Network Programming

COSC 1176/1179

Lecture 6
Signals and Socket Related Functions
Lecture Overview

During this lecture, we will cover:

- Socket control and options
- Common signals and handling them
Signals Concepts

- Signals provide a mechanism for notifying programs that certain events have occurred.
  - usually the process doesn’t know ahead of time exactly when a signal will occur
  - Example: a timer expires, a user types the “interrupt” character

- Software interrupts

- Signals can be sent by:
  - a process to another process (or to itself)
  - the kernel to a process

- A signal is an asynchronous notification
  - ie: can occur anytime (e.g. user types ctrl-C)
  - Notification is delivered to the program regardless of where in the code it is executing
### Some Signal Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>Interrupt character typed</td>
<td>terminate process</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>Quit character typed (^)</td>
<td>create core image</td>
</tr>
<tr>
<td>SIGKILL</td>
<td><strong>kill</strong> -9</td>
<td>terminate process</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Invalid memory reference</td>
<td>create core image</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>Write on pipe but no reader</td>
<td>terminate process</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>alarm() clock 'rings'</td>
<td>terminate process</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>user-defined signal type</td>
<td>terminate process</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>user-defined signal type</td>
<td>terminate process</td>
</tr>
<tr>
<td>SIGIO</td>
<td>handle is ready for I/O</td>
<td>terminate process</td>
</tr>
</tbody>
</table>
Signal Sources

- terminal driver
  - SIGINT
  - SIGQUIT
  - SIGHUP
- memory management
  - SIGSEGV
- kernel
  - SIGPIPE
  - SIGALRM
- other user processes
  - SIGUSR1
- a process
  - SIGKILL
  - SIGWINCH

- shell command
- window manager
  - SIGWINCH

Lecture 10
Dr. Andrew Fry
© School of Computer Science and Information Technology
# Solaris Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>Signal#</th>
<th>Description</th>
<th>Name</th>
<th>Signal#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Hangs up</td>
<td>SIGSTOP</td>
<td>23</td>
<td>Stop*</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Interrupts</td>
<td>SIGTSTP</td>
<td>24</td>
<td>User stop TTY</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Quits</td>
<td>SIGCONT</td>
<td>25</td>
<td>Stopped process has continued</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Illegal instruction</td>
<td>SIGTTIN</td>
<td>26</td>
<td>BG TTY read attempted</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>5</td>
<td>Trace trap</td>
<td>SIGTTOU</td>
<td>27</td>
<td>BG TTY write attempted</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Used by abort</td>
<td>SIGVTALRM</td>
<td>28</td>
<td>Virtual timer expired</td>
</tr>
<tr>
<td>SIGEMT</td>
<td>7</td>
<td>EMT instruction</td>
<td>SIGPROF</td>
<td>29</td>
<td>Profiling timer expired</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Floating-point exception</td>
<td>SIGXCPU</td>
<td>30</td>
<td>Exceeded CPU limit</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Kill*</td>
<td>SIGXFSZ</td>
<td>31</td>
<td>Exceeded file size limit</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>10</td>
<td>Bus error</td>
<td>SIGWAITING</td>
<td>32</td>
<td>Process LWPs are blocked</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Segmentation violation</td>
<td>SIGLWP</td>
<td>33</td>
<td>used by thread library</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>12</td>
<td>Bad arg. to system call</td>
<td>SIGFREEZE</td>
<td>34</td>
<td>used by CPR</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>Pipe write with no reader</td>
<td>SIGTHAW</td>
<td>35</td>
<td>used by CPR</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Alarm clock</td>
<td>SIGCANCEL</td>
<td>36</td>
<td>Thread cancellation signal</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Software termination</td>
<td>SIGLWT</td>
<td>37</td>
<td>Resource lost</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>16</td>
<td>User-defined signal</td>
<td>SIGRTMIN</td>
<td>38</td>
<td>Highest priority RT signal</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>17</td>
<td>User-defined signal</td>
<td>SIGRTMAX</td>
<td>45</td>
<td>Lowest priority RT signal</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>18</td>
<td>Child status change alias</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGPWR</td>
<td>19</td>
<td>Power-fail restart</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGWINCH</td>
<td>20</td>
<td>Window size change</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGURG</td>
<td>21</td>
<td>Urgent socket condition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGIO</td>
<td>22</td>
<td>Socket I/O now possible</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* cannot be blocked or caught
Listing Signals

goanna.cs.rmit.edu.au% kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL
BUS SEGV SYS PIPE ALRM TERM USR1 USR2
CLD PWR WINCH URG POLL STOP TSTP CONT
TTIN TTOU VTALRM PROF XCPU XFSZ WAITING
LWP FREEZE THAW CANCEL LOST RTMIN RTMIN+1
RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1 RTMAX
Generating a Signal

