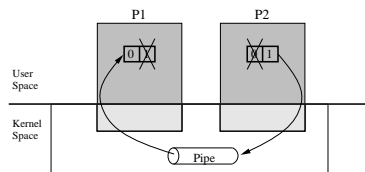


## Labinstruction 2, Process Synchronization

- Process communication
- Shared memory
- Semaphores
- Labinstruction
- Signals ... forgotten slides

## Process communication



Mechanisms:

- **pipes** - have we already looked at, no structured information.
- **message queues** - like pipes but with structured messages.
- **shared memory** - several processes can write and read to the same memory block.
- **semaphores** - used to synchronize access to common data structures.
- **signals** - software interrupts. Processes can define how to handle signals.

## System V IPC

IPC = Interprocess Communication. A set of mechanisms for: Message Queues, Shared memory and Semaphores.

## Message Queues

- Similar to pipes - they provide us with a mechanism to allow related processes to communicate.
- While a pipe can only be used to send "flat" messages (i.e. stream model), message queues can be used to send structured messages.
- We will not be looking at message queues in the lab course.

## Shared memory

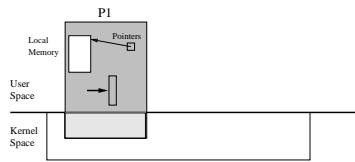
Problem:

- Global variables and memory obtained with `malloc(3)` are local to the calling process.
- Processes created with `fork()` get only *copies* of all data, so subsequent changes are local to the process that made them.
- Sometimes we want to allow separate cooperating processes to share access to the *same* data structure, i.e. changes are seen by other processes.

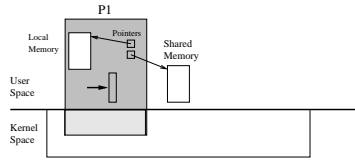
**Shared memory** addresses these problems.

- One process allocates shared memory for common data structures, maintains pointer.
- Other processes created using `fork()` get copy of pointer but *same* allocated block.
- Subsequent changes to structure are seen by all processes.

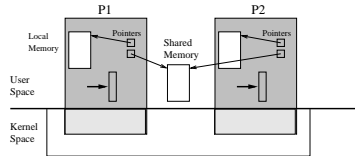
## Shared memory, continued.



After P1 has allocated local memory.



After P2 has allocated shared memory.



After a new process, P2, has been created using `fork()`. P2 gets a copy of P1's local memory block. Both P1 and P2 can access the same shared memory block. There is no difference in the usage of the different memories.

## Shared memory, continued

- `malloc(3)` is a way of dynamically allocating memory: `void *malloc(size_t m);`
- `malloc(3)` returns a void pointer to a block of at least `size` bytes suitably aligned, or `NULL` on failure. The size of `m` is normally calculated with the `sizeof` operator to be movable between architectures.
- The pointer has to be type converted to a pointer of a suitable memory type:

```
int *ip;          /* Declare an integer pointer */
ip = (int *) malloc(100 * sizeof(int));
                /* Allocate a block of size of 100 integers */
ip[5] = 3;       /* Put 3 in the 6:th element */
```
- For shared memory we have two operations defined, corresponding to the standard library functions `malloc(3)` and `free(3)`:
  - `void *shmalloc(size_t size);` - allocate a block of size bytes of shared memory.
  - `int shfree(void *ptr);` - release a block of shared memory.
- There is no manual pages available for these shared memory functions.

## Synchronization

We need a way to synchronize access to the common data objects.  
Code example:

```
/* Some initialization */
int *sh_var = NULL;
sh_var = (int *) shmalloc(sizeof(int));
*sh_var = 1;

/* Create a new process */
fork();

/* Both processes goes here */
if(*sh_var == 1)
{
    do_something();
    *sh_var = (*sh_var) + 1;
}
```

Indeterministic behaviour:

`sh_var` might both have 2 and 3 as its final value.

- `sh_var = 2`: if one process runs the `if`- statement, calls `do_something()` and increases `sh_var` without getting interrupted.
- `sh_var = 3`: if one gets interrupted after the `if`- statement but before the increase of `sh_var`. The second one will also run the body of the `if`- statement, because `sh_var` still is equal to 1. Both processes will finally increase the `sh_var` variable.

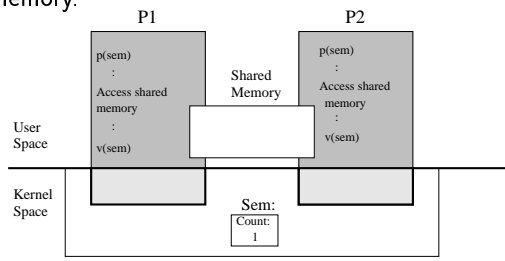
## Semaphores

Semaphores are used to synchronize access to common data structures, such as those created using the shared memory facility.

