List of Slides

1. Intro to Concurrency and Synchronization
2. Forms of Concurrency
3. parbegin/parend notation
4. Unix Fork/Wait Usage
5. Why is Concurrency Hard?
6. Mutual Non-Interference
7. Preventing Interference
8. Resource Types
9. A Producer/Consumer System
10. The Producer/Consumer Problem
11. The Critical Section Problem
12. Dekker’s Algorithm
14 Software Mutual Exclusion Algorithm - 1st Try
15 Software Mutual Exclusion - 2nd Try
16 Software Mutual Exclusion Algorithm - 2nd Try Continued
17 Software Mutual Exclusion Algorithm - 2nd Try Fixed
18 Software Mutual Exclusion Algorithm - 3nd Try
19 Intro to Livelock
20 Characteristics of a good solution
21 Dekker’s Algorithm Solution Method
22 Dekker’s Algorithm A Working Version
23 Peterson’s Algorithm A Simpler Working Version
24 Hardware — Disabling Interrupts
25 Hardware — Test and Set Instructions
26 Hardware — Exchange Instructions
Concurrency takes two major forms:

1. Interleaving
2. Overlapping
Many notations exist for expressing concurrency. Define each process as a statement (typically a subroutine) and have concurrent statements of the form:

```
parbegin
  s1;
  s2;
  ...
  sn;
parend
```

where:

1. Any programming statement may have `parbegin/parend` pairs.

2. The `parend` waits for the last statement in `s1`, `s2`, `s3` to complete.

How can we express this using Unix?
Recall that in Unix:

1. `fork()` creates a new child process, returning 0 to child.

2. `wait()` waits for child process termination.

so one solution is:

```c
if (fork() == 0){
    s1; // Child does s1
} else if (fork() == 0){
    s2; // Child does s2
} .... {
} else { wait(); wait(); ... wait(); }
```
Concurrency is hard because sometimes order of evaluation matters.

*Interference* occurs when two processes could manipulate a shared resource in a conflicting fashion.

\[
\begin{array}{|c|c|}
\hline
\text{Process 1} & \text{Process 2} \\
\hline
x := x + 1; & x := x + 2; \\
\hline
\end{array}
\]

Table 1: Two Concurrent processes

Consider the example in Table 1 with \( x = 0 \) initially. Then \( x \) can become either 1, 2 or 3 depending on order of evaluation (and code generation).

This is called an *overwrite condition* or a *race condition*. 
6 – Mutual Non-Interference

A more formal model of interference can be formulated using the following notation:

- $P_i$ denotes the $i$th process.
- $W(P_i)$ be the addresses written to by $P_i$.
- $R(P_i)$ be the addresses read by $P_i$.

Two processes $P_i, P_j, i \neq j$ are **mutually noninterfering** if:

$$\left( W(P_i) \cap W(P_j) \right) \cup \left( R(P_i) \cap W(P_j) \right) \cup \left( R(P_j) \cap W(P_i) \right) = \emptyset$$

This is sometimes called Bernstein’s condition.
To prevent interference processes limit concurrency using:

1. *Synchronization* — forces a process to *wait* for another process.

2. *Communication* — transfers information between processes.
   - (a) *Shared Memory*
   - (b) *Message Passing*
8 – Resource Types

Systems resources can be categorized as:

1. **Renewable** — The resource has a fixed total allocation, and is neither created nor destroyed. Examples include:
   (a) Memory (Main and Secondary storage)
   (b) Processors
   (c) Busses

2. **Consumable** — The resource has a variable allocation with new units being created (produced) and destroyed (consumed). Examples include:
   (a) Buffers
   (b) Messages
   (c) Signals
9 – A Producer/Consumer System

Consider the following system:

1. A process reads input from a buffer

2. A process reads input from a source buffer and writes output to a destination buffer.

3. A process reads from the destination buffer and processes the data (e.g. displays it).

Typically processes that write without reading an input buffer are not modeled.
The producer/consumer problem is defined as having the producer/consumer system shown before.

Some challenges include:

1. Ensuring that data is available for a process reading a buffer.

2. Preventing interference during buffer access. If data is overwritten before consumer can access it, it is corrupted.
11 – The Critical Section Problem

The producer consumer problem is part of a larger class of problems. Some important issues include:

- A *critical section* is a program segment which is susceptible to interference. (e.g. In our example, the buffer access was the critical section.)

- To avoid interference, only one process is allowed in its critical section at a time.

- *Mutual exclusion* ensures that only one process of a mutually interfering pair of processes is in its critical section at a time.

- A (possibly compound) statement is considered *atomic* if it is executed as if it were indivisible.
12 – Dekker’s Algorithm

Dekker made the first software mutual exclusion algorithm which assumes that concurrent reads and writes are serialized in an unspecified order.

The algorithm has two processes, which access a critical section inside a loop.

