes created by clone
      - Much less expensive than a fork, basically targeting pthreads
  - A thread executes the function and (usually) exits
  - Variables in a function's scope are private to a thread
  - Parent can be made to wait for a thread to finish
    - » pthread_join(thread_id, status)

POSIX Threads

- Thread operations
  - creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads
  - nor does it know the thread that created it.
- All threads within a process share
  - Process instructions
  - Most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id

POSIX Threads

- Each thread has a unique:
  - Thread ID
  - Registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: errno
- A pthread functions return "0" if all is OK



POSIX Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
main()
  {  pthread_t thread1, thread2;
     int  iret1, iret2;
     iret1 = pthread_create( &thread1, NULL, X_function, (void*)
     message1)
     iret2 = pthread_create( &thread2, NULL, X_function, (void*)
     message2);
      /* Do something while waiting for threads to complete */
     pthread_join( thread1, NULL);
     Pthread_join( thread2, NULL);
     }
```

POSIX Threads

pthread_exit(retval);

- **Threads can use mutex's**
  - **pthread_mutex_t MV = PTHREAD_MUTEX_INITIALIZER;**
  - **Pthread_mutex_init(mutex, attr) – dynamic creation**

  - pthread_mutex_lock(&mutex1)

  - pthread_mutex_unlock(&mutex1)