- Abstract datatype with two operations defined on it `semwait (or p)` and `semsignal (or v)`.
- Each semaphore has an internal counter to represent the number of available resources, e.g how many processes are allowed to simultaneous access a object.
- Each time a process does `semwait` on a semaphore, the counter is decremented with 1 and the process is allowed to proceed. However, if the counter was at zero, then `semwait` will cause the calling process to block until another process increments the counter again.
- `semsignal` increments the counter with 1, and if any processes are blocked waiting for the semaphore at the time, one of them (but we don't know which one) will be unblocked and allowed to proceed.

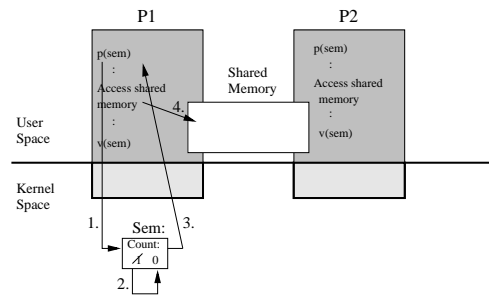
## Semaphores continued

A semaphore can be used to synchronize access to shared memory.



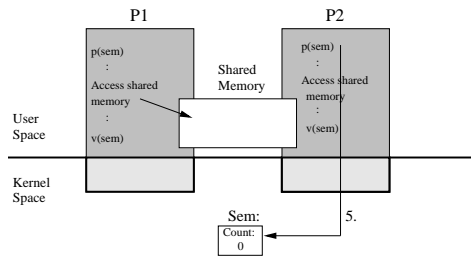
- **Mutual exclusion:** Only one process at a time shall have access to an object. By using a semaphore with 1 as its internal counter only one process at a time can access the shared memory.
- **Critical sections:** The sections in the code where the shared memory is accessed. These must mutual exclusive between the processes.

## Semaphores continued



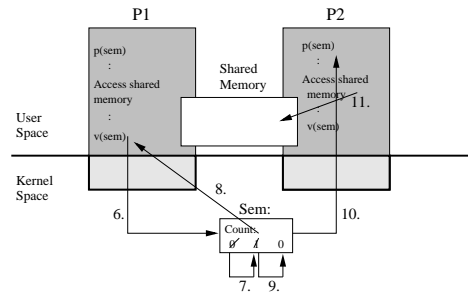
1. Process P1 wants to read from shared memory and makes a `p(sem)` system call.
2. The OS gets control and decreases the internal counter of the semaphore.
3. The control is returned to P1.
4. P1 can access and manipulate the shared memory.

## Semaphores continued



5. Process P2 also wants to read from the shared memory and makes a `p(sem)` system call. But when the internal counter of the semaphore is equal to zero the calling process will be blocked.

## Semaphores continued



6. When P1 doesn't want to access the shared memory any more it makes a `v(sem)` call.
7. The OS gets control and increases the internal counter of the semaphore.
8. The control is returned to P1 that continues executing.
9. Next time P2 gets scheduled by OS the OS sees that it is suspended on the semaphore and that the internal counter of the semaphore is equal to 1. The OS decreases the internal counter of the semaphore.
10. The control is returned to P2.
11. P2 can access and manipulate the shared memory.

## Semaphores continued

The standard operations on semaphores are:

- `semid_t semcreate(int n);` – create a semaphore with an initial count of `n`.
- `int semwait(semid_t sem);` – obtain the semaphore `sem`.
- `int semsignal(semid_t sem);` – release the semaphore `sem`.
- `int semdestroy(semid_t sem);` – free the semaphore `sem`.

These operations, like `shmalloc()` and `shfree()`, are declared in `process_sync/ipc.h` in the course directory `/stud/docs/kurs/os/`.

## Lab 2 - Process Synchronization

This lab consists of three parts (as before):

- Using the tools that are available
- Adding synchronization to an existing program
- Writing a program that solves one of the “classical” synchronization problems

## Tools

Goal: learn how to examine and manage IPC objects.

The two tools we will use are:

- `ipcs` – examine IPC objects
- `ipcrm` – remove IPC objects

Steps:

- Read the manual pages!
- Use the test program found in the course directory to create IPC objects
- IPC is an expensive resource - determine system limits

## Adding Synchronization

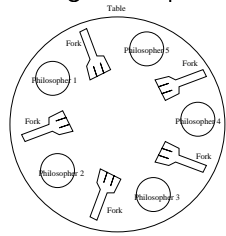
Goal: Put synchronization in the right places to make the pencil program work appropriately.

