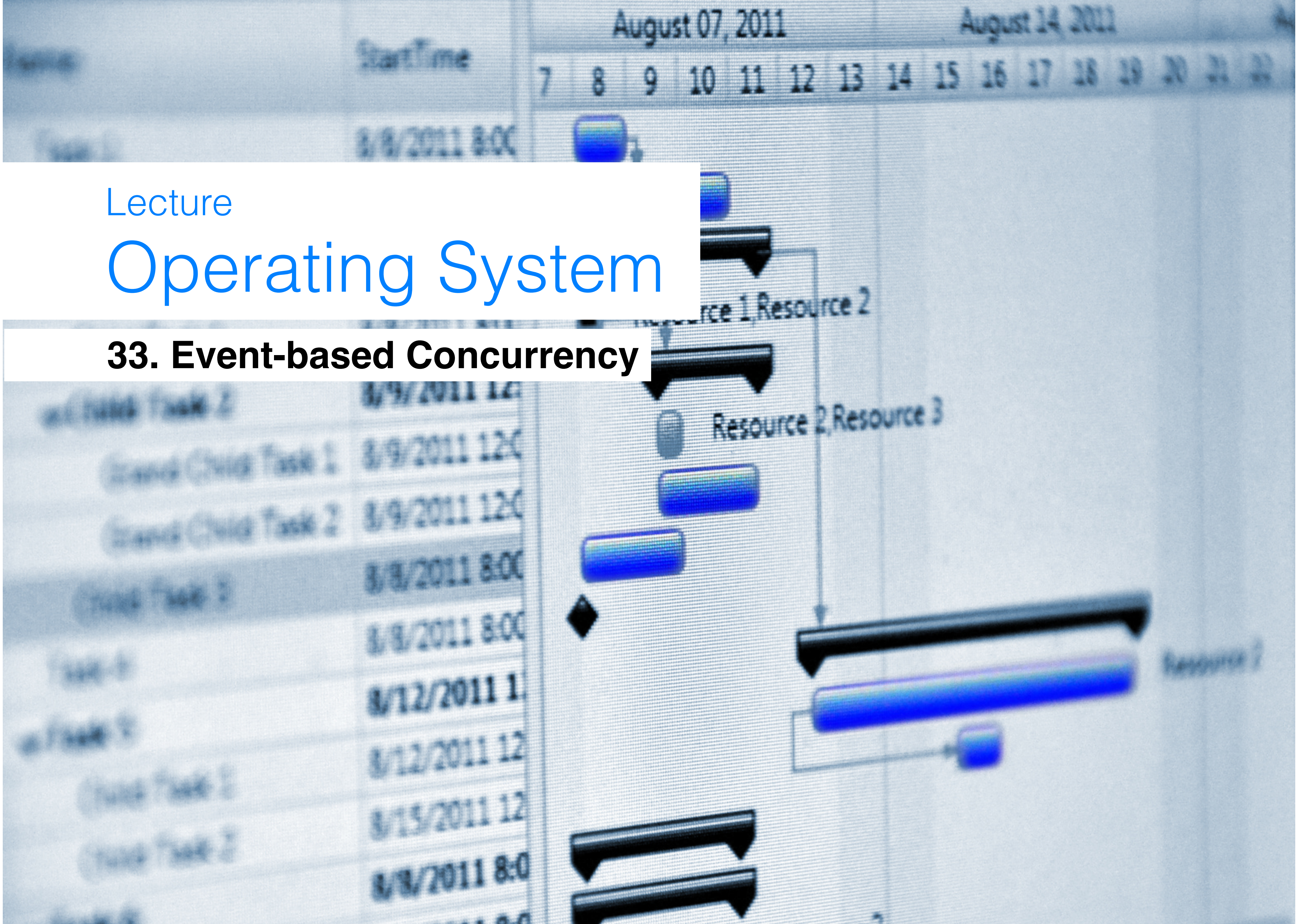

Lecture

Operating System

33. Event-based Concurrency

Professor Dr. Michael Mächtel

33. Event-based Concurrency

1. **Basic Idea**
2. **Problems**
3. **Signals**

33. Event-based Concurrency

1. **Basic Idea**
2. Problems
3. Signals

Event-based Concurrency

- A different style of **concurrent programming**
  - Used in GUI-based applications, some types of internet servers.
- **The problem** that event-based concurrency addresses is two-fold.
  - **Managing concurrency correctly** in multi-threaded applications.
    - Missing locks, deadlock, and other nasty problems can arise.
  - The developer has little or no control over **what is scheduled** at a given moment in time.

The Crux

- How can we build a **concurrent** server **without** using **threads**
  - but **retain control** over concurrency
  - and **avoid** some of the **problems** that seem to plague multi-threaded applications?

The Basic Idea: An Event Loop

- The approach:
  - **Wait** for something (i.e., an "*event*") to occur.
  - When it does, **check** what type of event it is.
  - **Do** the small amount of work it requires.
- Example:

```
while(1){
    events = getEvents();
    for( e in events )
        processEvent(e); // event handler
}
```

A canonical event-based server (Pseudo code)

How exactly does an event-based server determine which events are **taking place**.

An Important API: select()

(or poll())

- Check whether there is any **incoming I/O** that should be attended to.
  - `select()`

```
int select(int nfds,
    fd_set * restrict readfds,
    fd_set * restrict writefds,
    fd_set * restrict errorfds,
    struct timeval * restrict timeout);
```

  - Lets a server determine that a **new packet has arrived** and is in need of processing.
  - Let the service know when **it is OK to reply**.
  - `timeout`
    - `NULL`: Cause `select()` to block indefinitely until some descriptor is ready.
    - `0`: Use the call to `select()` to return immediately.

Using `select()`

```
while (1) {
        // initialize the fd_set to all zero
