Semaphores
Semaphore

Purpose: To prevent inefficient spin lock of the `await` statement.

Idea: Instead of spinning, the process calls a method that puts it in a special queue of PCBs, the “blocked on semaphore” queue.
The PCB contains additional information to help the operating system schedule the CPU. An example is a unique process identification number assigned by the system, labeled Process ID in Figure 8.18, that serves to reference the process. Suppose a user wants to terminate a process before it completes execution normally, and he knows the ID number is 782. He could issue a *KILL(782)* command that would cause the operating system to search through the queue of PCBs, find the PCB with ID 782, remove it from the queue, and deallocate it.

Another example of information stored in the PCB is a record of the total amount of CPU time used so far by the suspended process. If the CPU becomes available and the operating system must decide which of several suspended processes gets the CPU, it can use the recorded time to make a fair decision.

As a job progresses through the system toward completion, it passes through several states, as Figure 8.19 shows. The figure is in the form of a state transition diagram and is another example of a finite state machine. Each transition is labeled with the event that causes the change of state.

When a user submits a job for processing, the operating system creates a process for it by allocating a new PCB and attaching it to a queue of processes that are waiting for CPU time. It loads the program into main memory and sets the copy of PC in the PCB to the address of the first instruction of the process. That puts the job in the ready state.

Eventually, the operating system should select the job to receive some processing time. It sets the alarm clock to generate an interrupt after a quantum of time and puts the copies of the registers from the PCB into the CPU. That puts the job in the running state.

While in the running state, three things can happen: (1) The running process may time out if it is still executing when the alarm clock interrupts. If so, the operating system attaches the process’s PCB to the ready queue, which puts it back in the ready state. (2) The process may complete its execution normally, in which case the last instruction it executes is an *SVC* to request that the operating system terminate it. (3) The process may need some input, in which case it executes an *SVC* for the request. The operating system would transfer the request to the appropriate I/O device and put the PCB in another queue of processes that are waiting for their I/O operations to complete. That puts the process in the waiting-for-I/O state.
Add fourth state for a process

1. Ready in the ready queue
2. Running in the cpu
3. Waiting for I/O in the I/O-wait queue
4. Blocked in the blocked-on-semaphore queue
Figure 8.29

- Start
  - Create process
- Ready
  - I/O complete
  - signal(s)
- Waiting for I/O
  - I/O request
  - Select to run
- Running
  - Time out
  - wait(s)
- Finish
  - Terminate process
State Changes of a Process

inactive → ready ↔ running → completed

inactive → blocked

blocking
Semaphore type

Semaphore $S$ has two fields:

integer $S.V$

queue $S.L$
Semaphore atomic operations
Memorize this
Semaphore atomic operations
Memorize this

Initialization
semaphore $S \leftarrow (k, \emptyset)$
Semaphore atomic operations

Memorize this

**Initialization**

semaphore $S \leftarrow (k, \emptyset)$

wait($S$)

if $S.V > 0$

$S.V \leftarrow S.V - 1$

else

$S.L \leftarrow S.L \cup p$

$p.state \leftarrow$ blocked
Semaphore atomic operations

Memorize this

**Initialization**

semaphore $S \leftarrow (k, \emptyset)$

**wait($S$)**

if $S.V > 0$

\[ S.V \leftarrow S.V - 1 \]

else

\[ S.L \leftarrow S.L \cup p \]

$p.state \leftarrow \text{blocked}$

**signal($S$)**

if $S.L = \emptyset$

\[ S.V \leftarrow S.V + 1 \]

else

let $q$ be some process in $S.L$

\[ S.L \leftarrow S.L - \{q\} \]

$q.state \leftarrow \text{ready}$
Binary semaphore

The value $S.V$ is only allowed to be 0 or 1.

Also called “mutex” for mutual exclusion.
Critical section

The critical section problem is trivial when you have semaphores.
### Algorithm 6.1: Critical section with semaphores (two processes)

Binary semaphore $S \leftarrow (1, \emptyset)$

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p2: wait($S$)</td>
<td>q2: wait($S$)</td>
</tr>
<tr>
<td>p3: critical section</td>
<td>q3: critical section</td>
</tr>
<tr>
<td>p4: signal($S$)</td>
<td>q4: signal($S$)</td>
</tr>
</tbody>
</table>

Class exercise

Construct the first part of the state transition diagram:
\(p, p, q, q, p, p, (p \text{ and } q), \ldots\)

<table>
<thead>
<tr>
<th>Algorithm 6.1: Critical section with semaphores (two processes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary semaphore S (\leftarrow (1, \emptyset))</td>
</tr>
<tr>
<td>(p)</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: wait(S)</td>
</tr>
<tr>
<td>p3: critical section</td>
</tr>
<tr>
<td>p4: signal(S)</td>
</tr>
</tbody>
</table>