- The program simulates a pencil factory.
- The program creates a number of processes to make pencils, and one process to provide them with supplies when necessary.
- Read the section in the lab handout on synchronisation theory for an introduction.
- The compiled version in the course directory works correctly.
- Add necessary synchronisation mechanisms only in `manufacture` and `supply` functions.
- Not necessary to hand in entire program, just these two functions.

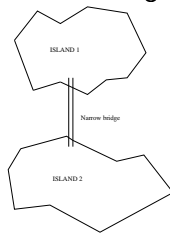
## Classical synchronization problem

Choose one of the two suggested problems:

### Dining Philosophers



### Narrow Bridge

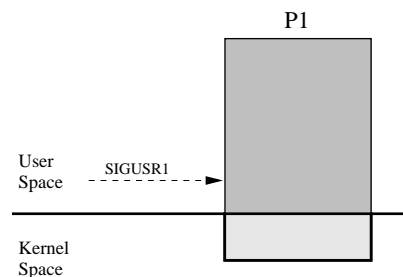


You need to solve the problem and show informally that it is correct:

- Describe the necessary conditions for deadlock and how to avoid it
- Explain deadlock and starvation in terms of the problem you are solving
- Show why your solution is deadlock free
- Show how you have avoided starvation

## Signals

- *Signals* are short (one byte) messages sent to a process, usually in response to some abnormal event.  
For example: `SIGKILL == 9`.
- Signals are asynchronous and can be sent to a process at anytime, independent of the process state.
- The effect of receiving a signal is much like the occurrence of a software trap.
- There are 29 system defined signals.
- There are 2 user defined signals.



Process P1 get a SIGUSR1 signal.

## Compiling

The “real” interface to the System V IPC functions is rather complex so a simple one has been provided for the lab. It is declared in `ipc.h` and defined in `libipc.a`.

Compile: `make`

## Finally

Remove all IPC objects!!!!!!!!!!

They will not disappear when you log out!

## Handling Signals

A process receiving a signal can act in a number of different ways, in particular:

- it can perform the *default action* associated with the signal.
- it can *ignore* the signal. This means that all such signals are to be discarded (i.e. never delivered).
- it can *block* the signal. The kernel queues blocked signals for delivery at some later time.
- it can *catch* the signal, by installing a special *signal handler* to respond to and deal with the signal.

There are 2 notable exceptions to this rule: `SIGKILL` and `SIGSTOP` cannot be caught, blocked or ignored – we must use the default action.

## Default Actions

The default action for most signals is to terminate. In addition, many of the signals cause the receiving process to dump core.

These signals cause the process to terminate:

- SIGHUP hangup
- SIGINT interrupt
- SIGKILL kill (cannot be caught, blocked, or ignored)
- SIGPIPE write on a pipe or other socket with no one to read it
- SIGTERM software termination signal
- SIGXCPU cpu time limit exceeded
- SIGUSR1 user-defined signal 1
- SIGUSR2 user-defined signal 2

These cause the process to terminate and dump core:

- SIGQUIT quit
- SIGTRAP trace trap
- SIGABRT abort
- SIGFPE arithmetic exception
- SIGBUS bus error
- SIGSYS bad argument to system call

## Signal Actions

If a process wishes to *ignore* a particular signal, say SIGUSR1, it must do the following:

```
signal(SIGUSR1, SIG_IGN);
```

It can restore the default action like this:

```
signal(SIGUSR1, SIG_DFL);
```

A signal can be blocked, i.e. queued for delivery at a later time:

```
mask=sigmask(SIGUSR1);  
org_mask=sigblock(mask);
```

If the process later wants the default action for the signal to be reinstated, it can do this:

```
sigsetmask(org_mask);
```

At this time, queued signals will be delivered.