        fd_set readFDs;
        FD_ZERO(&readFDs);

        // now set the bits for the descriptors
        // this server is interested in
        // (for simplicity, all of them from min to max)
        int fd;
        for (fd = minFD; fd < maxFD; fd++)
            FD_SET(fd, &readFDs);

        // do the select
        int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);

        // check which actually have data using FD_ISSET()
        for (fd = minFD; fd < maxFD; fd++)
            if (FD_ISSET(fd, &readFDs))
                processFD(fd);
    }
```

Why Simpler? No Locks Needed

- The event-based server **cannot be interrupted** by another thread:
  - With a **single CPU** and an **event-based application**.
  - It is decidedly **single threaded**.
  - Thus, **concurrency bugs** common in threaded programs **do not manifest** in the basic event-based approach.

33. Event-based Concurrency

1. Basic Idea
2. **Problems**
3. Signals

A Problem: Blocking System Calls

- What if an event requires that you issue **a system call** that might block?
  - There are no other threads to run: *just the main event loop*
  - The entire server will do just that: **block until the call completes**.
  - **Huge potential waste of resources**

In event-based systems: **no blocking calls** are allowed.

A Solution: Asynchronous I/O

- Enable an application to issue an I/O request and **return control immediately** to the caller, before the I/O has completed.
  - Example:

```
struct aiocb {
    int aio_fildes;             /* File descriptor */
    off_t aio_offset;           /* File offset */
    volatile void *aio_buf;     /* Location of buffer */
    size_t aio_nbytes;          /* Length of transfer */
```

    - An Interface provided on *Mac OS X*
    - The APIs revolve around a basic structure, the `struct aiocb` or **AIO control block** in common terminology.

Example: Async I/O

- To issue an asynchronous read to a file

```
int aio_read(struct aiocb *aiocbp);
```

  - If successful, it returns right away and the application can continue with its work.