\(\text{wait}(S)\)

if \(S.V > 0\)

\[S.V \leftarrow S.V - 1\]

else

\[S.L \leftarrow S.L \cup p\]

\(p\.\text{state} \leftarrow \text{blocked}\)

\(\text{signal}(S)\)

if \(S.L = \emptyset\)

\[S.V \leftarrow S.V + 1\]

else

let \(q\) be some process in \(S.L\)

\[S.L \leftarrow S.L \setminus \{q\}\]

\(q\.\text{state} \leftarrow \text{ready}\)
States E and F have no transition on q because q is blocked. q3’ means q3 cannot execute.
Class exercise
Construct the complete state transition diagram

Algorithm 6.2: Critical section with semaphores (two proc., abbrev.)

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>binary semaphore $S \leftarrow (1, \emptyset)$</td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td>wait$(S)$</td>
<td>q1: wait$(S)$</td>
</tr>
<tr>
<td>p2</td>
<td>signal$(S)$</td>
<td>q2: signal$(S)$</td>
</tr>
<tr>
<td>p</td>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>1</td>
<td>wait$(S)$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>signal$(S)$</td>
<td></td>
</tr>
</tbody>
</table>

wait$(S)$
- if $S.V > 0$
  - $S.V \leftarrow S.V - 1$
- else
  - $S.L \leftarrow S.L \cup p$
  - $p.state \leftarrow$ blocked

signal$(S)$
- if $S.L = \emptyset$
  - $S.V \leftarrow S.V + 1$
- else
  - let $q$ be some process in $S.L$
  - $S.L \leftarrow S.L - \{q\}$
  - $q.state \leftarrow$ ready
Correctness
Correctness

Mutual exclusion: Yes, no state $p2, q2$
Correctness

Mutual exclusion: Yes, no state $p_2, q_2$

Deadlock free: Yes, no state with both blocked
Correctness

Mutual exclusion: Yes, no state $p2, q2$

Deadlock free: Yes, no state with both blocked

Starvation free: Yes (next slide)
    If $p$ executes $\text{wait}(S)$ either
    (a) $p$ not blocked, can execute $\text{signal}(S)$ so $q$ can proceed
    (b) $p$ blocked, $q$ will proceed
$p$ executes $\operatorname{wait}(S)$

\[
\begin{align*}
  p_1, q_1 & \quad (1, \emptyset) \\
  p_2, q_1 & \quad (0, \emptyset) \\
p & \quad (p) \\
  p_1, q_2 & \quad (0, \emptyset) \\
  p_2, q_2' & \quad (0, \{q\}) \\
  p_2, q_2' & \quad (0, \{p\})
\end{align*}
\]
Complete execution of \texttt{wait(S)}

When \texttt{wait(S)} causes a process to block, the \texttt{wait} statement has not been completely executed.

The \texttt{wait(S)} statement completes its execution when it is unblocked.
Complete execution of \( \text{wait}(S) \)

- \( p \) executes \( \text{wait}(S) \)

\[ p \]$\xrightarrow{p}$ $p2, q1$
\[ \begin{array}{c} p1, q1 \\ (1, \emptyset) \end{array} \]$\xrightarrow{q}$ $p2, q1$
\[ \begin{array}{c} p1, q2 \\ (0, \emptyset) \end{array} \]$\xrightarrow{q}$ $p2', q2$
\[ \begin{array}{c} p2, q2' \\ (0, \{ q \}) \end{array} \]
The strength of a semaphore

A strong semaphore uses a queue (FIFO) of blocked processes. The process unblocked is the one in the queue for the longest time.

A weak semaphore uses a set of blocked processes. The process unblocked is unpredictable.

The semaphore policy is distinct from the scheduling policy in the ready queue.
Semaphores in C--

Demo alg-6-1.cm

Output statements are intermingled because they are in the noncritical sections.
int n = 0;
semaphore s = 1;

void r() {
    int temp, i;
    for (i = 0; i < 10; i++) {
        // non-critical section
        cout << "r.i = " << i << endl;
        // preprotocol
        wait(s);
        // critical section
        temp = n;
        n = temp + 1;
        // postprotocol
        // postprotocol
        signal(s);
    }
}
```cpp
void q() {
    int temp, i;
    for (i = 0; i < 10; i++) {
        // non-critical section
        cout << "q.i = " << i << endl;
        // preprotocol
        wait(s);
        // critical section
        temp = n;
        n = temp + 1;
        // postprotocol
        signal(s);
    }
}

void main() {
    cobegin { r(); q(); }
    cout << "The value of n is " << n << "\n";
}
```
Semaphores in Java

Constructor

• The first parameter is the integer value of S.V

• The second optional parameter is true for fair scheduling (FIFO, a strong semaphore).