(1) The kill command can be used to send signals.
(2) Certain terminal characters generate signals
   - an interrupt character (<cntrl>-c or Delete) generates a signal called SIGINT
   - a quit character (<cntrl>-backslash)
     » terminates a process
     » generates a signal called SIGQUIT to process and generates a core image of the process
     » (a core file image can be used with a debugger for analysis)
(3) Certain hardware conditions generate signals
    floating point arithmetic errors generate a signal called SIGFPE
(4) Certain software conditions can generate signals - the SIGURG signal is generated when some urgent data arrives on a socket
Generating a Signal

- Use the UNIX command:
  - $ kill -KILL 4481
    sends a SIGKILL signal to pid 4481
  - check using
    - `ps -l`
    to make sure process died
- `kill` is not a good name; `send_signal` might be better.
kill()

- Send a signal to a process (or group of processes).
- `#include <signal.h>`
- `int kill( pid_t pid, int signo );`
- Returns 0 if ok, -1 on error.
Possible Actions associated with Signals

- When a signal is delivered to a running program, one of followings may happen:
  - *ignore*: the signal is ignored. The process is never aware that the signal was delivered.
  - *default (usually exit)*: the program is forcibly terminated by the operating system
  - *blocked*: The signal is blocked.
    - prevented from having any effect until the program takes action to allow its delivery.
  - *user-specified signal handler*
    - Signal handlers are installed by the program
    - Signal handlers remain installed until explicitly altered
    - Signals associated with a given signal handler are blocked while that handler executes
    - Signals are not queued
    - (Posix signal semantics)
The **SIGPIPE** Signal

- Writing to a TCP socket which has been disconnected at the far end causes:
  - a **SIGPIPE** signal to be raised;
  - write to fail and `errno` to be set to `EPIPE`

- Solution:
  - set **SIGPIPE** to be ignored; and
  - handle **EPIPE** appropriately.
Catching a signal: An Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

main()
{
    void catchint();

    signal (SIGINT, catchint);
    printf("sleep call 1 \n");
    sleep(1);
    printf("sleep call 2 \n");
    sleep(1);
    printf("sleep call 3 \n");
    sleep(1);
    printf ("Exiting \n");
    exit(0);
}

/* trivial function to handle SIGINT */
void catchint (int signo)
{
    printf("\nCATCHINT: signo=%d\n", signo);
    printf("CATCHINT: returning\n\n");
}
```
Catching a signal: An Example

No user signals generated (i.e. <cntrl> - c)

User generates a signal

---

© School of Computer Science and Information Technology
signal() to trap interrupt

```c
#include <stdio.h>
#include <signal.h>
main()
{
    void cnt(int sig);

    signal(SIGINT, cnt);
    printf("Begin counting and INTERRUPTs
    for(;;); /* infinite loop */
}

void cnt(int sig)
{
    static int count=0;
    printf("Total of %d INTERRUPTS received\n", ++count);
    /* signal(SIGINT, cnt); */
}
```

Goanna.cs.rmit.edu.au% ./signalex1
Begin counting and INTERRUPTs
^CTotal of 1 INTERRUPTs received
^C
Goanna.cs.rmit.edu.au%
signal() to trap interrupt - Another example

```c
#include <stdio.h>     /* standard I/O functions */
#include <unistd.h>    /* standard unix functions, like getpid() */
#include <signal.h>    /* signal name macros, and the signal() prototype */

/* first, here is the signal handler */
void catch_int(int sig_num)
{
    /* re-set the signal handler again to catch_int, for next time */
    signal(SIGINT, catch_int);
    printf("Don't do that\n");
    fflush(stdout);
}

int main(int argc, char* argv[]) {
    /* set the INT (Ctrl-C) signal handler to 'catch_int' */
    signal(SIGINT, catch_int);

