CS 33 Week 7

Section 1G, Spring 2015
Prof. Eggert (TA: Eric Kim)
v1.0

Announcements

- Lab 3 was due Wednesday ("smashing" lab)
  - May be on midterm 2!
- HW 5 out!
  - Due: May 29th (2 weeks from now)
- Midterm 2 on Tuesday!
  - Open book, open notes

Overview

- Concurrency
  - Process-level, multiplexing, thread-level
- Synchronization
  - Semaphores, Mutexes
- MT 2 Review

Motivation

- Why do we care about concurrency?

  Primarily: Performance!

  To take advantage of multiple cores, run code in parallel.

    (We've seen this already in Instruction-Level Parallelism, such as pipelining)

Scenario

- We have a problem that can be easily broken up into separate "jobs".
- Goal: efficiently execute all jobs.

Concurrency: Processes

- Simple idea: create a separate process for each job.

Processes

- A process is an executing program
- Linux: Use 'top' or 'ps' to view processes

Each process has a Process ID (PID).

```
[ericki@lnxsrv04 ~]$ ps -u ericki
  PID TTY          TIME CMD
 7040 ?        00:00:00 sshd
 7042 pts/27   00:00:00 bash
 7116 pts/27   00:00:00 emacs
26762 ?        00:00:00 sshd
26764 pts/19   00:00:00 bash
26792 pts/19   00:00:00 ps
```

Processes in C: fork()

- In C, can create a new process with fork()

```c
#include <stdio.h>
#include <sys/types.h>

int main() {
  pid_t pid = fork();
  int val = 0;
  if (pid == 0) {
    printf("In child process! pid was: %d\n", pid);
    val = 7;
  } else {
    printf("In parent process! (Child's pid is: %d)\n", pid);
    val = 42;
  }
  printf("  [val=%d] Exiting.\n", val);
  return 0;
}
```

```c
#include <stdio.h>
#include <sys/types.h>

int main() {
  pid_t pid = fork();
  int val = 0;
  if (pid == 0) {
    printf("In child process! pid was: %d\n", pid);
    val = 7;
  } else {
    printf("In parent process! (Child's pid is: %d)\n", pid);
    val = 42;
  }
  printf("  [val=%d] Exiting.\n", val);
  return 0;
}
```

```
$ gcc -o exfork exfork.c
$ ./exfork
In child process! pid was: 0
  [val=7] Exiting.
In parent process! (Child's pid is: 17042)
  [pid=42] Exiting.
```

Question: Can this program output other printout orders?

fork() properties

- For most part, child and parent processes are separate
- Separate memory/address space, registers, etc.
- Note: Child inherits parent's open file descriptors!

```c
#include <stdio.h>
#include <sys/types.h>
void disp_file(FILE* f, int val) {
  char line[100];
  fgets(line, 99, f);
  line[100] = 0;
  printf("[val=%d] File contents: %s\n", val, line);
}
int main() {
  FILE* f = fopen("myfile.txt", "r");
  pid_t pid = fork();
  int val = 0;
  if (pid == 0) {
    printf("In child process! pid was: %d\n", pid);
    val = 7;
  } else {
    printf("In parent process! (Child's pid is: %d)\n", pid);
    val = 42;
  }
  disp_file(f, val);
  return 0;
}
```

```
$ gcc -o exfork2 exfork2.c
$ ./exfork2
```

Question: What gets output?

Answer:

```
In parent process! (Child's pid
is: 18063)
In child process! pid was: 0
[val=42] File contents: badwolf

[val=7] File contents: @
```

Huh?!

```c
#include <stdio.h>
#include <sys/types.h>
void disp_file(FILE* f, int val) {
  char line[100];
  fgets(line, 99, f);
  line[100] = 0;
  printf("[val=%d] File contents: %s\n", val, line);
}
int main() {
  FILE* f = fopen("myfile.txt", "r");
  pid_t pid = fork();
  int val = 0;
  if (pid == 0) {
    printf("In child process! pid was: %d\n", pid);
    val = 7;
  } else {
    printf("In parent process! (Child's pid is: %d)\n", pid);
    val = 42;
  }
  disp_file(f, val);
  return 0;
}
```

Child and parent *share* file descriptor tables, including seek locations!

Fix: Add a line that resets pointer to beginning of file.

```c
void disp_file(FILE* f, int val) {
  char line[100];
  fseek(f, 0, SEEK_SET);
  fgets(line, 99, f);
  line[100] = 0;
  printf("[val=%d] File contents: %s\n", val, line);
}
```

Processes: Pros/Cons

Pros

Simple to code. Processes can't interfere with each other (separate memory/stack, etc.).
Utilizes *OS's* process scheduling system to maximize concurrency (less work for us!)