• The default value is false (a weak semaphore).
Semaphores in Java

Demo Alg0601.java

wait(S) is s.acquire()

signal(S) is s.release()

Output statements are rarely intermingled because the processor is so fast that the probability of an output statement completing in a single time slice is high.
Alg0601.java

import java.util.concurrent.*;

class Alg0601 extends Thread {

    static volatile int n = 0;
    static Semaphore s = new Semaphore(1);
    private int processID;

    Alg0601(int pID) {
        processID = pID;
    }
}
public void run() {
    int temp, delay;
    for (int i = 0; i < 10; i++) {
        try {
            // non-critical section
            System.out.println("p" + processID + ".i = " + i);
            // preprotocol
            s.acquire();
            // critical section
            delay = (int) (10 * Math.random());
            Thread.sleep(delay);
            temp = n;
            delay = (int) (10 * Math.random());
            Thread.sleep(delay);
            n = temp + 1;
            // postprotocol
            s.release();
        } catch (InterruptedException e) {
        }
    }
}
public static void main(String[] args) {
    Alg0601 p1 = new Alg0601(1);
    Alg0601 p2 = new Alg0601(2);
    p1.start();
    p2.start();
    try {
        p1.join();
        p2.join();
    } catch (InterruptedException e) {
    }
    System.out.println("The value of n is " + n);
}
Semaphore invariants

For proving correctness without model checking
Theorem 6.1
Memorize this.

Let

\[ k = \text{initial value of } S.V \geq 0 \]
\[ \#\text{signal}(S) = \text{the number of } signal(S) \text{ statements executed} \]
\[ \#\text{wait}(S) = \text{the number of } wait(S) \text{ statements completely executed} \]
Theorem 6.1
Memorize this.

Let

\[ k = \text{initial value of } S.V \geq 0 \]
\[ \#\text{signal}(S) = \text{the number of } signal(S) \text{ statements executed} \]
\[ \#\text{wait}(S) = \text{the number of } wait(S) \text{ statements completely executed} \]

The following expressions are invariant:

\[ S.V \geq 0 \]
\[ S.V = k + \#\text{signal}(S) - \#\text{wait}(S) \]
Theorem 6.1
Memorize this.

Let

\[ k = \text{initial value of } S.V \geq 0 \]
\[ \#\text{signal}(S) = \text{the number of } signal(S) \text{ statements executed} \]
\[ \#\text{wait}(S) = \text{the number of } wait(S) \text{ statements completely executed} \]

The following expressions are invariant:

\[ S.V \geq 0 \]
\[ S.V = k + \#\text{signal}(S) - \#\text{wait}(S) \]

Can be proved simply by mathematical induction.
Theorem 6.1

Ideas behind proof
Theorem 6.1

Ideas behind proof

\[ S.V \geq 0 \]

*wait*(\( S \)) only subtracts 1 from \( S.V \) when \( S.V > 0 \)

*signal*(\( S \)) only adds 1 to \( S.V \) when the queue is empty
Theorem 6.1

Ideas behind proof

\( S.V \geq 0 \)

wait\((S)\) only subtracts 1 from \( S.V \) when \( S.V > 0 \)
signal\((S)\) only adds 1 to \( S.V \) when the queue is empty

\[ S.V = k + \# \text{signal}(S) - \# \text{wait}(S) \]

If wait\((S)\) blocks a process, it does not subtract 1 from \( S.V \),
but its execution has not completed
If signal\((S)\) unblocks a process, it does not add 1 to \( S.V \),
but it also triggers the completion of a \#wait\((S)\) statement
Algorithm 6.2

Let $#CS$ be the number of processes in their critical sections

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p1$: wait($S$)

$p2$: signal($S$)

loop forever

$q1$: wait($S$)

$q2$: signal($S$)
Algorithm 6.2

Let $#CS$ be the number of processes in their critical sections

Lemma

$#CS + S.V = 1$

Proof

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p1$: wait($S$)

$p2$: signal($S$)

loop forever

$q1$: wait($S$)

$q2$: signal($S$)
Algorithm 6.2

Let \( \#CS \) be the number of processes in their critical sections

Lemma

\[
\#CS + S.V = 1
\]

Proof

From the code of Algorithm 6.2

\[
\#CS = \#\text{wait}(S) - \#\text{signal}(S)
\]
Algorithm 6.2

Let $\#CS$ be the number of processes in their critical sections

Lemma

$$\#CS + S.V = 1$$

Proof

From the code of Algorithm 6.2

$$\#CS = \#\text{wait}(S) - \#\text{signal}(S)$$

$$= \langle \text{Theorem 6.1 with } k = 1 \rangle$$
Algorithm 6.2

Let \#CS be the number of processes in their critical sections.