    /* now, lets get into an infinite loop of doing nothing. */
    for ( ;; )
        pause();
}
```

**signal() to trap interrupts**

```c
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <strings.h>
#include <string.h>

main()
{
    signal(SIGINT, cnt);
    signal(SIGTSTP, cnt2); /* SIGTSTP = cntl-Z */
    printf("Begin counting INTERRUPTs and cntl-Z \n");
    for(;;); /* infinite loop */
}

void cnt(int sig)
{
    static int count=0;
    printf("Total of %d INTERRUPTs received\n", ++count);
    /* signal(SIGINT, cnt); */
}

void cnt2(int sig)
{
    static int count=0;
    printf("Total of %d Cntl-Z 's received\n", ++count);
    signal(SIGTSTP, SIG_DFL);
    /* signal(SIGTSTP, cnt2); */
}
```

Begin counting INTERRUPTs and cntl-Z
^ZTotal of 1 Cntl-Z 's received
^CTotal of 1 INTERRUPTs received
alarm()

- Set an alarm timer that will 'ring' after a specified number of seconds
  - a SIGALRM signal is generated
- #include <unistd.h>
  long alarm(long secs);
- Returns 0, or number of seconds until previously set alarm would have 'rung'.
- Don't mix alarm() and sleep()
alarm() example

```c
#include <stdio.h>     /* standard I/O functions */
#include <unistd.h>    /* standard unix functions, like alarm() */
#include <signal.h>    /* signal name macros, and the signal() prototype */

char user[40];        /* buffer to read user name from the user */

/* define an alarm signal handler. */
void catch_alarm(int sig_num)
{
    printf("Operation timed out. Exiting...\n\n");
    exit(0);
}

int main(int argc, char* argv[])
{
    signal(SIGALRM, catch_alarm);       /* set a signal handler for ALRM signals */
    printf("Username: ");              /* prompt the user for input */
    fflush(stdout);
    alarm(30);                          /* start a 30 seconds alarm */
    gets(user);                         /* wait for user input */
    alarm(0);                           /* got input, remove the timer */
    printf("User name: '%s'\n", user);
    return 0;
}
```
signal(): library call

- Specify a signal handler function to deal with a signal type.

- `#include <signal.h>`
  
  ```c
  typedef void Sigfunc(int); /* my defn */
  
  Sigfunc *signal( int signo, Sigfunc *handler );
  
  - signal returns a pointer to a function that returns an int (i.e. it returns a pointer to Sigfunc)
  ```

- The actual prototype, listed in the “man” page is a bit perplexing but is an expansion of the `Sigfunc` type:

  ```c
  void (*signal(int signo, void(*handler)(int)))(int);
  ```
signal() and sigaction()

An application program can change the default behaviour for a particular signal using sigaction()

```c
int sigaction(int sig,  
              const struct sigaction *act,  
              struct sigaction *oact);
```

```c
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

- **sig** specifies the signal for which the behaviour is being changed.
- **act** parameter points to a sigaction structure that defines the new behaviour
- **oact** describes old behaviour
Blocking Behavior and examples

• The default behaviour of socket call is to block until the requested action is completed.

• recv() function in an echo client does not return until at least one message from the echo server is received. So, recv() and recvfrom() block when data is not available.

• A send on a tcp socket may block if there is not sufficient space to buffer the transmitted data.

• accept() in a server blocks until a client establishes a connection with connect().

• Long RTT, high error rate connections, slow server may cause a call to connect() to block for a significant amount of time.
Method for non-blocking using select()

struct connState {
  int socket;  // socket we're waiting for
  int action;  // what we will do next
  ... other params
};
struct connState conn[NN];

prepare sockets (and files) for use
while not finished
  prepare readset/writeset of conn sockets
  select(..., readset, writeset, ..., timeout)
  with each conn
    if FD_ISSET(conn[].socket, readset)
      performNextAction(conn);
    if FD_ISSET(conn[].socket, writeset)
      performNextAction(conn);
Non-Blocking `connect(...)`

- Blocking calls to `connect(...)` always block for at least twice the round trip time
  - can be longer
- Non-blocking calls to `connect(...)`:
  - may complete immediately; or
  - may return immediately with `errno=EINPROGRESS`
- We can change the default blocking behaviour with a call to `fcntl()` “file control”
The \texttt{fcntl()} Function

