CS 162: Operating Systems and Systems Programming

Lecture 8: Deadlock, Introduction to I/O

July 8, 2019
Instructor: Jack Kolb
https://cs162.eecs.berkeley.edu
Logistics

- Project 1 Milestone Due Wednesday
- Homework 1 Due Friday
- Project/HW “Party” Friday 3-5PM
  - Wozniak Lounge, Soda Hall
  - Full course staff will be there to help you
Recall: Producer/Consumer

```
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```
Recall: Producer/Consumer

Producer(item) {
    mutex.P();
    emptySlots.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}

Deadlock
Recall: Why deadlock?

Assume we currently have a full buffer

\[\begin{align*}
\text{mutex} &= 1 \quad \text{emptySlots} = 0 \quad \text{fullSlots} = N \\
\text{mutex} &= 0 \quad \text{emptySlots} = 0 \quad \text{fullSlots} = N \\
\text{mutex} &= 0 \quad \text{emptySlots} = 0 \quad \text{fullSlots} = N-1
\end{align*}\]

\begin{align*}
\text{Producer}(\text{item}) \{ & \\
\quad & \text{mutex.P();} \\
\quad & \text{emptySlots.P();} \\
\quad & \text{// Stuck!}
\}
\end{align*}

\begin{align*}
\text{Consumer()} \{ & \\
\quad & \text{fullSlots.P();} \\
\quad & \text{mutex.P();} \\
\quad & \text{// Stuck!}
\}
\end{align*}
Recall: Condition Variables

• A queue of threads waiting inside a critical section

• Operations:
  • `wait(&lock)`: Atomically release lock and go to sleep. Re-acquire the lock before returning.
  • `signal()`: Wake up on waiting thread (if there is one)
  • `broadcast()`: Wake up all waiting threads

• **Rule:** Hold lock when using a condition variable
Recall: Monitors

- **Lock**: protects access to shared data
- **Condition Variables**: queue of threads waiting for something to become true inside critical section.
Recall: Hoare vs. Mesa Semantics

while (queue.isEmpty()) {
    dataready.wait(&lock);
}

if (queue.isEmpty()) {
    dataready.wait(&lock);
}

Mesa Monitors
• A signaled thread becomes ready again
• But another thread may acquire lock and enter monitor before signaled thread runs
• Need to re-check condition upon wakeup

Hoare Monitors
• A signaled thread acquires the lock and runs immediately
• No other thread can “sneak in” and change the monitor’s state
• Described in many textbooks
Recall: Reader/Writer Sync.

- Readers can access when no writers
- Writers can access when no readers and no other writers

- A lock will satisfy these requirements
  - But we want to allow **multiple readers**
  - Better efficiency
Recall: RW Sync. With a Monitor

Reader() {
    Wait until no active or waiting writers
    Access database
    Maybe wake up a writer
}

Writer() {
    Wait until no active readers or writers
    Access database
    If waiting writer, wake it up;
    Otherwise, wakeup readers;
}

int activeReaders;
int activeWriters;
int waitingWriters
Definition: Resources

- Entities needed by threads to do work
  - CPU Time
  - Disk Space
  - Memory

- Preemptible vs. Non-Preemptible
  - Can OS safely take it away from a thread?
  - Yes: CPU
  - No: Disk space, printer, right to enter a critical section

- Shareable vs Exclusive
**Starvation vs. Deadlock**

- **Starvation:** A thread waits indefinitely
  - Ex: Low-priority threads, when other high-priority threads are always present

- **Deadlock:** Circular waiting for resources
  - A case of starvation
Deadlock with Locks

Thread A
x.Acquire();
y.Acquire();
...
y.Release();
x.Release();

Thread B
y.Acquire();
x.Acquire();
...
x.Release();
y.Release();

Nondeterministic Deadlock
Deadlock with Locks: “Lucky” Case

Thread A
x.Acquire();
y.Acquire();
...
y.Release();
x.Release();

Thread B

y.Acquire();
x.Acquire();
...
x.Release();
y.Release();

Sometimes schedule won’t trigger deadlock
Deadlock with Locks: Unlucky Case

Thread A
x.Acquire();
y.acquire(); <stalled>
<unreachable>
...
y.Release();
x.Release();

Thread B
y.Acquire();
x.Acquire(); <stalled>
<unreachable>
...
x.Release();
y.Release();
## Deadlock with Space

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
</tbody>
</table>

If only 2 MB of space, we get same deadlock situation
Real Example: Single-Lane Bridge Crossing

- Each segment of road is a resource
  - Car must obtain next segment to progress

- **Deadlock**: Two cars in opposite directions meet in middle

- Resolving deadlock: “Preempt” road segment, force one car to back up

- **Starvation** (not deadlock): Eastbound traffic doesn’t stop for westbound traffic