Lemma

\[ \#CS + S.V = 1 \]

Proof

From the code of Algorithm 6.2

\[ \#CS = \#\text{wait}(S) - \#\text{signal}(S) \]

\[ = \langle \text{Theorem 6.1 with } k = 1 \rangle \]

\[ \#CS = 1 - S.V \]

\[ = \langle \text{math} \rangle \]
Algorithm 6.2

Let $\#CS$ be the number of processes in their critical sections

Lemma

$\#CS + S.V = 1$

Proof

From the code of Algorithm 6.2

$\#CS = \#\text{wait}(S) - \#\text{signal}(S)$

$= \langle \text{Theorem 6.1 with } k = 1 \rangle$

$\#CS = 1 - S.V$

$= \langle \text{math} \rangle$

$\#CS + S.V = 1$  //

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

p1: wait(S)

p2: signal(S)

loop forever

q1: wait(S)

q2: signal(S)
Algorithm 6.2

Mutual exclusion

Proof

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p1$: wait($S$)

$p2$: signal($S$)

loop forever

$q1$: wait($S$)

$q2$: signal($S$)
Algorithm 6.2

Mutual exclusion

Proof

By the previous Lemma

\[ \#CS = 1 - S.V \]
Algorithm 6.2

Mutual exclusion

Proof

By the previous Lemma

#CS = 1 − S.V

⇒ ⟨Theorem 6.1, S.V ≥ 0⟩
Algorithm 6.2
Mutual exclusion

Proof

By the previous Lemma

#CS = 1 − S.V

⇒ 〈Theorem 6.1, S.V ≥ 0〉

#CS ≤ 1  //

binary semaphore S ← (1, ∅)

loop forever

p1:  wait(S)

p2:  signal(S)

loop forever

q1:  wait(S)

q2:  signal(S)
Algorithm 6.2

Deadlock free

Proof

binary semaphore S ← (1, Ø)

loop forever

p1: wait(S)
p2: signal(S)

loop forever

q1: wait(S)
q2: signal(S)
Algorithm 6.2

Deadlock free

Proof

Deadlock
Algorithm 6.2

Deadlock free

**Proof**

Deadlock

$\implies \langle \text{Code inspection} \rangle$

State is $p_1, q_1$, both blocked, and $S.V = 0$

**Proof**

Deadlock

$\implies \langle \text{Code inspection} \rangle$

State is $p_1, q_1$, both blocked, and $S.V = 0$
Algorithm 6.2

Deadlock free

Proof

Deadlock

⇒ ⟨Code inspection⟩

State is $p_1, q_1$, both blocked, and $S.V = 0$

⇒ ⟨Lemma, $#CS + S.V = 1$⟩

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p_1$: wait($S$)

$p_2$: signal($S$)

$q_1$: wait($S$)

$q_2$: signal($S$)
Algorithm 6.2

Deadlock free

Proof

Deadlock

⇒  ⟨Code inspection⟩

State is p1, q1, both blocked, and $S.V = 0$

⇒  ⟨Lemma, $\#CS + S.V = 1⟩$

$\#CS = 1$, Contradiction  //

Proof

Deadlock

⇒  ⟨Code inspection⟩

State is p1, q1, both blocked, and $S.V = 0$

⇒  ⟨Lemma, $\#CS + S.V = 1⟩$

$\#CS = 1$, Contradiction  //
Algorithm 6.2

Starvation free

Proof

binary semaphore S ← (1, ø)

loop forever
p1: wait(S)
p2: signal(S)

loop forever
q1: wait(S)
q2: signal(S)
Algorithm 6.2

Starvation free

Proof

$p$ is starved

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p_1$: wait($S$)

$p_2$: signal($S$)

loop forever

$q_1$: wait($S$)

$q_2$: signal($S$)
Algorithm 6.2

Starvation free

Proof

$p$ is starved

$\Rightarrow \langle p \text{ is blocked} \rangle$

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p1$: wait($S$)

$p2$: signal($S$)

loop forever

$q1$: wait($S$)

$q2$: signal($S$)
Algorithm 6.2

Starvation free

Proof

$p$ is starved

$\Rightarrow \langle p \text{ is blocked}\rangle$

$S = (0, \{p\}) \vee S = (0, \{p, q\})$

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p1$: wait($S$)

$p2$: signal($S$)

$q1$: wait($S$)

