Synchronization

Alexandru Andrei
alean@ida.liu.se
phone: 013-322828, room: B 3D:439

Lecture Plan

1. What is an operating system? What are its functions?
   Basics of computer architectures. (Part I of the textbook)
2. Processes, threads, schedulers (Part II, chap. III-V)
3. Synchronization & Deadlock (Part II, chap. VI, VII)
4. Primary memory management. (Part III, chap. VIII, IX)
5. File systems and secondary memory management (Part IV, chap. X, XI, XII)

Outline

- Need for synchronization
- Synchronization mechanisms (1)
  - System calls: pause, select, waitpid
  - Message passing: blocking read/writes
- Race conditions, critical regions, atomic operations, busy-waiting, hardware support for their implementation
- Synchronization mechanisms (2)
  - Semaphores
  - Monitors
  - Re-entrant code
  - Deadlock
- Classical problems of synchronization
  - Readers/writers, bounded buffer, dining philosophers

Need for Synchronization

- Processes that do not interact with other processes (from their functionality point of view) are independent
- Processes that do interact are co-operating

E.g.:
- Web server and browser (client) (and all applications in the client-server paradigm)
- Mail composer and spell checker
- sort --nr +1 -2 < grades.txt | cut --f2 | uniq | head -10
does a files where the grades are on the second column, sorts it in descending order of the grades, takes the second column, eliminates duplicates, and prints the highest ten grades

Last on TTIT61

- Processes are executing programs
- Kernel manages them, their state, associated data (open files, address translation tables, register values, etc.)
- Threads are lightweight processes, i.e. they share data segments, are cheaper to spawn and terminate
- Scheduler selects next process to run among the ready ones
- Various scheduling algorithms and criteria to evaluate them

E.g.:
- Web server pumps bytes down the connection after getting a request. Browser reads served document after the client starts sending it.
- The spell checking is triggered by the appearance of new text in the composer.
One Producer/One Consumer

- Producer inserts data in the buffer
- Consumer removes data from the buffer

Simple Solutions: Pause

```c
#include <signal.h>

int main() {
    signal(SIGINT, sigproc);
    signal(SIGUSR1, quitproc);
    printf("ctrl-c disabled use ctrl-\ to quit");
    for(;;); /* infinite loop */
}
```

```c
void sigproc() {
    signal(SIGINT, sigproc);
    /* Some versions of UNIX will reset signal to default after each call. So for portability reset signal each time */
    printf("you have pressed ctrl+c but i wont die");
}
```

```c
void quitproc() {
    printf("now i will");
    exit(0); /* normal exit status */
}
```

More on Signals

```c
[sasa@lap-154 ~]$ ./a.out
you have pressed ctrl-c but i'm not dying
you have pressed ctrl-c but i'm not dying
you have pressed ctrl-c but i'm not dying
now i'm dying
```

More on Signals (2)

```c
[sasa@lap-154 ~]$ ps aux | grep a.out
sasa 12237 95.3 0.0 1500 296 pts/0 S+ 20:37 0:24 ./a.out
sasa 12247 0.0 0.1 3900 724 pts/1 S+ 20:38 0:00 grep a.out
[sasa@lap-154 ~]$ kill -SIGINT 12237     #it wont die
[sasa@lap-154 ~]$ kill -SIGUSR1 12237 #it will die
```

Blocking Reads/Writes

- The previous problems could also be solved with the synchronization mechanism of blocking reads or writes
- Blocking read:
  - The invoking process attempts to read something from a descriptor (socket, pipe, file, message queue). If no data is available for reading, the invoking process blocks until data is available.
- Blocking write:
  - The invoking process attempts to write a data item to a descriptor. It blocks until a process consumes the data item.