Consider a solution using a shared scalar variable, turn indicating which process can enter its critical section. Two processes $P_0$ and $P_1$ can be thought of as competing for the turn variable.
Suppose we use a solution of the form:

process 0          process 1
while $turn \neq 0$ do while $turn \neq 1$ do
    nothing           nothing
 $<$Critical Section$>$   $<$Critical Section$>$
 $turn := 1$           $turn := 0$

This solution has drawbacks:

1. Busy waiting
2. Strictly alternating critical section access
14 – Software Mutual Exclusion - 2nd Try

Rather than having a single value, let’s keep an array of shared flags for each process, where $flag[i]$ is the flag for the $i$th process.

Both processes set their flags to true when the process is in its critical section and reset it to false when leaving the critical section. Both processes have their flags initialized to false.

To enter a critical section a process checks the other processes flag. If the other processes flag is false, then it writes true on its own flag and enters the critical section.

When leaving the critical section it writes false on its own flag.
Both processes execute:

```plaintext
while (flag[1 - i]) do nothing
flag[i] := true
<critical section>
flag[i] := false
```

Problem: This algorithm is susceptible to a race condition, where both processes examine the others’ flag and both flags are false. This allows both into the critical section - bad!
To fix this each process must set its flag before checking the other’s flag:

flag[i] := true
while (flag[1 - i]) do nothing
<critical section>
flag[i] := false

Problem: Both processes may wait forever (deadlock) if both flags are set to true simultaneously.
Rather than assert its rights to go into its critical section, a process could signal intent, and back off if it is blocked:

flag[i] := false;
while flag[1 - i] do
  begin
    flag[i] := false;
    <delay for a short time>
    flag[i] := true;
  end while;
<Critical Section>
flag[i] := false;

Figure 1: A 3rd attempt at Software Mutual Exclusion

Does this solution have any problems?
18 – Intro to Livelock

The previous algorithm, (see Figure 1) has the following flaw:

\[
\begin{align*}
\text{P}[0]: & \quad \text{flag}[0] := \text{true}; \\
\text{P}[1]: & \quad \text{flag}[1] := \text{true}; \\
\text{P}[0]: & \quad \text{Test flag}[1] \text{ in while loop} \\
\text{P}[1]: & \quad \text{Test flag}[0] \text{ in while loop} \\
\text{P}[0]: & \quad \text{flag}[0] := \text{false}; \\
\text{P}[1]: & \quad \text{flag}[1] := \text{false}; \\
\text{P}[0]: & \quad \text{flag}[0] := \text{true}; \\
\text{P}[1]: & \quad \text{flag}[1] := \text{true};
\end{align*}
\]

Livelock is refers to a synchronization error when two (or more) processes cycle through a set of states in a failed attempt at synchronization.
19 – Characteristics of a good solution

A good Mutual Exclusion Algorithm:

1. Actually enforces the mutual exclusion.
2. Prevents starvation (i.e. has bounded waiting).
3. Allows a process, say process $i$, to go through its critical section many times if the other process, process $1 - i$, is not attempting to go through its critical section. (ensures progress).

The solution can assume that a process remains in its critical section for a finite period of time, and should NEVER make assumptions about processor speeds.
To do this:

- The turn variable is reintroduced.
- When process $i$ attempts to the critical section, it sets $\text{flag}[i] := \text{true}$ and checks $\text{flag}[1-i]$:
  - if $\text{flag}[1-i] = \text{false}$, then process $i$ enters its critical section.
  - otherwise $\text{flag}[1-i] = \text{true}$, then periodically check $\text{turn}$ until $\text{turn} = i$.
- When process $i$ leaves the critical section it it sets $\text{flag}[i]=\text{false}$ and gives the other process priority by setting $\text{turn}=1-i$. 
other := 1 - i;
flag[i] := true;
while flag[other] do
  if turn = other then
    begin
      flag[i] := false;
      while turn = other do nothing
      flag[i] := true;
    end
  <Critical Section>
  turn := other;
  flag[0] := false;
Peterson’s algorithm “politely” gives the other process a turn.

other := 1 - i;

loop

flag[i] := true;
turn := other;
while(flag[other] and (turn == other)){
  do nothing
}

Critical section

flag[i] = false;

Non Critical Section

end loop
23 – Hardware — Disabling Interrupts

Typically the solution is not solely implemented in software for efficiency reasons.

For uniprocessor architectures disabling interrupts prevents preemption. The program structure is:

disable interrupts
<Critical Section>
enable interrupts
24 – Hardware — Test and Set Instructions

The test and set operation performs an atomic read and update of a memory location as follows:

```c
atomic bit testandset(bit &i) {
    if (i = 0) {
        i := 1;
        return 1;
    } else {
        return 0;
    }
}
```

The program structure is:

```c
shared integer bolt = 0; // common to all processes
// local to each process
while(!testset(bolt));
<Critical Section>
bolt := 0;
```
25 – Hardware — Exchange Instructions

Exchange operation atomically swaps two values, as follows:

```c
atomic void exchange( register &r, &m){
    temp := m; // r, m and t are same type
    m := r;
    r := temp;
}
```

The program structure is:

```
// Common to all processes
const int N= ... // number of processes
shared int bolt = 0; // initialization
// For process i
int keyi = 1; // NOT a shared variable
repeat
    exchange(bolt, keyi); // lock the bolt
until keyi = 0;
<Critical Section>
exchange(bolt, keyi); // unlock the bolt
```
References