$q2$: signal($S$)
Algorithm 6.2

Starvation free

Proof

\[
p \text{ is starved} \\
\Rightarrow \langle p \text{ is blocked} \rangle \\
S = (0, \{p\}) \lor S = (0, \{p, q\}) \\
\Rightarrow \langle \text{Lemma, } \#CS = 1 - S.V = 1 - 0 = 1 \rangle
\]

binary semaphore \( S \leftarrow (1, \emptyset) \)
loop forever

\[
p1: \text{ wait}(S) \\
p2: \text{ signal}(S)
\]
loop forever

\[
q1: \text{ wait}(S) \\
q2: \text{ signal}(S)
\]
Algorithm 6.2

Starvation free

Proof

\[ p \text{ is starved} \]
\[ \Rightarrow \langle p \text{ is blocked} \rangle \]
\[ S = (0, \{ p \}) \lor S = (0, \{ p, q \}) \]
\[ \Rightarrow \langle \text{Lemma, } #CS = 1 - S.V = 1 - 0 = 1 \rangle \]
\[ #CS = 1, \text{ there is one process in its critical section} \]
Algorithm 6.2

Starvation free

Proof

$p$ is starved

$\Rightarrow \langle p \text{ is blocked} \rangle$

$S = (0, \{p\}) \lor S = (0, \{p, q\})$

$\Rightarrow \langle \text{Lemma, } \#CS = 1 - S.V = 1 - 0 = 1 \rangle$

$\Rightarrow \langle \#CS = 1, \text{ there is one process in its critical section} \rangle$

$\Rightarrow \langle \text{There are only two processes in the program} \rangle$

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

p1: wait(S)

p2: signal(S)

loop forever

q1: wait(S)

q2: signal(S)
Algorithm 6.2

Starvation free

Proof

\( p \) is starved

\[ \Rightarrow \langle p \text{ is blocked} \rangle \]

\[ S = (0, \{p\}) \lor S = (0, \{p, q\}) \]

\[ \Rightarrow \langle \text{Lemma, } \#CS = 1 - S.V = 1 - 0 = 1 \rangle \]

\#CS = 1, there is one process in its critical section

\[ \Rightarrow \langle \text{There are only two processes in the program} \rangle \]

\( q \) is in its critical section and \( S = (0, \{p\}) \)

binary semaphore \( S \leftarrow (1, \emptyset) \)

loop forever

p1: wait(S)
p2: signal(S)

loop forever

q1: wait(S)
q2: signal(S)
Algorithm 6.2

Starvation free

Proof

$p$ is starved

$\Rightarrow \langle p \text{ is blocked} \rangle$

$S = (0, \{p\}) \lor S = (0, \{p, q\})$

$\Rightarrow \langle \text{Lemma, } \#CS = 1 - S.V = 1 - 0 = 1 \rangle$

$\#CS = 1$, there is one process in its critical section

$\Rightarrow \langle \text{There are only two processes in the program} \rangle$

$q$ is in its critical section and $S = (0, \{p\})$

$\Rightarrow \langle q \text{ must execute } signal(S) \rangle$

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p1$: wait($S$)

$p2$: signal($S$)

$q1$: wait($S$)

$q2$: signal($S$)
Algorithm 6.2

Starvation free

Proof

$p$ is starved

$\Rightarrow \langle p$ is blocked$\rangle$

$S = (0, \{p\}) \lor S = (0, \{p, q\})$

$\Rightarrow \langle$ Lemma, $\# CS = 1 - S.V = 1 - 0 = 1 \rangle$

$\# CS = 1$, there is one process in its critical section

$\Rightarrow \langle$ There are only two processes in the program$\rangle$

$q$ is in its critical section and $S = (0, \{p\})$

$\Rightarrow \langle q$ must execute $\text{signal}(S)$\rangle

$p$ enters its critical section and is not starved  //

binary semaphore $S \leftarrow (1, \emptyset)$

loop forever

$p1$: wait$(S)$

$p2$: signal$(S)$

loop forever

$q1$: wait$(S)$

$q2$: signal$(S)$
The CS problem with more than two processes

<table>
<thead>
<tr>
<th>Algorithm 6.3: Critical section with semaphores ($N$ proc.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary semaphore $S \leftarrow (1, \emptyset)$</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: wait($S$)</td>
</tr>
<tr>
<td>p3: critical section</td>
</tr>
<tr>
<td>p4: signal($S$)</td>
</tr>
</tbody>
</table>
The CS problem with more than two processes

<table>
<thead>
<tr>
<th>Algorithm 6.3: Critical section with semaphores ($N$ proc.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary semaphore $S \leftarrow (1, \emptyset)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop forever</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: wait($S$)</td>
</tr>
<tr>
<td>p3: critical section</td>
</tr>
<tr>
<td>p4: signal($S$)</td>
</tr>
</tbody>
</table>