Producer/Consumer

- Producers insert data in a file
- Consumer reads data from the files
Simple Solutions: Select

- The select system call
  - The caller of select(fd, ...) blocks until a specified operation (read or write) is possible on the file descriptor fd without blocking.
  - The mail composer could write everything the user types to a pipe descriptor and the spell checker could call select on the pipe descriptor.
  - Disadvantage: There must be a pipe, or socket, or file shared between the caller of select and the processes that makes the unblocking read or write possible.

man select in Solaris/Linux

Non-blocking read (Multiplexing) Using select

```c
#include <unistd.h>

int fd1, fd2, max_fd;
fd_set readset, writeset;

fd1=open("file1", O_RDONLY); fd2=open("file2", O_WRONLY);
max_fd=max(fd1, fd2)+1;

int result;
while (1) {
    FD_ZERO(&readset);
    FD_SET(fd1, &readset);
    result = select(max_fd, &readset, &writeset, NULL, NULL);
    if (FD_ISSET(fd1, &readset)) {
        /* fd1 has data available to be read */
        result = read(fd1, some_buffer, 100);
    }
    if (FD_ISSET(fd2, &writeset)) {
        /* fd2 is available for writing */
        result = write(fd2, "blabla", 6);
    }
}
```

Unbounded Buffer

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Not blocking</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

How many elements would the buffer contain if the writing was blocking?

Race Condition

- We say that we have a race condition when, given that several processes access the same data, the execution outcome depends on the order in which the concurrent processes accessed the data.

- E.g.: `buffer[index] = processID; ++index;`

Outline

- Need for synchronization
- Synchronization mechanisms
  - System calls: pause, select, waitpid
  - Message passing: blocking read/write
- Race conditions, critical regions, atomic operations, busy-waiting, hardware support for their implementation
- Synchronization mechanisms
  - Semaphores
  - Monitors
  - Re-entrant code
  - Deadlock
- Classical problems of synchronization
  - Readers/writers, bounded buffer, dining philosophers

Producer/Consumer

- Producers insert data in the buffer
- Consumers remove data from the buffer
**Producer/Consumer**

Producer

/* produce an item and put in nextProduced */
while (count == BUFFER_SIZE)
    ; // do nothing
buffer [in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
count++; //consume the item in nextConsumed

Consumer

while (count == 0)
    ; // do nothing
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
count--; //consume the item in nextConsumed

---

**Critical Sections**

A critical section is a section of code that no process may execute while another process executes it.

The execution of critical sections by processes is mutually exclusive in time.

The critical section problem is to design a protocol to ensure the mutual exclusive execution.

---

**Critical-Section Requirements**

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the $N$ processes

---

**Two-Process Solution (1)**

$P_i$
do {
    flag[i] = true;
    while (flag[j]) no-op;
    critical section
    flag[i] = false; remainder
} while (true);

$P_j$
do {
    flag[j] = true;
    while (flag[i]) no-op;
    critical section
    flag[j] = false; remainder
} while (true);

---

**Two-Process Solution (2)**

$P_i$
do {
    flag[i] = true;
    turn = j;
    while (flag[j] & turn == j) no-op;
    critical section
    flag[i] = false; remainder
} while (true);

$P_j$
do {
    flag[j] = true;
    turn = i;
    while (flag[i] & turn == i) no-op;
    critical section
    flag[j] = false; remainder
} while (true);

---

**Two-Process Solution (3)**

$P_i$
do {
    flag[i] = true;
    turn = j;
    while (flag[j] & turn == j) no-op;
    critical section
    flag[i] = false; remainder
} while (true);

$P_j$
do {
    flag[j] = true;
    turn = i;
    while (flag[i] & turn == i) no-op;
    critical section
    flag[j] = false; remainder
} while (true);

---

**Deadlock!**

Works? YES!
Multiple-Process Solution

```c
do {
    choosing[i] = true;
    number[i] = max(number[0], …, number[n – 1]) + 1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) no-op;
        while (number[j] != 0 && (number[j], j) < (number[i], i))
            no-op;
    }
    critical section
    number[i] = 0;
    remainder
} while (true);
```

Busy Waiting

- All algorithms on previous slides contained while (condition) do nothing:
  - The process is marked as ready, and whenever running, it just wastes CPU time (and money, where one buys CPU time)
  - The process is said to be busy waiting

Hardware Support: Interrupts

- Disabling interrupts when entering the critical section.
  - Not efficient for multi-processor systems
  - If the critical section is long, the system is not responsive and/or it may lose events

