Inter-Process Communication

Semaphores

The action of process A waiting for process B to finish something before A can continue is very common

The action of process A waiting for process B to finish something before A can continue is very common

E.g., waiting for data to be written to an area of shared memory

Inter-Process Communication

Semaphores

The action of process A waiting for process B to finish something before A can continue is very common

E.g., waiting for data to be written to an area of shared memory

It is a very simple form of IPC

Inter-Process Communication

Semaphores

The action of process A waiting for process B to finish something before A can continue is very common

E.g., waiting for data to be written to an area of shared memory

It is a very simple form of IPC

Signals can be used, but an alternative is to use a *semaphore*

Inter-Process Communication

Semaphores

The action of process A waiting for process B to finish something before A can continue is very common

E.g., waiting for data to be written to an area of shared memory

It is a very simple form of IPC

Signals can be used, but an alternative is to use a *semaphore*

A signal is appropriate when you want to continue computing on something else while waiting; a semaphore is for *pausing* and waiting (i.e., blocked)

Inter-Process Communication

Semaphores

Invented by Dijkstra, semaphores have been used widely for may years

Inter-Process Communication

Semaphores

Invented by Dijkstra, semaphores have been used widely for may years

A semaphore is a variable whose value can only be accessed and altered by two operations V and P (Dijkstra is Dutch)

Invented by Dijkstra, semaphores have been used widely for may years

A semaphore is a variable whose value can only be accessed and altered by two operations V and P (Dijkstra is Dutch)

Alternative names are: signal and wait; post and wait; raise and lower; up and down; lock and unlock and others

Inter-Process Communication

Let S be a semaphore variable, usually residing in a chunk of shared memory

Inter-Process Communication

Let S be a semaphore variable, usually residing in a chunk of shared memory

Start with $S = 1$

Inter-Process Communication

Let S be a semaphore variable, usually residing in a chunk of shared memory

Start with $S = 1$

$P(S)$:
if $S = 1$ then set $S = 0$
else block on S

Inter-Process Communication

Let S be a semaphore variable, usually residing in a chunk of shared memory

Start with $S = 1$

$P(S)$:
if $S = 1$ then set $S = 0$
else block on S

$V(S)$:
if one or more processes are blocking on S then allow one to proceed
else set $S = 1$

Inter-Process Communication

Let S be a semaphore variable, usually residing in a chunk of shared memory

Start with $S = 1$

$P(S)$:
if $S = 1$ then set $S = 0$
else block on S

$V(S)$:
if one or more processes are blocking on S then allow one to proceed
else set $S = 1$

(There are many technical issues we are ignoring here. . .)

Inter-Process Communication

Semaphores

For synchronisation:

```
P(S)
...modify a resource...
V(S)
```

```
P(S) # wait for resource
...use resource...
V(S)
```

The second process will wait until the first has done a V to signal the resource is ready

If multiple processes attempt a $P(S)$ simultaneously only *one* will succeed and continue; the others will be blocked

Inter-Process Communication

Semaphores

If multiple processes attempt a $P(S)$ simultaneously only *one* will succeed and continue; the others will be blocked

So if we have code like

```
P(S)
some code
V(S)
```

being run by multiple processes using the shared semaphore S, only one process can execute the code at a time; the others will be blocked and get their turn later

Inter-Process Communication

Semaphores

If multiple processes attempt a $P(S)$ simultaneously only *one* will succeed and continue; the others will be blocked

So if we have code like

```
wait(S)
some code
signal(S)
```

being run by multiple processes using the shared semaphore S, only one process can execute the code at a time; the others will be blocked and get their turn later

More suggestively using names signal and wait (**not** the same signal as in signals, earlier!)

Inter-Process Communication

Semaphores

Generally, the code would be to access some shared resource (often shared memory, e.g., B shouldn't read until A has finished writing), so the semaphore makes sure only one process can access the resource at a time

Inter-Process Communication

Semaphores

Generally, the code would be to access some shared resource (often shared memory, e.g., B shouldn't read until A has finished writing), so the semaphore makes sure only one process can access the resource at a time

The protected code is called a *critical section*: it is critical that only one process runs it at a time

Inter-Process Communication

Semaphores

Generally, the code would be to access some shared resource (often shared memory, e.g., B shouldn't read until A has finished writing), so the semaphore makes sure only one process can access the resource at a time

The protected code is called a *critical section*: it is critical that only one process runs it at a time

```
P(S)                P(S)
...resource...      ...same resource...
V(S)                V(S)
```

To be effective, all accesses to the resource must be protected by the semaphore

Inter-Process Communication

Semaphores

This is a *binary semaphore*, as it take just two values, 0 and 1

Inter-Process Communication

Semaphores

This is a *binary semaphore*, as it take just two values, 0 and 1

There is a simple generalisation to a *counting semaphore*

Inter-Process Communication

Semaphores

This is a *binary semaphore*, as it take just two values, 0 and 1

There is a simple generalisation to a *counting semaphore*

Start with $S = n$

$P(S)$:
if $S > 0$ then set $S = S - 1$ else
block on S

$V(S)$:
if one or more processes are blocking on S then allow one to proceed
else set $S = S + 1$