Cons

Difficult for processes to communicate. Can still be done, but is somewhat expensive.
Large overhead to spawning new processes - if each job is fairly quick, then might simply be *faster* to do jobs in single process!

Multiplexing

- Idea: Only use *one* process to perform multiple jobs.
- "Take turns" executing each job.

Multiplexing

- Typical ingredients:
  - select(), FD_SET, FD_ISSET, FD_CLEAR, etc.

Terminology: "blocking"

A function fn() is called "blocking" if it:

Halts execution of the current thread while fn() is running.

```
int main() {
  char* dataset = read_dataset(); // blocking
  float mn = compute_mean(dataset); // blocking
  printf("  Mean is: %f\n", mn);
  return 0;
}
```

Non-blocking

```
int main() {
  struct waitstruct* rval = read_dataset(); // non-blocking
  while (rval->status == 0)
    sleep(1);
  float mn = compute_mean(rval->dataset); // blocking
  printf("  Mean is: %f\n", mn);
  return 0;
}
```

Multiplexing: Pros/Cons

Pros:

Shared memory, easy to communicate information between each job.

Cons:

Only one process runs at a time! No performance gains from parallelism here.
Your program must be structured in a particular way to use this approach.

Threading

- "Lightweight" method of concurrency
- Similar to processes:
  - Main thread spawns new threads
- Similar to multiplexing:
  - All threads share same memory space

Best of both worlds?

Note: gcc and threads

- To use pthreads in your C programs, add the "-lpthread" option to gcc command:

```
$ gcc -o mythread mythread.c -lpthread
```

Note: goes at end!

Example: C

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
void* threadjob(void *arg) {
  int* val = ((int*) arg);
  *val = 7;
  printf("  Thread finished.\n");
  return NULL;
}
int main() {
  pthread_t pth;
  int myval = 42;
  printf("  (1) myval is: %d\n", myval);
  pthread_create(&pth, NULL, threadjob, &myval);
  pthread_join(pth, NULL);
  printf("  (2) myval is: %d\n", myval);
  return 0;
}
```

Question: What does program output?

Answer:

(1) myval is: 42
Thread finished.
(2) myval is: 7

Questions: Are there other possible outputs?

Answer:
No! pthread_join() enforces a consistency.

Example: C

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
void* threadjob(void *arg) {
  int* val = ((int*) arg);
  *val = 7;
  printf("  Thread finished.\n");
  return NULL;
}
int main() {
  pthread_t pth;
  int myval = 42;
  printf("  (1) myval is: %d\n", myval);
  pthread_create(&pth, NULL, threadjob, &myval);

  printf("  (2) myval is: %d\n", myval);
  return 0;
}
```

Question: What are the possible outputs of the program?

Answer:

(1) myval is: 42
Thread finished.
(2) myval is: 7

(1) myval is: 42
(2) myval is: 42
Thread finished.

Suppose we removed that pthread_join() call...

Threading: Pros/Cons

Pros:

Shared memory, easy to communicate information between threads.
Much less overhead than processes.
Performance improves due to parallel execution!

Cons:

Programmer must be careful about threads reading/writing to **shared memory**. Concurrency bugs may occur if not careful.

Synchronization

- Threads allow a lightweight way to perform concurrency with shared variables
  - "With great power, comes great responsibility…"

Concurrency bugs: Program *sometimes* works, data is *sometimes* wrong, crashes *sometimes*...

Must carefully govern access to shared variables!

Semaphores

- Popular synchronization primitive
- A counter
    - Semaphores are created with a fixed number of N "tickets"
- If N=1, then is a <u>binary</u> semaphore
    - aka **"mutex"**

Operations

"I want access!"

```
void P(sem_t* s)
```

Aka "sem_wait(s)". Decrements semaphore by 1 if possible. If not possible, then wait until possible, ie another thread calls V().

"I'm done!"

```
void V(sem_t* s)
```

Aka "sem_post(s)", or "sem_wakeup(s)". Increments semaphore by 1. If there are threads waiting to increment s, then this wakes up one of the threads, allowing the thread to continue running.