The Dining Philosophers Problem

• Five chopsticks, five philosophers
  • Goal: Grab two chopsticks to eat

• **Deadlock** if they all grab chopstick to their right

• **Fix:** Never take the last chopstick if no one will have two afterwards
Formalizing: Four Requirements for Deadlock to Occur

1. **Mutual Exclusion**: One thread at a time can use a resource (not shareable)

2. **Hold and Wait**: Thread holding a resource waits to acquire another resource

3. **No Preemption**: Resources are released voluntarily, threads can’t steal instead of waiting

4. **Circular Wait**: There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads such that:
   - \( T_1 \) is waiting for a resource held by \( T_2 \)
   - \( T_2 \) is waiting for a resource held by \( T_3 \)
   - \( \ldots \)
   - \( T_n \) is waiting for a resource held by \( T_1 \)
Resource-Allocation Graph

• System Model
  • A set of Threads $T_1, T_2, \ldots, T_n$
  • Resource types $R_1, R_2, \ldots, R_m$
    
    \textit{CPU cycles, memory space, I/O devices}
  • Each resource type $R_i$ has $W_i$ instances
  • Each thread utilizes a resource as follows:
    • \text{Request()} \ / \ \text{Use()} \ / \ \text{Release()}

• Resource-Allocation Graph:
  • $V$ is partitioned into two types:
    • $T = \{T_1, T_2, \ldots, T_n\}$, the set of threads in the system.
    • $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  • Request edge – directed edge $T_i \rightarrow R_j$
  • Assignment edge – directed edge $R_j \rightarrow T_i
Resource-Allocation Graph Examples

$T_i \rightarrow R_j$: Thread requests a resource

$R_j \rightarrow T_i$: Resource is assigned to a thread

Does this graph represent a deadlock?

1. $T_3$ Finishes, releases $R_4$, $R_2$
2. $T_2$ Finishes, releases $R_1$, $R_2$, $R_3$
3. $T_1$ Finishes, releases $R_1$, $R_3$
Resource-Allocation Graph Examples

$T_i \rightarrow R_j$: Thread requests a resource

$R_j \rightarrow T_i$: Resource is assigned to a thread

What about these graphs?

Yes

No
Options for Handling Deadlocks

• **Recover** by preempping resources
  • Need to *detect* deadlock first
  • Resources must be preemptible, e.g., terminate thread

• **Prevent** by monitoring or design
  • Deny/don’t attempt resource acquisitions that *might* lead to a deadlock

• **Do Nothing**
  • Most operating systems do this
Deadlock Detection Algorithm

• If only one type of resource: Look for loops in resource allocation graph
• More general deadlock detection algorithm:
  • For each resource in the system, record:
    1. Fraction of resource currently free
    2. Fraction requested by each thread in the system
    3. Fraction already allocated to each thread in the system
  • Key Idea: Find a task that can finish on its own (all of its pending requests for resources can be satisfied)
    • Mark those task’s resources as free for future iterations (assume task finishes)
  • Repeat this until no remaining task can have its requests satisfied
  • Unfinished threads remaining? => Deadlock
Deadlock Detection Algorithm

// Assign each resource an index \( i \)
\[ \text{Avail}[i] = \text{Free}[i] \text{ for all } i \]
Add all nodes to UNFINISHED

do {
    done = true;
    for each node in UNFINISHED {
        if (Requested_{node}[i] <= \text{Avail}[i] \text{ for all } i) {
            remove node from UNFINISHED
            \text{Avail}[i] += Allocated_{node}[i] \text{ for all } i
            done = false
        }
    }
}
} until(done)

Nodes left in UNFINISHED => Deadlocked
Deadlock Detected: Options?

- Terminate thread, force it to give up resources
  - Ex: Kick a philosopher out of the restaurant
  - Not always possible

- Preempt resources without terminating thread
  - Not always correct

- Revert actions of deadlocked threads
  - Databases do this (transaction rollback) – not easy for an operating system
  - Ex: Back up cars on single-lane bridge to clear a path
Recall: Four Requirements for Deadlock to Occur

1. **Mutual Exclusion**: One thread at a time can use a resource (not shareable)

2. **Hold and Wait**: Thread holding a resource waits to acquire another resource

3. **No Preemption**: Resources are released voluntarily, threads can’t steal instead of waiting

4. **Circular Wait**: There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads such that:
   - \( T_1 \) is waiting for a resource held by \( T_2 \)