- Used to control such options as:
  - non-blocking I/O
  - signal driven I/O

- \texttt{int fcntl(int \textit{fd}, int \textit{cmd}, ... /*int \textit{arg}*/* \ ));}
  - \textit{fd} is the file descriptor to control
  - \textit{cmd} is a command, such as \texttt{F_SETFL}
  - \textit{arg} depends on the command, often an integer

```c
if (-1 == fcntl(sock, F_SETFL, O_NONBLOCK))
    DieWithError("Unable to set socket to non-blocking mode");
```
fcnt1() commands

F_GETFL / F_SETFL
    get/set main behavior flags
F_DUPFD
    duplicate handle
F_GETLK / F_SETLK
    creates/controls locks
F_GETOWN / F_SETOWN
    which process receives IO signals
F_GETSIG / F_SETSIG
    control which signal is sent on IO
### fcntl() F_SETFL flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_ASYNC</td>
<td>send SIGIO when IO is possible</td>
</tr>
<tr>
<td>O_NONBLOCK</td>
<td>use non-blocking IO</td>
</tr>
<tr>
<td>O_SYNC</td>
<td>use fully synchronous IO</td>
</tr>
</tbody>
</table>
Non-Blocking `accept(…)`

- Expect a listening socket indicated by `select(...)` to be readable to have a waiting connection.
- Some systems allow waiting connections to reset 'silently'.
- `accept(…)` should be non-blocking, and several errors are normally ignored:
  - EWOULDBLOCK
  - ECONNABORTED
  - EPROTO
  - EINTR
Method for non-blocking using signals

- In a normal UDP echo server when recvfrom() is called it blocks until a datagram arrives.
- In the modified echo server after creating and binding the socket, instead of calling recvfrom() and blocking until a datagram arrives, the echo server establishes a signal handler and begins doing other work.
- When a datagram arrives, signal is delivered to the process, triggering action of handler function.
- The handler function calls recvfrom(), echoes back any datagrams and then returns.
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind, and connect() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() and getpid() */
#include <fcntl.h>      /* for fcntl() */
#include <sys/file.h>   /* for O_NONBLOCK and FASYNC */
#include <signal.h>     /* for signal() and SIGALRM */
#include <errno.h>      /* for errno */

#define ECHOMAX 255     /* Longest string to echo */

void DieWithError(char *errorMessage);  /* Error handling function */
void UseIdleTime();                     /* Function to use idle time */
void SIGIOHandler(int signalType);      /* Function to handle SIGIO */

int sock;                        /* Socket -- GLOBAL for signal handler */
Modified Echo Server

```c
/* Bind to the local address */
if (bind(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
  DieWithError("bind() failed");

handler.sa_handler = SIGIOHandler; /* Use local signal handler for SIGIO */
if (sigfillset(&handler.sa_mask) < 0) /* Create mask that masks all signals */
  DieWithError("sigfillset() failed");
handler.sa_flags = 0; /* No flags */

if (sigaction(SIGIO, &handler, 0) < 0)
  DieWithError("sigaction() failed for SIGIO");

/* We must own the socket to receive the SIGIO message */
if (fcntl(sock, F_SETOWN, getpid()) < 0)
  DieWithError("Unable to set process owner to us");

/* Arrange for nonblocking I/O and SIGIO delivery */
if (fcntl(sock, F_SETFL, O_NONBLOCK | FASYNC) < 0)
  DieWithError("Unable to put client sock into non-blocking/async mode");

/* Go off and do real work; echoing happens in the background */

for (;;) 
  UseIdleTime();

/* NOTREACHED */
```
void SIGIOHandler(int signalType)
{
    struct sockaddr_in echoClntAddr; /* Addr of src datagram */
    unsigned int clntLen;              /* Address length */
    int recvMsgSize;                    /* Size of datagram */
    char echoBuffer[ECHOMAX];           /* Datagram buffer */

    do /* As long as there is input ... */
    {
        clntLen = sizeof(echoClntAddr); /* Set the size of the in-out parameter */

        if ((recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0, (struct sockaddr *) &echoClntAddr, &clntLen)) < 0)
        {
            if (errno != EWOULDBLOCK) /* acceptable error: recvfrom() would have blocked */
                DieWithError("recvfrom() failed");
        }
        else
        {
            printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

            if (sendto(sock, echoBuffer, recvMsgSize, 0, (struct sockaddr *) &echoClntAddr, sizeof(echoClntAddr)) != recvMsgSize)
                DieWithError("sendto() failed");
        }
    } while (recvMsgSize >= 0);
/* ... now nothing left to receive */
}
Three main approaches to configuring options associated with sockets:

- `getsockopt(...)` and `setsockopt(...)` functions
- the `fcntl(...)` function
- the `ioctl(...)` function
Accessing Socket Options

- \texttt{int getsockopt(int sockfd, int level, int optname, void \*optval, socklen\_t \*optlen);} 

- \texttt{int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t \*optlen);}
Socket Level Options

- **When level is SOL_SOCKET**
- **Options include:**
  - `SO_BROADCAST`: Broadcast allowed
  - `SO_ERROR`: fetch the most recent error on the socket
  - `SO_KEEPALIVE`: send periodic keep-alive probes to make sure both ends are still alive.
  - `SO_LINGER`: make close(...) ‘linger’ as the connection is terminated
  - `SO_RCVBUF` / `SO_SNDBUF`: set/get kernel buffer sizes
  - `SO_RCVLOWAT` / `SO_SNDLOWAT`: minimum number of available bytes that will cause recv() to return
  - `SO_RCVTIMEO` / `SO_SNDTIMEO`: provide timeouts
  - `SO_REUSEADDR`: allow port number to be re-used
  - `SO_TYPE`: used to find the type of the socket.
**TCP Level Options**

- **When level is IPPROTO_TCP**
- **Options include:**
  - TCP_MAXSEG: set/get the maximum segment size
  - TCP_NODELAY: turn off Nagle's Algorithm
    - One very common cause of long latencies is if the sender socket is opened with Nagel's algorithm enabled. Disabling Nagel's algorithm means that the data is sent immediately. Nagel's algorithm is used to "wait" with sending of small packets until more data is available, because sending bigger packets has less overhead.
  - TCP_KEEPALIVE: set the period for keepalive probes. The default timeout is 2 hours. On Linux systems the timeout can be set by writing the time in seconds to the file /proc/sys/net/ipv4/tcp_keepalive_time
  - TCP_MAXRT: set the maximum retransmission attempt time
IP Level Options

- When `level` is `IPPROTO_IP`
- Options include:
  - `IP_TTL`: time to live for unicast IP packets
  - Others will be covered in lecture 7

<table>
<thead>
<tr>
<th>Option</th>
<th>setsockopt()</th>
<th>getsockopt()</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_MULTICAST_LOOP</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IP_MULTICAST_TTL</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IP_MULTICAST_IF</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IP_ADD_MEMBERSHIP</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>IP_DROP_MEMBERSHIP</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
The `shutdown()` Function

- **Close communication without closing the socket**

UNIX:

```c
int shutdown(int sock, int how);
```

how values:

- 1 or `SD_SEND`
- 0 or `SD_RECEIVE`
- 2 or `SD_BOTH`

Returns 0 if successful or -1 on SOCKET_ERROR
Graceful Disconnect Procedure

First Side

- `shutdown(as, SD_SEND)`
- `recv(as, etc) = 0, or FD_READ`
- `close(as)`

Second Side

- `FIN`
- `ACK`
- `FIN`
- `ACK`
- `recv(bs, etc) = 0, or FD_READ`
- `shutdown(bs, SD_SEND)`
- `close(bs)`
The SO_LINGER Option

- By default, close() returns immediately
- The SO_LINGER option allows the user to over-ride this default.

```c
struct linger {
    int l_onoff;     // 0=off, nonzero=on
    int l_linger;    // linger time, probably in seconds
};
```
References

*Stevens, Unix Network Programming.*

Unix man pages

http://users.actcom.co.il/~choo/lupg/tutorials/signals/signals-programming.html

http://www.tech-faq.com/unix-signals.shtml

http://www.math.ups.edu/~brichards/Classes/325/Docs/signals.html

**Ack:** David Lowenthal  The University of Georgia
http://www.cs.uga.edu/~dkl/
(some of the slides here are from his course)