Inter-Process Communication

Semaphores

This is a *binary semaphore*, as it take just two values, 0 and 1

There is a simple generalisation to a *counting semaphore*

Start with $S = n$

$P(S)$:
if $S > 0$ then set $S = S - 1$ else
block on S

$V(S)$:
if one or more processes are blocking on S then allow one to proceed
else set $S = S + 1$

This allows no more than n processes into the region at once

Semaphores were first used within OS kernels to protect shared resources but can be used in user programs to protect resources there, too: for example, a chunk of shared memory (e.g., shared memory IPC)

Inter-Process Communication

Semaphores

Correct implementation of user mode semaphores is very hard

Correct implementation of user mode semaphores is very hard

We have to ensure that it works even if

Inter-Process Communication

Semaphores

Correct implementation of user mode semaphores is very hard

We have to ensure that it works even if

1. the process is rescheduled in the middle between the test and the decrement of the count

Inter-Process Communication

Semaphores

Correct implementation of user mode semaphores is very hard

We have to ensure that it works even if

1. the process is rescheduled in the middle between the test and the decrement of the count
2. there are multiple parallel processors accessing the semaphore simultaneously

Inter-Process Communication

Semaphores

Correct implementation of user mode semaphores is very hard

We have to ensure that it works even if

1. the process is rescheduled in the middle between the test and the decrement of the count
2. there are multiple parallel processors accessing the semaphore simultaneously

Exercise. Read about the implementation of semaphores

Semaphores are widely used

Inter-Process Communication

Semaphores

Semaphores are widely used

- each semaphore only needs a few bytes of shared memory

Inter-Process Communication

Semaphores

Semaphores are widely used

- each semaphore only needs a few bytes of shared memory
- they are small and fast given hardware support

Inter-Process Communication

Semaphores

Semaphores are widely used

- each semaphore only needs a few bytes of shared memory
- they are small and fast given hardware support
- and OK in software

Inter-Process Communication

Semaphores

Semaphores are widely used

- each semaphore only needs a few bytes of shared memory
- they are small and fast given hardware support
- and OK in software
- used both in OSs and user programs to protect critical resources

Inter-Process Communication

Semaphores

Semaphores are widely used

- each semaphore only needs a few bytes of shared memory
- they are small and fast given hardware support
- and OK in software
- used both in OSs and user programs to protect critical resources
- and are widely available in POSIX libraries

Inter-Process Communication

Semaphores

On the other hand, semaphores are a very low-level mechanism and it is easy to cause deadlock

Inter-Process Communication

Semaphores

On the other hand, semaphores are a very low-level mechanism and it is easy to cause deadlock

Suppose we have semaphore S_1 protecting file F_1 and semaphore S_2 protecting file F_2. Process A wants to read from F_1 and write to F_2, while process B wants to read from F_2 and write to F_1

Inter-Process Communication

Semaphores

On the other hand, semaphores are a very low-level mechanism and it is easy to cause deadlock

Suppose we have semaphore S_1 protecting file F_1 and semaphore S_2 protecting file F_2. Process A wants to read from F_1 and write to F_2, while process B wants to read from F_2 and write to F_1

To make things consistent in the read/writes, both processes must grab both semaphores

Inter-Process Communication

Semaphores

On the other hand, semaphores are a very low-level mechanism and it is easy to cause deadlock

Suppose we have semaphore S_1 protecting file F_1 and semaphore S_2 protecting file F_2. Process A wants to read from F_1 and write to F_2, while process B wants to read from F_2 and write to F_1

To make things consistent in the read/writes, both processes must grab both semaphores

- Process A grabs semaphore S_1

Inter-Process Communication

Semaphores

On the other hand, semaphores are a very low-level mechanism and it is easy to cause deadlock

Suppose we have semaphore S_1 protecting file F_1 and semaphore S_2 protecting file F_2. Process A wants to read from F_1 and write to F_2, while process B wants to read from F_2 and write to F_1

To make things consistent in the read/writes, both processes must grab both semaphores

- Process A grabs semaphore S_1
- Process B grabs semaphore S_2

Inter-Process Communication

Semaphores

On the other hand, semaphores are a very low-level mechanism and it is easy to cause deadlock

Suppose we have semaphore S_1 protecting file F_1 and semaphore S_2 protecting file F_2. Process A wants to read from F_1 and write to F_2, while process B wants to read from F_2 and write to F_1

To make things consistent in the read/writes, both processes must grab both semaphores

- Process A grabs semaphore S_1
- Process B grabs semaphore S_2
- A tries to grab S_2 and blocks

Inter-Process Communication

Semaphores

On the other hand, semaphores are a very low-level mechanism and it is easy to cause deadlock

Suppose we have semaphore S_1 protecting file F_1 and semaphore S_2 protecting file F_2. Process A wants to read from F_1 and write to F_2, while process B wants to read from F_2 and write to F_1

To make things consistent in the read/writes, both processes must grab both semaphores

- Process A grabs semaphore S_1
- Process B grabs semaphore S_2
- A tries to grab S_2 and blocks
- B tries to grab S_1 blocks

Exercise. Identify the four conditions for deadlock in the above

Exercise: use a counting semaphore to solve the Dining Philosopher's problem

□