   - \( T_2 \) is waiting for a resource held by \( T_3 \)
   - \( \ldots \)
   - \( T_n \) is waiting for a resource held by \( T_1 \)
Deadlock Prevention

**Infinite** resources
- Or just large enough capacity so that no one runs out
- Ex: Virtual memory gives the illusion of infinite space

Avoid **sharing** of resources

Don’t allow **waiting**
- Phone system: busy signal
- Networking: transmission collision – back off and retry later

 Attacks **mutual** exclusion

 Attacks **hold** and **wait / no preemption**
Deadlock Prevention

• Get resources in **consistent order**
  • Prevents cycles
  • Ex: Acquire locks in order by name

• Request **all necessary resources at once**
  • Get two chopsticks at once or not at all
Prevention: Maximum Resources

- Relax the all resources at once solution
- Extra information:
  - **Maximum resources** a thread might need in its lifetime
- Strawman solutions:
  - Reserve maximum resources when thread starts
  - Reserve maximum resource when any resource is requested
- Something less conservative/wasteful?
Prevention: Banker’s Algorithm

• Still need to know each thread’s maximum resource requirements

• Block a thread from getting a resource unless system would remain in a safe state
  • There is an ordering of threads $T_1, T_2, \ldots, T_n$ such that running $T_1$ to completion, $T_2$ to completion, and so on would not deadlock

• Use modified version of deadlock detection algo.
  • Pretend each resource request has been satisfied
  • Pretend each running thread is requesting its maximum
  • If no deadlock, then an ordering exists
Banker’s Algorithm

// Assign each resource an index i
Avail[i] = Free[i] for all i
Add all nodes to UNFINISHED

do {
    done = true;
    for each node in UNFINISHED {
        if (Max_node[i] - Alloc_Node[i] <= Avail[i] for all i) {
            remove node from UNFINISHED
            Avail[i] += Allocated_node[i] for all i
            done = false
        }
    }
} until(done)

Nodes left in UNFINISHED => Unsafe State
Banker’s Algorithm Example

• With dining philosophers, safe state:
  • Not acquiring the last chopstick or
  • Last chopstick but someone will have two chopsticks

• What if we needed $k$ chopsticks to eat? Block if:
  • Last Chopstick: No one would have $k$ chopsticks
  • Second to last: No one would be $k-1$ chopsticks
  • Third to last: No one would have $k-2$ chopsticks
  • And so on…
Break
POSIX I/O: Everything is a “File”

Identical interface for:
• Devices (terminals, printers, etc.)
• Regular files on disk
• Networking (sockets)
• Local interprocess communication (pipes, sockets)

Based on `open()`, `read()`, `write()`, and `close()`
POSIX I/O Design Patterns

• Open before use
  • Access control check, setup happens here

• Byte-oriented
  • Least common denominator
  • OS responsible for hiding the fact that real devices may not work this way (e.g. hard drive stores data in blocks)

• Explicit close
POSIX I/O: Kernel Buffering

• Reads are buffered
  • Part of making everything byte-oriented
  • Process is **blocked** while waiting for device
  • Let other processes run while gathering result

• Writes are buffered
  • Complete in background (more later on)
  • Return to user when data is “handed off” to kernel
Putting it together: web server

- **Server**

  - **Kernel**
    - request buffer
    - reply buffer
    - kernel copy from user buffer to network buffer
    - format outgoing packet and DMA
    - disk data (DMA)
    - disk request

  - **Network interface**
    - network socket read
    - kernel copy
    - network socket write
    - file read
    - socket write
    - socket read

- **Hardware**

  - Network interface
  - Disk interface

**Request**

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Request

Reply

Kernel buffer write
I/O & Storage Layers

Application / Service

High Level I/O
Low Level I/O
Syscall
File System
I/O Driver

Commands and Data Transfers

Disks, Flash, Controllers, DMA

streams
handles
registers
descriptors
The File System Abstraction

Regular File
- Named collection of data
- POSIX: Sequence of bytes
  - Could really be text, binary, serialized objects, etc.
- Also size, modification time, owner, access control info

Directory
- “Folder” containing files and other directories
- Hierarchical Naming: /home/oski/cs162
C stdio (High Level) File API: Streams

FILE* - stream, sequence of bytes and position