Example: Bounded Shared Buffer

- "Producer/Consumer" Scenario
- Application: Playing a video
  - Video decoder is constantly decoding frames and placing them in a buffer (ie each frame is an image)
  - Video player is constantly taking images from the buffer, and displaying them on the screen
- Guard access to buffer carefully

```c
typedef struct {
  int *buf; /* Buffer array */
  int n; /* nb slots in buffer */
  int front; /* buf[(front+1)%n] is first item */
  int rear; /* buf[rear%n] is last item */
  sem_t mutex; sem_t slots; sem_t items;
} sbuf_t;
```

First attempt

```c
void insert(sbuf_t* sp, int item) {
  P(&sp->mutex);
  sp->buf[(++sp->rear)%(sp->n)] = item;
  V(&sp->mutex);
}
```

```c
int remove(sbuf_t *sp) {
  int x;
  P(&sp->mutex);
  x=sp->buf[(++sp->front)%(sp->n)];
  V(&sp->mutex);
  return x;
}
```

Question: What's wrong with this implementation? Any synchronization bugs?

Answer: Insert can overwrite existing entries! No synchronization bugs though.

Second attempt

```c
void insert(sbuf_t* sp, int item) {
  P(&sp->slots);
  P(&sp->mutex);
  sp->buf[(++sp->rear)%(sp->n)] = item;
  V(&sp->mutex);
  V(&sp->items);
}
```

```c
int remove(sbuf_t *sp) {
  int x;
  P(&sp->items);
  P(&sp->mutex);
  x=sp->buf[(++sp->front)%(sp->n)];
  V(&sp->mutex);
  V(&sp->slots);
  return x;
}
```

Question: Anything wrong with this implementation? Any synch bugs?

Answer: Nope!

MT2 Review

- Floating Point
- Program Optimization
- (Basic) Processor Architecture
- Instruction Level Parallelism
- Concurrency
- Synchronization
- *MT 1 topics*

Q: Optimization

One compiler optimization makes use of the associative property to break data dependencies:

`acc=(acc*data[i])*data[i+1]` **vs** `acc=acc*(data[i]*data[i+1])`

Would an optimization based on the **commutative** property ever speed up a program? If so, give a scenario where a speedup would occur due to the commutative property, and explain why. If not, explain why not.

`a + b = b + a` ← Commutative Property

A: Optimization

Applying the commutative property will **not** speed up execution.

The processor is *already* utilizing the commutative property. When the processor is determining which micro-instructions to run, it will perform operations out-of-order to maximize performance. For instance, if an Adder functional unit is idle, the processor will send any (independent) pending addition executions to the Adder, regardless of order.

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i, *ptr;
  for (i=0; i<N; i++) {
    ptr = Malloc(sizeof(int)); *ptr = i;
    Pthread_create(&tid[i],NULL,fn,ptr);
  }
  for (i=0; i<N; i++)
    Pthread_join(tid[i], NULL);
  exit(0);
}
```

```
void *fn(void *vargp) {
  int myid = *((int *)vargp);
  Free(vargp);
  printf("%d\n",myid);
  return NULL;
}
```

Question: Are there any race conditions in this code?

Answer: Nope. Careful use of Malloc/Free prevents possible bugs.

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i, *ptr;
  for (i=0; i<N; i++) {
    ptr = Malloc(sizeof(int)); *ptr = i;
    Pthread_create(&tid[i],NULL,fn,ptr);
  }
  for (i=0; i<N; i++)
    Pthread_join(tid[i], NULL);
  exit(0);
}
```

```
void *fn(void *vargp) {
  int myid = *((int *)vargp);
  Free(vargp);
  process(myid);
  return NULL;
}
```

Question: Outline an approach to avoid race conditions that doesn't use Malloc/Free. What are the advantages/disadvantages of your approach?

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i, *ptr;
  for (i=0; i<N; i++) {
    Pthread_create(&tid[i],NULL,fn,(void*)i);
  }
  for (i=0; i<N; i++)
    Pthread_join(tid[i], NULL);
  exit(0);
}
```

```
void *fn(void *vargp) {
  int myid = (int) vargp;
  process(myid);
  return NULL;
}
```

Answer: Simply pass in the int directly!
Pro: No added overhead due to malloc/free.
Con: Assumes that pointer datatype is at least bigger than size of int. May not be true on all systems.

Q: Semaphores

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Question: What are the possible outputs of this program? Explain your answer.

Q: Semaphores

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Answer: Nothing - this program will always deadlock!

Q: Semaphores

```c
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```c
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Question: Now, what are the possible outputs of the program? Can deadlock still happen?

Q: Semaphores

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Answer: Either "Here: 1" -> "Here: 2", or vice-versa. Dead lock can't happen anymore.

Q: More semaphores

Thread 1:

```
P(&s)
P(&t)
do_work();
V(&t)
V(&s)
```

Thread 2:

```
P(&t)
P(&s)
do_work();
V(&s)
V(&t);
```

```
sem_t t;    // N = 1
sem_t s;    // N = 1
```

Will this always deadlock? Sometimes deadlock? Never deadlock? Show execution order for possible cases.