## Default Actions

Some of the signals cause the process to stop, i.e. block until allowed to continue:

- SIGSTOP stop (cannot be caught, blocked, or ignored)
- SIGTSTP stop signal generated from keyboard
- SIGTTIN background read attempted from control terminal
- SIGTTOU background write attempted to control terminal

These signals are normally ignored:

- SIGCONT continue after stop
- SIGURG urgent condition present on socket
- SIGCHLD child status has changed
- SIGIO I/O is possible on a descriptor
- SIGWINCH window changed

## Catching signals

To *catch* a signal, we must first declare (and define) a function to act as a signal handler:

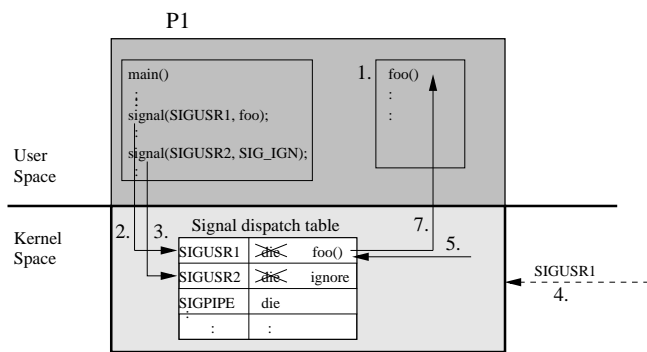
```
void sigusr1_handler()  
{  
    fprintf(stderr, "Ouch!\n");  
}
```

Then the process must register the function with the kernel:

```
signal(SIGUSR1, sigusr1_handler);
```

Later, when the process receives the signal (SIGUSR1 in this case), the function `sigusr1_handler` will be called. Different handlers can be created for different signals.

## Catching signals, continued



1. `foo()` is declared.
2. `main()` calls `signal(SIGUSR1, foo)` which declares `foo()` as SIGUSR1 signal handler.
3. `main()` calls `signal(SIGUSR2, SIG_IGN)` which tells the kernel to ignore the SIGUSR2 signal.
4. Later the process gets a SIGUSR1 signal.
5. The signal handler for SIGUSR1, will be looked up in the signal dispatch table.
6. (The system reinstalls the default handler for SIGUSR1.)
7. The function `foo()` is found and called.

## Sending Signals

We can send some of the more common signals directly from the keyboard.

`stty` shows us (among other things) what keys to use:

```
$ stty -a
[...]
susp  intr  quit
^Z    ^C    ^\
```

Here we see that the following mappings are in effect:

Ctrl-Z ⇒ SIGTSTP

Ctrl-C ⇒ SIGINT

Ctrl-\ ⇒ SIGQUIT

When we press any of these keys, the corresponding signal is sent to the foreground process in that window.

## Sending Signals, cont.

We can use the program `kill(1)` to send signals to our processes.

Send SIGTERM to a process with process identifier 16743:

```
$ kill 16743
```

Send SIGKILL to a process (can't be ignored):

```
$ kill -KILL 16743 or kill -9 16743
```

Stop a process:

```
$ kill -STOP 16743
```

Continue a process after SIGSTOP:

```
$ kill -CONT 16743
```

## Sending Signals, cont.

We can use the system call `kill(2)` to send signals between processes:

```
#include <signal.h>
kill(pid, SIGUSR1);
```

(We assume here that the receiving process handles SIGUSR1)

Whenever a child process exits, a SIGCHLD signal is automatically sent to the parent. This gives us a simple way to avoid zombies:

```
void child_handler()
{
    int status;
    wait(&status);
    signal(SIGCHLD, child_handler);
}
```

```
main() {
    ...
    signal(SIGCHLD, child_handler);
    ...
}
```

## Delivering Signals

Delivering a signal is done by the kernel, and consists of two steps:

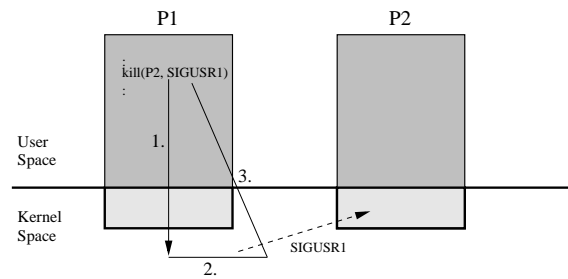
- posting the signal
- delivering the signal

*Posting* a signal means that the kernel makes note, in the pcb of the receiving process, that a signal is to be delivered. A signal that is posted but not yet delivered is said to be *pending*.

At each entry into the system (i.e. when a system call is invoked by the process) the list of pending signals is checked by the process. through a call to `psig()`.

If signals are pending, the process `psig()` takes appropriate action (i.e. default action or signal handler). At this point the signal is said to be *delivered*.

## Delivering Signals, continued



Process P1 is sending signal SIGUSR1 to process P2.

1. P1 makes a `kill(P2, SIGUSR1)` system call.
2. The OS starts executing and posts the signal SIGUSR1 to the P2 process.
3. Control is returned to P1.
4. When P2 gets scheduled the list of pending processes are checked by the process.