```c
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

<table>
<thead>
<tr>
<th>Mode Text</th>
<th>Binary</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>rb</td>
<td>Open existing file for reading</td>
</tr>
<tr>
<td>w</td>
<td>wb</td>
<td>Open for writing; created if does not exist</td>
</tr>
<tr>
<td>a</td>
<td>ab</td>
<td>Open for appending; created if does not exist</td>
</tr>
<tr>
<td>r+</td>
<td>rb+</td>
<td>Open existing file for reading &amp; writing.</td>
</tr>
<tr>
<td>w+</td>
<td>wb+</td>
<td>Open for reading &amp; writing; truncated to zero if exists, create otherwise</td>
</tr>
<tr>
<td>a+</td>
<td>ab+</td>
<td>Open for reading &amp; writing. Created if does not exist. Read from beginning, write as append</td>
</tr>
</tbody>
</table>
Connecting Processes and the Filesystem

• Each process has a *current working directory* (CWD)

• Absolute paths: /home/oski/cs162

• Relative paths
  • index.html, ./index.html (in CWD)
  • ../index.html (Parent of CWD)
  • ~, ~oski (home directory, *shell only*)
stdio Standard Streams

• Three predefined streams
  1. FILE* stdin – Normal source of input
  2. FILE* stdout – Normal source of output
  3. FILE* stderr – For error output

• All can be redirected
  • cat hello.txt | grep “World!”
  • cat’s stdout goes to grep’s stdin
C Streams: Read/Write

#include <stdio.h>

// character oriented
int fputc(int c, FILE *fp); // rtn c or EOF on err
int fputs(const char *s, FILE *fp); // rtn >0 or EOF
int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
    size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
    size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ...);
#include <stdio.h>

int main(void) {
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    int c;

    c = fgetc(input);
    while (c != EOF) {
        fputc(output, c);
        c = fgetc(input);
    }
    fclose(input);
    fclose(output);
}
What if we wanted block by block I/O?

```c
#include <stdio.h>

// character oriented
int fputc(int c, FILE *fp); // rtn c or EOF on err
int fputs(const char *s, FILE *fp); // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
    size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
    size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ...);
```
stdio Block-by-Block I/O

#include <stdio.h>
#define BUFFER_SIZE 1024
int main(void) {
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    char buffer[BUFFER_SIZE];
    size_t length;
    while (length > 0) {
        fwrite(buffer, length, sizeof(char), output);
        length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    }
    fclose(input);
    fclose(output);
}
stdio Block-by-Block I/O