Q: More semaphores

Thread 1:

```
P(&s)
P(&t)
do_work();
V(&t)
V(&s)
```

Thread 2:

```
P(&t)
P(&s)
do_work();
V(&s)
V(&t);
```

```
sem_t t;    // N = 1
sem_t s;    // N = 1
```

Deadlock:

```
T1        T2
P(&s)
          P(&t)
          P(&s)
P(&t)
  T1,T2 stuck!
```

OK:

```
T1        T2
P(&s)
P(&t)
do_work()
V(&t)
V(&s)
          P(&t)
          P(&s)
          ...
```

Q: More semaphores

Thread 1:
```
P(&t)
P(&s)
do_work();
V(&s)
```

Thread 2:
```
P(&t)
P(&s)
do_work();
V(&s)
V(&t);
```

```
sem_t t;    // N = 1
sem_t s;    // N = 1
```

Will this always deadlock? Sometimes deadlock? Never deadlock? Show execution order for possible cases.

Q: More semaphores

Thread 1:
```
P(&t)
P(&s)
do_work();
V(&s)
```

Thread 2:
```
P(&t)
P(&s)
do_work();
V(&s)
V(&t);
```

```
sem_t t;    // N = 1
sem_t s;    // N = 1
```

OK:
```
T1       T2
        P(&t)
        P(&s)
        do_work()
        V(&s)
        V(&t)
P(&t)
...
```

Deadlock:
```
T1              T2
P(&t)
P(&s)
do_work()
V(&s)
                P(&t)
      T2 is stuck!
```

Q: Volatility

Louis fell asleep during lecture, and woke up to Prof. Eggert saying "we must use the `volatile` keyword to avoid the compiler optimizing away access to this variable".

Louis thinks: "That's silly. I want my code to be as fast as possible, so I will never use `volatile` in my code."

Give a scenario in which Louis' code may produce incorrect results/behavior.

Q: Volatility

```c
int status; // asynch-modified by hardware
void fn1() {
  status = 0; // reset status var
  while (status == 0)
    sleep(1); // wait for non-zero status
  handle_status(status);
}
```

Since no other visible code can modify status, an aggressive compiler *may* optimize fn1() to be:

```c
status = 0;
while (true)
  sleep(1);
...
```

However, an "outside" source, ie hardware, may modify status (say, when the user presses a key).

Q: Canaries

8. Your friend implements his stack-protector as follows:

```
static int canary_safe;
void mygets(char *buff) {
    int canary = rand();
    canary_safe = canary;
    gets(buff);
    if (canary != canary_safe) {
        perror("Smash detched!")
    }
}
```

Is he safe?

A: Canaries

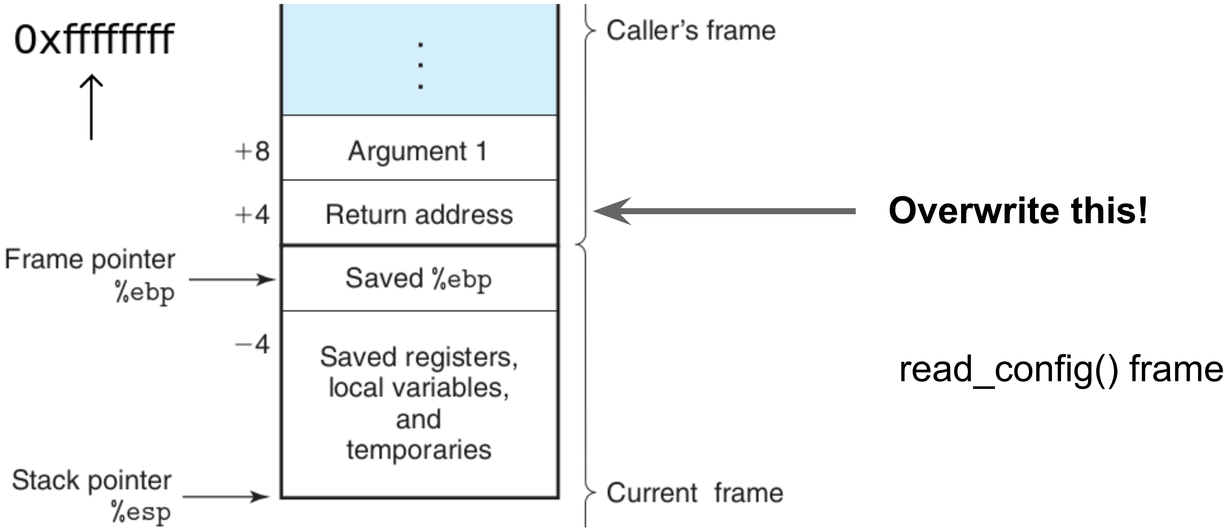
No! buf doesn't live in this stack frame, yet the canary lives in the stack frame of mygets(). Thus, we can overwrite values in the *caller's* stack frame with impunity, ie the caller's saved-eip.