- Checks whether the request referred to by `aiocbp` has completed.

```
int aio_error(const struct aiocb *aiocbp);
```

  - An application can **periodically pool** the system via `aio_error()`.
    - If it has completed, returns success.
    - If not, EINPROGRESS is returned.

Example: Async I/O (Signaling)

- Interrupt
  - Remedy **the overhead to check** whether an I/O has completed
  - Using **UNIX signals** to inform applications when an asynchronous I/O completes.
  - Removing the need to *repeatedly ask the system*.

Another Problem: State Management

- The code of event-based approach is generally **more complicated** to write than *traditional thread-based* code.
  - It must package up some program state for the next event handler to use when the I/O completes.
  - The state the program needs is on the stack of the thread.
    - **manual stack management** !

Example: Problem with 'state'

- Example (an thread-based system):

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

- How to do in asynchronous systems?
  - First **issue** the read asynchronously.
  - Then, **periodically check** for completion of the read.
  - That call informs us that the **read is complete**.
  - How does the event-based server know **what to do**?

Solution to 'state' Problem

- Solution: **continuation**
  - **Record** the needed information to finish processing this event in *some data structure*.
  - When the event happens (i.e., when the disk I/O completes), **look up** the needed information and process the event.

What is still difficult with Events.

- Systems moved from a single CPU to **multiple CPUs**.
  - Some of the simplicity of the event-based approach disappeared.
- It **does not integrate well** with certain kinds of systems activity.
  - **Ex. Paging**: A server will not make progress until page fault completes (implicit blocking).
- Hard to manage overtime: The exact semantics of various routines changes.
- Asynchronous disk I/O **never quite integrates with asynchronous network I/O** in as simple and uniform a manner as you might think.

33. Event-based Concurrency

1. Basic Idea
2. Problems
3. **Signals**

UNIX signals

- Provide a way to communicate with a process.
  - HUP (hang up), INT(interrupt), SEGV(segmentation violation), and etc.
  - Example: When your program encounters a segmentation violation, the OS sends it a SIGSEGV.

Example: Standard Signals

```
static void SigHandler( int SignalNumber )
{
    printf("got signal %d\n", SignalNumber );
}
```

```
int main( int argc, char **argv )
{
    int i;

    signal( SIGINT, SigHandler );
    printf("my PID is %d\n", getpid() );
    for( i=1; i<100; i++ ) {
        printf("%d seconds\n", i);
        sleep( 1 );
    }
    return( 0 );
}
```

Unreliable Signals

- Standard Signals
  - poor performance
  - not reliable
  - possible race conditions
  - not all system calls are 'signal' safe

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPT	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALARM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCLD	Death of a child
19	SIGPWR	Power failure

Reliable POSIX Signals

- `sigaction()`: Install a signal handler
- `sigprocmask()`: Mask signals
- `sigsendset()`: Send signal
- `sigsuspend()`: Block until signal received

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

#define message "SIGINT caught\n"

void signal_handler(int value)
{
// printf is not signalsafe
    write( 1, message, strlen(message) );
}
```

Reliable POSIX Signals (Cont.)

```
int main(int argc, char **argv, char **envp )
{
    struct sigaction new_action;
    int time_to_sleep;

    new_action.sa_handler = signal_handler;
    sigemptyset( &new_action.sa_mask );
    new_action.sa_flags = 0;

    sigaction( SIGINT, &new_action, NULL );

    printf("pid: %d\n", getpid() );
    time_to_sleep = 10;
    while( time_to_sleep )
        time_to_sleep = sleep( time_to_sleep );

    return 0;
}
```

Signale Multithreading

- Which thread of a multithreaded app receives the signal?
  - Synchronous Signal (e.g. `SIGFPE`) ?
  - from `kill()` ?
- Another example:
  - `mutex1` protects critical section, also used in signal handler

```
void signal_handler( int sig )
{
        ...
        pthread_mutex_lock( &mutex1 );
        ...
        pthread_mutex_unlock( &mutex1 );
        ...
}
```

Thanks

Questions?