#include <stdio.h>
#define BUFFER_SIZE 1024
int main(void) {
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    char buffer[BUFFER_SIZE];
    size_t length;
    length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    while (length > 0) {
        fwrite(buffer, length, sizeof(char), output);
        length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    }
    fclose(input);
    fclose(output);
}
Aside: Systems Programming

- Systems programmers are paranoid
- We should really be writing things like:
  ```c
  FILE* input = fopen(“input.txt”, “r”);
  if (input == NULL) {
      // Prints our string and error msg.
      perror(“Failed to open input file”)
  }
  ```

- Be thorough about checking return values
  - Want failures to be systematically caught and dealt with
C Streams: Positioning

```c
int fseek(FILE *stream, long int offset, int whence);
long int ftell (FILE *stream)
void rewind (FILE *stream)
```
I/O & Storage Layers

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

Streams

Handles

Registers

Descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA
POSIX Low-Level I/O

- Operations on file descriptors (represented as ints)
  - Per-process table pointing to open file descriptors
- Created by `open()`, removed by `close()`

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:
- Access modes (Rd,Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:
- User|Group|Other × R|W|X
#include <unistd.h>

STDIN_FILENO - macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

• Correspond to stdio.h’s stdin, stdout, stderr
• Don’t mix them!
C Low Level Operations

ssize_t read (int filedes, void *buffer, size_t maxsize)
- returns bytes read, 0 => EOF, -1 => error
 ssize_t write (int filedes, const void *buffer, size_t size)
- returns bytes written

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) – wait for i/o to finish
void sync (void) – wait for ALL to finish

• When write returns, data is on its way to disk (for regular files), but it may not actually be permanent
Low-Level I/O: Example

```c
#include <fcntl.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int main(void) {
    int input_fd = open("input.txt", O_RDONLY);
    int output_fd = open("output.txt", O_WRONLY);
    char buffer[BUFFER_SIZE];
    ssize_t length;
    length = read(input_fd, buffer, BUFFER_SIZE);
    while (length > 0) {
        write(output_fd, buffer, length);
        length = read(input_fd, buffer, BUFFER_SIZE);
    }
    close(input_fd);
    close(output_fd);
}
```
Low-Level I/O: Other Operations

- Operations specific to terminals, devices, networking, …
- Memory-Mapping Files
- File Locking
- Asynchronous I/O
- Duplicating descriptors
  - int dup2(int old, int new);
  - int dup(int old);
Streams vs. File Descriptors

• Streams are **buffered in user memory**:

  ```c
  printf("Beginning of line ");
  sleep(10); // sleep for 10 seconds
  printf("and end of line\n");
  ```

  Prints out **everything at once**

• Operations on file descriptors are **visible immediately**

  ```c
  write(STDOUT_FILENO, "Beginning of line ", 18);
  sleep(10);
  write("and end of line \n", 16);
  ```

  Outputs "Beginning of line" 10 seconds earlier
Why Buffer in Userspace? Overhead!

• Avoid system call overhead
  • Time to copy registers, transition to kernel mode, jump to system call handler, etc.

• Minimum syscall time: $\sim 100$ns of nanoseconds
  • Read/write a file byte by byte?
  • Max throughput of $\sim 10$MB/second
  • With `fgetc`? Keeps up with your SSD
Why Buffer in Userspace? Functionality.

- System call operations less capable
  - Simplifies operating system

- Example: No "read until new line" operation
  - Solution: Make a big read syscall, find first new line in userspace
  - Could simulate by one syscall per character, but we already know this is a bad idea
Summary

• Starvation (indefinite wait) vs. Deadlock (circular wait)

• Conditions for Deadlock
  1. Mutual Exclusion
  2. Hold and Wait
  3. No Preemption
  4. Circular Wait

• Dealing with Deadlock
  • Detect and recover (preemption, rollback)
  • Prevent by blocking on certain resource requests

• File I/O: open, read, write, seek, close
  • POSIX: Files are the universal interface