Lab 3 Q9 Review ("smashing" lab)

- Several approaches to exploit
  - Assume NX-bit is disabled. Must bypass ASLR.
  - Assume ASLR is disabled. Must bypass NX-bit.
  - Assume neither is disabled.

Overwriting saved-eip

In all attacks, we will be overwriting the saved return address to point to something *we* control.



Approach 1: Disable NX bit

With NX-bit disabled, stack memory is executable. Inject x86 code to delete file.

x86 opcodes

```
(gdb) disas /r unlink
Dump of assembler code for function unlink:
=> 0x00a51d20 <+0>: 89 da    mov    %ebx,%edx
   0x00a51d22 <+2>: 8b 5c 24 04 mov    0x4(%esp),%ebx
   0x00a51d26 <+6>: b8 0a 00 00 00  mov    $0xa,%eax
   0x00a51d2b <+11>:    65 ff 15 10 00 00 00    call   *%gs:
0x10
```

Each x86 instruction is actually represented as hex bytes ("opcodes")

```
mov %ebx,%edx -> 0x89da
mov 0x4(%esp),%ebx -> 0x8b5c2404
```

Inject the opcodes to the stack.

Which opcodes to inject? Say I stepped into the "`call  *%gs:0x10`" line in gdb:

```
=> 0x00110420 <+0>: 51   push   %ecx
   0x00110421 <+1>: 52   push   %edx
   0x00110422 <+2>: 55   push   %ebp
   0x00110423 <+3>: 89 e5    mov    %esp,%ebp
   0x00110425 <+5>: 0f 34    sysenter
```

sysenter: Asks system to do a system call.
%eax: Which syscall to do (0xa: unlink)
%ebx: First argument to syscall (ie char* filename)

(Or, can use "int 0x80" to make system call.)

Approach 1: Steps

Two steps:

(1) Inject relevant x86 code to stack that deletes "target.txt"

How? Write opcode bytes to config file!

(2) Compute the address of the "target.txt" string.

Approach 1: Steps

(2) Compute the address of the "target.txt" string.

Hard way: guess the address of start of string. With ASLR, there can ~2^20 choices...ouch.

Easier way: use relative addressing!

```
leal $0x42(%esp), %ebx
```

Use gdb to determine exact relative offset from %esp to your "target.txt" string.

Approach 1: NOP sled

Recall: We have to deal with ASLR. How to guess start address of our x86 opcodes?

NOP-sled!

```
0x90 90 90 90 90 ... 90 90 <REAL CODE>
```

Guess *any* of these addresses, and win!

Attack won't always work, due to ASLR.

But - if you run it enough times, you'll get a win.

Approach 2: Disable ASLR

Can't inject x86 code onto stack, due to NX bit.

Instead: trick program into calling the unlink() function!

Challenge: set up stack memory so that we pass in correct args to unlink().

(1) Overwrite read_config()'s saved-eip to point to unlink.

```
(gdb) p/x &unlink
0x00a51d20
```

Note: This address may change if you change machines.

(2) Write address of "target.txt" to correct stack location for unlink to use.

Where on stack?

```
(gdb) disas /r unlink
Dump of assembler code for function unlink:
=> 0x00a51d20 <+0>: 89 da    mov    %ebx,%edx
   0x00a51d22 <+2>: 8b 5c 24 04 mov    0x4(%esp),%ebx
   0x00a51d26 <+6>: b8 0a 00 00 00  mov    $0xa,%eax
   0x00a51d2b <+11>:    65 ff 15 10 00 00 00    call   *%gs:
0x10
```

Approach 3: Hard mode

Bypass NX-bit: Overwrite saved return-addr to &unlink

Bypass ASLR: Use a "NOP-sled", but for file names!

```
./././././././././././././target.txt
```

As long as we land on a '.', we win!

Approach 3: Hard mode

Obstacle: Unix defines a max filename length of 4096 bytes.

So, can't have too many repeated "./". Restricts our ability to improve chances of guessing successfully.

Question: How to bypass this (annoying) max filename length?

```
./././././././././././target.txt
```