Hardware Support: Test and Set

```c
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

- It runs atomically, i.e. as an uninterruptible unit.
- lock set to false means it is free

```c
do {
    while (TestAndSet(lock)) no-op;
    critical section
    lock = false;
    remainder
} while (true);
```

Hardware Support: Swap

- key[i] is local to process i; lock is global

```c
do {
    key[i] = true;
    while (key[i] == true)
        swap(lock, key[i]);
    critical section
    lock = false;
    remainder
} while (true);
```

- Are these two solutions, TestAndSet-based and Swap-based, two-process solutions or multiple-process solutions?
Are these two solutions, TestAndSet-based and Swap-based, two-process solutions or multiple-process solutions?

Single processor or multiprocessor?

Semaphores

Semaphores:
- Synchronization tool introduced by Dijkstra with the following two operations:
  - \( P() \) (Dutch: probeeren = to test)
    - while (\( S \leq 0 \)) no-op; \(-S\);
  - \( V() \) (Dutch: verhogen = to increment)
    - \(+S\);
- No modification of \( S \) is allowed to be performed by a process while another process operates on \( S \).
- Testing and decrementing of \( S \) in \( P \) has to be done without interruption by another process.

Non-Blocking Write, Blocking Read with Sem.

Writer:
write(fd, data item);
sem.V();

Reader:
sem.P();
data item = read(fd);

We assume an unbounded buffer associated to the descriptor fd.

Critical Sections with Semaphores

do {
  sem.P();
critical section
  sem.V();
  remainder
} while (true)

Multi-process or two-process solution?
The semaphore can be binary or counting
To do busy waiting or not to do busy waiting

P()
```c
void Semaphore::V() {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    if (thread != NULL) // make thread ready, consuming the V immediately
        scheduler->ReadyToRun(waiting_thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

**Synchronization**

- The **synchronized** keyword used for methods inside a class

```java
class A {
    void synchronized X();
    void synchronized Y();
    ...
}
```

**Java Monitors**

- The `synchronized` keyword used for methods inside a class

```java
Thread 1: A p; Thread 2: A q;
p.X(); q.Y()
```

**Deadlock and Starvation**

- Let S and Q be two semaphores initialized to 1
  
  ```
  P0
  S.P();
  Q.P();
  Q.P();
  S.P();
  ... 
  Q.V();
  S.V();
  S.V();
  Q.V();
  ```

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

**Condition Variables**

- Acquire a semaphore only when you are sure it can be used
- Condition variables:
  - Semaphore.P() -> Condition.Wait(semaphore)
  - Semaphore.V() -> Condition.Signal()
Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Re-Entrant Code

- A section of code that may be executed by a process before another process leaves it

  E.g.:
  ```c
  char kernel_buffer[BUF_MAX];
  void user_to_kernel(int virtualAddr, char kernel_buf[]) {
    int physicalAddr, i = 0;
    do {
      physicalAddr = translate(virtualAddr);
      kernel_buf[i++] = c = memory[physicalAddr];
    } while (c != ‘\0’);
  } user_to_kernel(1000, kernel_buffer) || user_to_kernel(2000, kernel_buffer)
  ```

Re-Entrant Code

- The problem with the previous code is that both processes share the `kernel_buffer` array.

  - Do not use global variables when writing re-entrant code
  - If you have to use them, then protect them with locks. The protected accesses will not be re-entrant any more.

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Outline

- Need for synchronization
- synchronization mechanisms
  - System calls: pause, select, waitpid
  - Message passing: blocking read/writes
- Race conditions, critical regions, atomic operations, busy-waiting, hardware support for their implementation
- synchronization mechanisms
  - Semaphores
  - Monitors
  - Re-entrant code
  - Deadlock
  - Classical problems of synchronization
  - Readers/writers, bounded buffer, dining philosophers

Re-Entrant Code

- The problem with the previous code is that both processes share the `kernel_buffer` array.