Mutual exclusion: yes
The CS problem with more than two processes

<table>
<thead>
<tr>
<th>Algorithm 6.3: Critical section with semaphores ($N$ proc.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary semaphore $S \leftarrow (1, \emptyset)$</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: wait($S$)</td>
</tr>
<tr>
<td>p3: critical section</td>
</tr>
<tr>
<td>p4: signal($S$)</td>
</tr>
</tbody>
</table>

Mutual exclusion: yes
Deadlock free: yes
The CS problem with more than two processes

<table>
<thead>
<tr>
<th>Algorithm 6.3: Critical section with semaphores ($N$ proc.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary semaphore $S \leftarrow (1, \emptyset)$</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: wait($S$)</td>
</tr>
<tr>
<td>p3: critical section</td>
</tr>
<tr>
<td>p4: signal($S$)</td>
</tr>
</tbody>
</table>

Mutual exclusion: yes
Deadlock free: yes
Starvation free: Only if the semaphore is strong.
Algorithm 6.4: Critical section with semaphores ($N$ proc., abbrev.)

<table>
<thead>
<tr>
<th>binary semaphore $S \leftarrow (1, \emptyset)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: $\text{wait}(S)$</td>
</tr>
<tr>
<td>p2: $\text{signal}(S)$</td>
</tr>
</tbody>
</table>
Scenario for Starvation

<table>
<thead>
<tr>
<th>n</th>
<th>Process p</th>
<th>Process q</th>
<th>Process r</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1: wait(S)</td>
<td>q1: wait(S)</td>
<td>r1: wait(S)</td>
<td>(1, ∅)</td>
</tr>
<tr>
<td>2</td>
<td>p2: signal(S)</td>
<td>q1: wait(S)</td>
<td>r1: wait(S)</td>
<td>(0, ∅)</td>
</tr>
<tr>
<td>3</td>
<td>p2: signal(S)</td>
<td>q1: blocked</td>
<td>r1: wait(S)</td>
<td>(0, {q})</td>
</tr>
<tr>
<td>4</td>
<td>p1: signal(S)</td>
<td>q1: blocked</td>
<td>r1: blocked</td>
<td>(0, {q, r})</td>
</tr>
<tr>
<td>5</td>
<td>p1: wait(S)</td>
<td>q1: blocked</td>
<td>r2: signal(S)</td>
<td>(0, {q})</td>
</tr>
<tr>
<td>6</td>
<td>p1: blocked</td>
<td>q1: blocked</td>
<td>r2: signal(S)</td>
<td>(0, {p, q})</td>
</tr>
<tr>
<td>7</td>
<td>p2: signal(S)</td>
<td>q1: blocked</td>
<td>r1: wait(S)</td>
<td>(0, {q})</td>
</tr>
</tbody>
</table>
Concurrent merge sort

Sort the first half and the second half concurrently.

Constrain the scheduling so that the merge operation can start only after the two sorts have completed.

This is a prerequisite scheduling problem.
Algorithm 6.5: Mergesort

integer array A
binary semaphore S1 ← (0, ∅)
binary semaphore S2 ← (0, ∅)

<table>
<thead>
<tr>
<th>sort1</th>
<th>sort2</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: sort 1st half of A</td>
<td>q1: sort 2nd half of A</td>
<td>r1: wait(S1)</td>
</tr>
<tr>
<td>p2: signal(S1)</td>
<td>q2: signal(S2)</td>
<td>r2: wait(S2)</td>
</tr>
<tr>
<td>p3:</td>
<td>q3:</td>
<td>r3: merge halves of A</td>
</tr>
</tbody>
</table>
The producer consumer problem

Assumptions
The producer consumer problem

Assumptions

• Operation append(d, buffer) appends data d
The producer consumer problem

Assumptions
• Operation append(d, buffer) appends data d
• Operation take(buffer) deletes an item and returns it
The producer consumer problem

Assumptions

• Operation append(d, buffer) appends data d

• Operation take(buffer) deletes an item and returns it

• The capacity of buffer is infinite
The producer consumer problem

Assumptions

• Operation append(d, buffer) appends data d

• Operation take(buffer) deletes an item and returns it

• The capacity of buffer is infinite

• Must not delete from an empty buffer
### Algorithm 6.6: Producer-consumer (infinite buffer)

<table>
<thead>
<tr>
<th></th>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>infinite queue of dataType buffer</td>
<td>← empty queue</td>
<td></td>
</tr>
<tr>
<td>semaphore notEmpty</td>
<td>← (0, ∅)</td>
<td></td>
</tr>
<tr>
<td><strong>producer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dataType d</td>
<td></td>
<td>dataType d</td>
</tr>
<tr>
<td>loop forever</td>
<td></td>
<td>loop forever</td>
</tr>
<tr>
<td>p1:</td>
<td>d ← produce</td>
<td>q1:</td>
</tr>
<tr>
<td>p2:</td>
<td>append(d, buffer)</td>
<td>q2:</td>
</tr>
<tr>
<td>p3:</td>
<td>signal(notEmpty)</td>
<td>q3:</td>
</tr>
<tr>
<td><strong>consumer</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Class exercise

Construct the beginning of the state transition diagram for p, p, q, q, ... and q, p, p, ... for the abbreviated algorithm.

