Lecture 4
Interprocess Communication

2 September, 2011
why IPC?
message passing
shared memory
UNIX IPC
signal

event notification mechanism for process or thread
#include <signal.h>

: 

// pid is the process ID to send the // signal to // sig is the ID of the signal.

kill(pid, sig);

kill(pid, SIGTERM);
at the receiver, default actions are defined for each signals
but reaction to most signals can be customized
C
function pointers
// function taking in a char * and returning an int.

int foo(char *name) { ... }  

// declare a function pointer

// initialize a function pointer

// call the function
return type \((\ast \text{ var name})(\text{ arg type})\)
function pointers can be:
(i) passed into function,
(ii) returned
(iii) defined as new type
```c
#include <signal.h>

void my_handler(int sig) { return; }

void (*prev_handler)(int);

prev_handler = signal(SIGTERM, my_handler);

// restore back original handler
signal(SIGTERM, prev_handler);
```
execution order

void my_handler
(int sig)
{
    kill(pid, SIGTERM)
}
signal in Linux

PCB

pending signal queue

signal handler table
CPU

- frame pointer
- stack pointer
- program counter
- program status word

Memory

- stack
  - return address
  - saved frame pointer
  - local variables
- data
- code
- signal handler
demo
race conditions
mutual exclusion

only one process can access a shared resource at a time
critical region
processes typically alternate between critical and non-critical region.
while (1)

text()  
critical_region
leave()  
noncritical_region
how to implement enter() and leave()
lock variable
enter()
   while (lock);
lock = 1;

leave()
lock = 0;
process A

while (lock);

lock = 1;

process B

while (lock);

lock = 1;
interrupts
enter()  
disable interrupt

leave()  
enable interrupt
works for single CPU

reduce responsiveness of the system

may hang the system if critical region is buggy
Peterson’s Algorithm
variables: 

interested[x] == 1 iff x is interested in entering the critical region.

turn == x if x can enter the critical region
process A

enter()
    interested[A] = 1
    turn = B
while (turn == B && interested[B]);

leave()
    interested[A] = 0
process B

enter()
interested[B] = 1
turn = A
while (turn == A && interested[A]);

leave()
interested[B] = 0
process A

\[ i[A] = 1 \]
\[ t = B \]

while (t is B && i[A]);

critical region

while (t is B && i[B]);

i[A] = 0

process B

\[ i[B] = 1 \]
\[ t = A \]

while (t is A && i[A]);
process A

i [A] = 1

t = B

while (t is B && i [B]);

process B

i [B] = 1

t = A

while (t is A && i [A]);
atomic instructions
TSL R, lock

in one atomic step, copy lock to R & set lock to 1
\textbf{enter()} \\
TSL \ R, \ lock \\
CMP \ R, \ #0 \\
JNE \ enter \\
RET \\

\textbf{leave()} \\
MOV \ lock, \ #0
test&set(lock)

return lock and set lock to 1
```
enter()

while (test&set(lock));

leave()

lock = 0;
```
XCHG R, lock

in one atomic step, swap values of two locations
**enter()**

MOV R, #1  
XCHG R, lock  
CMP R, #0  
JNE enter  
RET

**leave()**

MOV lock, #0
while (1)
    enter()
critical_region
leave()
noncritical_region
busy waiting
vs.
sleep/wake
while (1)
  if (lock) sleep(A)
  critical_region
  if (B is sleeping)
    wake(B)
  noncritical_region
while (1)
  if (lock) sleep(B)
  critical_region
  if (A) is sleeping
  wake(A)
  noncritical_region
the
producer-consumer problem
while (1)

if (buffer is full)
  sleep

if (buffer is empty)
  produce
  wake up consumer

else
  produce
while (1)
  if (buffer is empty)
    sleep
  if (buffer is full)
    consume
    wake up producer
  else
    consume
while (1)
    if (buffer is empty)
        sleep
    if (buffer is full)
        consume
    wake up consumer

    sleep
    if (buffer is full)
        consume
    wake up producer