Concept of a process

• In the context of this course a process is a program whose execution is in progress.
• States of a process: running, ready, blocked
Concurrent processes

• In a multiprocessor system two or more processes can be executing at the same time
  – physical concurrency - as opposed to logical concurrency achieved by interleaving process execution

• Concurrent processor interaction:
  – shared variables
  – message passing

• If they don’t interact their execution is functionally the same as their serial execution
The critical section problem

- A critical section is a code segment of a concurrent process in which a shared resource is accessed.
- Concurrent access to a shared variable is potentially dangerous.
  - Example: if \( a = 0 \), what is the result of the command \( a = a + 1 \) executed simultaneously by processes A and B?
  - A common solution is the mutual exclusion i.e. serialization of accesses.
Early Solutions

• Busy Waiting
  – Wastes cycles

• Disabling Interrupts
  – Only applicable to uniprocessors

• A special test-and-set instruction
Example of busy waiting on a lock (1/2)

- One could think of using a variable as a flag to be checked upon entering a critical section ...
- … but the lock itself is a critical section!

```c
Shared integer lock = 0;
Process i
  .
  while lock == 1;
  lock = 1;
  execute CS;
  lock = 0;
  .
Process A
  .
  while lock == 1;
  lock = 1;
  .
  .
Process B
  .
  while lock == 1;
  lock = 1;
  .
  .
```

Possible race condition
Example of busy waiting on a lock (2/2)

• The correct implementation uses a test-and-set instruction to avoid race conditions

Semantic of test-and-set instruction

```c
int test-and-set (int a) {
    int rv = a;
    a = 1;
    return rv;
}
```

Correct lock implementation

```c
Process A

Shared integer lock = 0;
.
.
While( test-and-set(lock) ==1) ;
.
.
```
Locks: pros and cons

• Pros:
  – simple and fast
  – ubiquitous: every processor has a test-and-set or equivalent operation

• Cons:
  – busy waiting is wasteful of resources (CPU cycles, memory bandwidth)
Semaphores - definition

- Proposed by Dijkstra, it was the first high level constructs used to synchronize concurrent processes.
- A semaphore $S$ is an integer variable on which two atomic operations are defined, $P(S)$ and $V(S)$, and has an associated queue.
- $P$ and $V$ semantics:

  $P(S)$: if $S \geq 1$ then $S := S - 1$
  else <block and enqueue the process>;

  $V(S)$: if <some process is blocked on the queue> then
  <unblock a process>
  else $S := S + 1;$
Semaphores - properties

- The P operation may block a process, but V does not
- Two type of semaphores
  - binary: initial value is 1
  - resource counting: any initial value
- P and V are atomic operations

\[
P(S) : \begin{align*}
\text{if } S & \geq 1 \text{ then } S := S - 1 \\
\text{else } & \text{<block and enqueue the process>};
\end{align*}
\]

\[
V(S) : \begin{align*}
\text{if } & \text{<some process is blocked on the queue>} \text{ then} \\
& \text{<unblock a process>} \\
\text{else } & S := S + 1;
\end{align*}
\]
Example of use

Shared var mutex: semaphore = 1;

Process $i$

begin
  .
  .
  .
  $P$(mutex);
  \emph{execute CS};
  $V$(mutex);
  .
  .
  .
  \textbf{End};
Other synchronization problems

• Semaphore can be used in other synchronization problems besides Mutual Exclusion

• The Producer-Consumer problem
  – a finite buffer pool is used to exchange messages between producer and consumer processes

• The Readers-Writers Problem
  – reader and writer processes accessing the same file

• The Dining Philosophers Problem
  – five philosophers competing for a pair of forks
Reader-Writers problem

- The shared resource is a file accessed by both reader and writer processes
- The synchronization constraints are:
  - readers should be able to concurrently access the file
  - only one writer at a time can access the file
  - readers and writers exclude each other
- Variants:
  - reader’s priority: arriving readers have priority over waiting writers
  - writer’s priority: writers have priority over waiting readers
Simple Readers-Writers solution

- The following scheme is very simple but does not allow concurrent reader access

**Procedure reader**

- P(mutex)
- `<read file>`
- V(mutex)

**Procedure writer**

- P(mutex)
- `<write file>`
- V(mutex)
Readers-Writers solution with concurrent reader access

**Procedure** reader

\[
\begin{align*}
P(\text{reader_mutex}) \\
\text{if readers} &= 0 \text{ then} \\
\quad \text{readers} &= \text{readers} + 1 \\
\quad P(\text{writer_mutex}) \\
\text{else} \\
\quad \text{readers} &= \text{readers} + 1 \\
V(\text{reader_mutex}) \\
\langle \text{read file} \rangle \\
\end{align*}
\]

**Procedure** writer

\[
\begin{align*}
P(\text{writer_mutex}) \\
\langle \text{write file} \rangle \\
V(\text{writer_mutex}) \\
\end{align*}
\]

\[
\begin{align*}
P(\text{reader_mutex}) \\
\text{readers} &= \text{readers} - 1 \\
\text{if readers} &= 0 \text{ then} \ V(\text{writer_mutex}) \\
V(\text{reader_mutex}) \\
\end{align*}
\]
Readers-Writers with reader’s priority

**Procedure reader**

P(reader_mutex)
if readers = 0 then
    readers = readers + 1
    P(writer_mutex)
else
    readers = readers + 1
V(reader_mutex)

<read file>

P(reader_mutex)
readers = readers - 1
if readers == 0 then V(writer_mutex)
V(reader_mutex)

**Procedure writer**

P(sr_mutex)
P(writer_mutex)

<write file>

V(writer_mutex)
V(sr_mutex)