<table>
<thead>
<tr>
<th>Algorithm 6.7: Producer-consumer (infinite buffer, abbreviated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>infinite queue of dataType buffer ← empty queue</td>
</tr>
<tr>
<td>semaphore notEmpty ← (0, Ø)</td>
</tr>
<tr>
<td>producer</td>
</tr>
<tr>
<td>dataType d</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: append(d, buffer)</td>
</tr>
<tr>
<td>p2: signal(notEmpty)</td>
</tr>
</tbody>
</table>
The producer-consumer problem

Invariant

In state $p1, q1, nonEmpty.V = \#buffer$
The producer-consumer problem

Invariant

In state $p1, q1$, $nonEmpty.V = \#\text{buffer}$

Proof by mathematical induction

Ideas behind proof:
The producer-consumer problem

Invariant

In state $p1, q1$, $nonEmpty.V = \#buffer$

Proof by mathematical induction

Ideas behind proof:

When the producer produces, it executes $signal(notEmpty)$, which adds 1 to $nonEmpty.V$
The producer-consumer problem

Invariant

In state $p1, q1$, $nonEmpty.V = \#\text{buffer}$

Proof by mathematical induction

Ideas behind proof:

When the producer produces, it executes $\text{signal}(notEmpty)$, which adds 1 to $nonEmpty.V$

When the consumer consumes, it executes $\text{wait}(notEmpty)$, which subtracts 1 from $nonEmpty.V$
The bounded buffer producer-consumer problem
The bounded buffer producer-consumer problem

Assumptions

- The capacity of buffer is finite
The bounded buffer producer-consumer problem

Assumptions

• The capacity of buffer is finite

• Must not delete from an empty buffer

• Must not insert into a full buffer
The bounded buffer producer-consumer problem

Assumptions
• The capacity of buffer is finite
• Must not delete from an empty buffer
• Must not insert into a full buffer

Implementation
• A circular queue (Exercise for the student)
**Algorithm 6.19: Producer-consumer (circular buffer)**

```plaintext
dataType array [0..N] buffer
integer in, out ← 0
semaphore notEmpty ← (0, Ø)
semaphore notFull ← (N, Ø)
```