  - Do not use global variables when writing re-entrant code
  - If you have to use them, then protect them with locks. The protected accesses will not be re-entrant any more.
Outline

- Need for synchronization
- Synchronization mechanisms
  - System calls: pause, select, waitpid
  - Message passing: blocking read/writes
- Race conditions, critical regions, atomic operations, busy-waiting, hardware support for their implementation
- Synchronization mechanisms
  - Semaphores
  - Monitors
  - Re-entrant code
  - Deadlock
  - Classical problems of synchronization
  - Readers/writers, bounded buffer, dining philosophers

Bridge Crossing Example

Kansas law: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other one has gone”

Deadlock

Process \( P_0 \) blocks if resource \( B \) is unavailable
Resource \( B \) may never become available because process \( P_1 \) holds it and never releases it
Process \( P_1 \) may never release it because in its turn, it waits for a different resource, \( A \), that could be held by process \( P_0 \)
Processes \( P_0 \) and \( P_1 \) deadlocked

Conditions for Deadlock

- Mutual exclusion: At least one resource cannot be shared
- Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources, held by other processes
- No preemption: Resources are released only voluntarily
- Circular wait: \( P_0 \) must wait on a resource held by \( P_1 \), which in turn waits on a resource held by \( P_2 \), … held by \( P_{n-1} \), which in turn waits on a resource held by \( P_0 \)

System Model

- Resource types \( R_1, R_2, \ldots, R_m \)
  - CPU cycles, memory space, I/O devices
- Each resource type \( R_i \) has \( W_i \) instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Resource-Allocation Graph

- Process
- Resource Type with 4 instances
  - \( P_i \) requests instance of \( R_j \)
  - \( P_j \) is holding an instance of \( R_i \)
A. Andrei, Process programming and operating systems, Synchronization

Resource Allocation Graph

- Process → Resource edge indicates resource request
- Resource → Process edge indicates that the resource is held by the process

Existence of cycles is a necessary condition for the existence of deadlocks
Also sufficient if there exists only one instance of each resource

We have a cycle, but no deadlock
When P₄ releases an instance of R₂, P₃ may take it

A. Andrei, Process programming and operating systems, Synchronization

Handling Deadlocks

- Deadlock avoidance
- Deadlock detection and recovery
- Ignore the problem

Outline

- Need for synchronization
- synchronization mechanisms
  - System calls: pause, select, waitpid
  - Message passing: blocking read/writes
- Race conditions, critical regions, atomic operations, busy-waiting, hardware support for their implementation
- synchronization mechanisms
  - Semaphores
  - Monitors
  - Re-entrant code
  - Deadlock
- Classical problems of synchronization
  - Readers/writers, bounded buffer, dining philosophers

A. Andrei, Process programming and operating systems, Synchronization

Classical Problems: Dining Philosophers

- Dining philosophers:
  - They may eat only when they have both forks
  - They cannot get both forks in one move
  - Implement a resource (fork) access protocol for philosophers such that they do not starve

Readers/Writers Problem

- There exists a shared resource
- Processes that do not change it upon an access are readers
  All the other are writers

First readers/writers problem:
  - No reader will be kept waiting unless a writer has already obtained writing permission (Read accesses are not blocked if a writer manifested interest in writing)

Second readers/writers problem:
  - Once a writer is ready, that writer performs its write as soon as possible (Read accesses are blocked if a writer manifested interest in writing)
Readers/Writers Problem

```c
int counter_readers;
Semaphore sem_readers;
Semaphore sem_writers;

Reader:  Writer:
  .     .
  .     .
  .     .
```

```
int counter_readers;
Semaphore sem_readers;
Semaphore sem_writers;

Reader:
sem_readers.P();
if (counter_readers==0)
  sem_writers.P();
counter_readers++;
sem_readers.V();
//READ
sem_readers.P();
counter_readers--;
if (counter_readers==0)
  sem_writers.V();
sem_readers.V();

Writer:
sem_writers.P();
//WRITE
sem_writers.V();
```

Summary

- Need for processes synchronization
- Race condition, critical section, atomic operations, mutual exclusion
- Critical section access protocols
- Busy waiting
- Hardware support
- Semaphores, monitors, conditions
- Deadlocks

Reading Material

- Silberschatz, 7th edition
  - Chapter 6: 6.1 – 6.8
  - Chapter 7: 7.1 – 7.4