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dataType d</code></td>
<td><code>dataType d</code></td>
</tr>
<tr>
<td><code>loop forever</code></td>
<td><code>loop forever</code></td>
</tr>
<tr>
<td>p1:</td>
<td>q1:</td>
</tr>
<tr>
<td><code>d ← produce</code></td>
<td><code>wait(notEmpty)</code></td>
</tr>
<tr>
<td>p2:</td>
<td>q2: <code>d ← buffer[out]</code></td>
</tr>
<tr>
<td><code>wait(notFull)</code></td>
<td>p3: <code>buffer[in] ← d</code></td>
</tr>
<tr>
<td>p4:</td>
<td>q3: <code>out ← (out+1) modulo N</code></td>
</tr>
<tr>
<td><code>in ← (in+1) modulo N</code></td>
<td>q4: <code>signal(notFull)</code></td>
</tr>
<tr>
<td>p5:</td>
<td>q5: <code>consume(d)</code></td>
</tr>
<tr>
<td><code>signal(notEmpty)</code></td>
<td></td>
</tr>
</tbody>
</table>
```
The dining philosophers
Activity of each philosopher

- Pick up forks
- Eat
- Put down forks
- Think
# Algorithm 6.9: Dining philosophers (outline)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td></td>
</tr>
<tr>
<td>p1:</td>
<td>think</td>
</tr>
<tr>
<td>p2:</td>
<td>preprotocol</td>
</tr>
<tr>
<td>p3:</td>
<td>eat</td>
</tr>
<tr>
<td>p4:</td>
<td>postprotocol</td>
</tr>
</tbody>
</table>
Dining philosophers implementation

- Each philosopher is a process
- Each fork is a semaphore
Array of forks and philosophers

- $ph[0]$
- $ph[1]$
- $ph[2]$
- $ph[3]$
- $ph[4]$
- $f[0]$
- $f[1]$
- $f[2]$
- $f[3]$
### Algorithm 6.10: Dining philosophers (first attempt)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>semaphore array [0..4] fork ← [1,1,1,1,1]</td>
</tr>
<tr>
<td></td>
<td>loop forever</td>
</tr>
<tr>
<td>p1:</td>
<td>think</td>
</tr>
<tr>
<td>p2:</td>
<td>wait(fork[i])</td>
</tr>
<tr>
<td>p3:</td>
<td>wait(fork[i+1])</td>
</tr>
<tr>
<td>p4:</td>
<td>eat</td>
</tr>
<tr>
<td>p5:</td>
<td>signal(fork[i])</td>
</tr>
<tr>
<td>p6:</td>
<td>signal(fork[i+1])</td>
</tr>
</tbody>
</table>


Slide 6.16
Algorithm 6.10

Mutual exclusion

Proof

semaphore array [0..4] fork ← [1,1,1,1,1]
loop forever
p1: think
p2: wait(fork[i])
p3: wait(fork[i+1])
p4: eat
p5: signal(fork[i])
p6: signal(fork[i+1])
Algorithm 6.10

Mutual exclusion

Proof

By code inspection and mathematical induction, the number of philosophers holding fork \( i \) is

\[
#P_i = \#\text{wait}(\text{fork}[i]) - \#\text{signal}(\text{fork}[i])
\]
Algorithm 6.10

Mutual exclusion

Proof

By code inspection and mathematical induction, the number of philosophers holding fork $i$ is

$$#P_i = \#\text{wait}(\text{fork}[i]) - \#\text{signal}(\text{fork}[i])$$

$$= \langle \text{Theorem (6.1)} \#\text{wait} - \#\text{signal} = 1 - S.V \rangle$$

$$#P_i = 1 - S.V$$
Algorithm 6.10

Mutual exclusion

Proof

By code inspection and mathematical induction, the number of philosophers holding fork $i$ is

$$
#P_i = \#\text{wait}(\text{fork}[i]) - \#\text{signal}(\text{fork}[i])
$$

$$
= \langle \text{Theorem (6.1)} \#\text{wait} - \#\text{signal} = 1 - S.V \rangle
$$

$$
#P_i = 1 - S.V
$$

$$
\Rightarrow \langle \text{Theorem (6.1)} S.V \geq 0 \rangle
$$

$$
#P_i \leq 1 \quad //
$$
Algorithm 6.10

Deadlock free: No

Proof

semaphore array [0..4] fork ← [1,1,1,1,1]
loop forever

p1: think
p2: wait(fork[i])
p3: wait(fork[i+1])
p4: eat
p5: signal(fork[i])
p6: signal(fork[i+1])
Algorithm 6.10

Deadlock free: No

Proof

$P_0$ picks up fork 0 on her left.
$P_1$ picks up fork 1 on his left.
$P_2$ picks up fork 2 on her left.
$P_3$ picks up fork 3 on his left.
$P_4$ picks up fork 4 on her left.

And now no philosopher can pick up his fork on his right.
Algorithm 6.11
Solves the deadlock problem by simulating a room with a room semaphore that only allows four philosophers in the room at the same time.

To be starvation free, the room semaphore must be strong, but the fork semaphores can be weak.
Algorithm 6.11: Dining philosophers (second attempt)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>semafor array [0..4] fork ← [1,1,1,1,1]</strong></td>
<td></td>
</tr>
<tr>
<td><strong>semafor room ← 4</strong></td>
<td></td>
</tr>
<tr>
<td><strong>loop forever</strong></td>
<td></td>
</tr>
<tr>
<td>p1:</td>
<td>think</td>
</tr>
<tr>
<td>p2:</td>
<td>wait(room)</td>
</tr>
<tr>
<td>p3:</td>
<td>wait(fork[i])</td>
</tr>
<tr>
<td>p4:</td>
<td>wait(fork[i+1])</td>
</tr>
<tr>
<td>p5:</td>
<td>eat</td>
</tr>
<tr>
<td>p6:</td>
<td>signal(fork[i])</td>
</tr>
<tr>
<td>p7:</td>
<td>signal(fork[i+1])</td>
</tr>
<tr>
<td>p8:</td>
<td>signal(room)</td>
</tr>
</tbody>
</table>
Algorithm 6.12

Solves the deadlock problem by having the fourth philosopher pick up his right fork first, and then his left fork. If he blocks on picking up his right fork, his left fork will be available for philosopher 3.

This is an asymmetric solution. One philosopher acts differently from the others.
### Algorithm 6.12: Dining philosophers (third attempt)

<table>
<thead>
<tr>
<th>philosopher 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: think</td>
</tr>
<tr>
<td>p2: wait(fork[0])</td>
</tr>
<tr>
<td>p3: wait(fork[4])</td>
</tr>
<tr>
<td>p4: eat</td>
</tr>
<tr>
<td>p5: signal(fork[0])</td>
</tr>
<tr>
<td>p6: signal(fork[4])</td>
</tr>
</tbody>
</table>

Semaphore array [0..4] fork ← [1,1,1,1,1]