Operating Systems 2024/2025

TP Class 03 – Shared Memory and Semaphores

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Some slides partially based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva.

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Abbreviation: OS

Source: Dictionary.com

1 2 9 0
Departamento de Engenharia Informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Updated on
06 February 2025

INTERPROCESS COMMUNICATIONS OVERVIEW

IPC - Interprocess Communications

Introduction

- How to enable communication between processes?
  - Until now the only option seen was by using common files or passing open files across forks.

- Efforts were made to standardize IPCs across different Unix implementations and other OSs
  - Some standards:
    - IEEE POSIX (Portable Operating System Interface for Unix)
    - SUS (Single Unix Specification)

IPC - Interprocess Communications

Introduction

- Comparing IPC facilities - includes communication and synchronization facilities between processes or threads

Facility type	Name used to identify object	Handle used to refer to object in programs
Pipe	no name	file descriptor
FIFO	pathname	file descriptor
UNIX domain socket	pathname	file descriptor
Internet domain socket	IP address + port number	file descriptor
System V message queue	System V IPC key	System V IPC identifier
System V semaphore	System V IPC key	System V IPC identifier
System V shared memory	System V IPC key	System V IPC identifier
POSIX message queue	POSIX IPC pathname	*mqd_t* (message queue descriptor)
POSIX named semaphore	POSIX IPC pathname	*sem_t* * (semaphore pointer)
POSIX unnamed semaphore	no name	*sem_t* * (semaphore pointer)
POSIX shared memory	POSIX IPC pathname	file descriptor
Anonymous mapping	no name	none
Memory-mapped file	pathname	file descriptor
flock() lock	pathname	file descriptor
fcntl() lock	pathname	file descriptor

Covered in the course

Source: "The Linux Programming Interface", Michael Kerrisk, No Starch Press, 2010

IPC - Interprocess Communications

Introduction - System V IPCs

- System V IPCs
  - Some history:
    - appeared in late 70s, in *Columbus UNIX*, a Bell UNIX for database and efficient transaction processing
    - In 1983 were used in *System V* what made them popular in mainstream UNIX-es
    - In 2001, SUSv3 is published and require implementation of all of them for XSI conformance (so, they are also called XSI IPC)
  - Includes:
    - **SysV Message Queues**
      - used to pass messages between processes
    - **SysV Semaphores**
      - used for process synchronization
    - **SySV Shared Mem**ory
      - used to share memory regions

IPC - Interprocess Communications

Introduction - System V IPCs

- Overview of System V IPC API

Aspect	Msg queues	Semaphores	Shared memory
Include file	<sys/**msg**.h>	<sys/**sem**.h>	<sys/**shm**.h>
Data type	**msq**id_ds	**sem**id_ds	**shm**id_ds
Create or open	**msg**get	**sem**get	**shm**get
Control operation	**msg**ctl	**sem**ctl	**shm**ctl
IPC operations	**msg**snd / **msg**rvc	**sem**op	**shm**at / **shm**dt

#include <sys/types.h>
#include <sys/ipc.h>

IPC - Interprocess Communications

Introduction - System V IPCs

- System V object persistence
  - System V IPC objects have kernel persistence: they remain available until kernel shutdown or explicit deletion
  - Advantages
    - processes can access the object, change its state, and then exit without having to wait; other processes can come up later and check the (modified) state
  - Disadvantages
    - IPC objects consume system resources and cannot be automatically garbage collected
      - hence the need of enforcing limits on their quantity
    - it’s hard to determine when it is safe to delete an object

IPC - Interprocess Communications

Introduction - System V IPCs

- Shell manipulation of IPC objects
  - ipcs
    - lists available System V IPC objects
  - ipcs -l
    - shows system limits on IPC object counts
  - ipcrm
    - deletes IPC objects (that the user owns)

  - On Linux, /proc/sysvipc/ provides a view on all existing IPC objects

IPC - Interprocess Communications

System V IPCs – `kill_ipc.sh`

- An example of a shell script to automatically clean SysV IPCs

```
#!/bin/bash

ME=`whoami`

IPCS_S=`ipcs -s | egrep "0x[0-9a-f]+ [0-9]+" | grep $ME | cut -f2 -d" "`
IPCS_M=`ipcs -m | egrep "0x[0-9a-f]+ [0-9]+" | grep $ME | cut -f2 -d" "`
IPCS_Q=`ipcs -q | egrep "0x[0-9a-f]+ [0-9]+" | grep $ME | cut -f2 -d" "`

for id in $IPCS_M; do
  ipcrm -m $id;
done

for id in $IPCS_S; do
  ipcrm -s $id;
done

for id in $IPCS_Q; do
  ipcrm -q $id;
done
```

Note: This script is available in class demos

Remember

- Always clean up!!!

```
user@UbuntuMachine:~$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00000000 32768      user       770        0            0
0x00000000 65537      user       770        0            0
0x00000000 98306      user       770        0            0

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 2785297    user       766        4          0
0x00000000 2818066    user       766        4          0
0x00000000 2850835    user       766        4          0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 65536      user       777        3
0x00000000 98305      user       777        3
0x00000000 131074     user       777        3

user@UbuntuMachine:~$
```

IPC - Interprocess Communications

Introduction - POSIX IPCs

- The POSIX.1b real-time extensions defined a set of IPC mechanisms that are analogous to the System V IPC mechanisms

- It implements:
  - Message queues
  - Shared memory
  - Semaphores (thread safe!!)

IPC - Interprocess Communications

Introduction - POSIX IPCs

- Overview of POSIX IPC API

Interface	Message queues	Semaphores	Shared memory
Header file	`<mqueue.h>`	`<semaphore.h>`	`<sys/mman.h>`
Object handle	*mqd_t*	*sem_t* *	*int* (file descriptor)
Create/open	*mq_open()*	*sem_open()*	*shm_open()* + *mmap()*
Close	*mq_close()*	*sem_close()*	*munmap()*
Unlink	*mq_unlink()*	*sem_unlink()*	*shm_unlink()*
Perform IPC	*mq_send()*, *mq_receive()*	*sem_post()*, *sem_wait()*, *sem_getvalue()*	operate on locations in shared region
Miscellaneous operations	*mq_setattr()*—set attributes *mq_getattr()*—get attributes *mq_notify()*—request notification	*sem_init()*—initialize unnamed semaphore *sem_destroy()*—destroy unnamed semaphore	(none)

Source: “The Linux Programming Interface”, Michael Kerrisk, No Starch Press, 2010

IPC - Interprocess Communications

Introduction - POSIX IPCs

- Shell manipulation of IPC objects (in Linux)
  - POSIX shared memory and semaphores are visible at:
    `/dev/shm`
  - They can be viewed with command `ls`, and deleted with `rm`

IPC - Interprocess Communications

Introduction - System V IPCs vs POSIX IPCs

- POSIX IPC advantages:
  - Simpler interface.
  - The use of names instead of keys, together with the open, close , and unlink functions, is more consistent with the traditional UNIX file model.
  - POSIX IPC objects are reference counted. It will be destroyed only when all processes have closed it.
  - POSIX IPC interfaces are all multithread safe.

IPC - Interprocess Communications

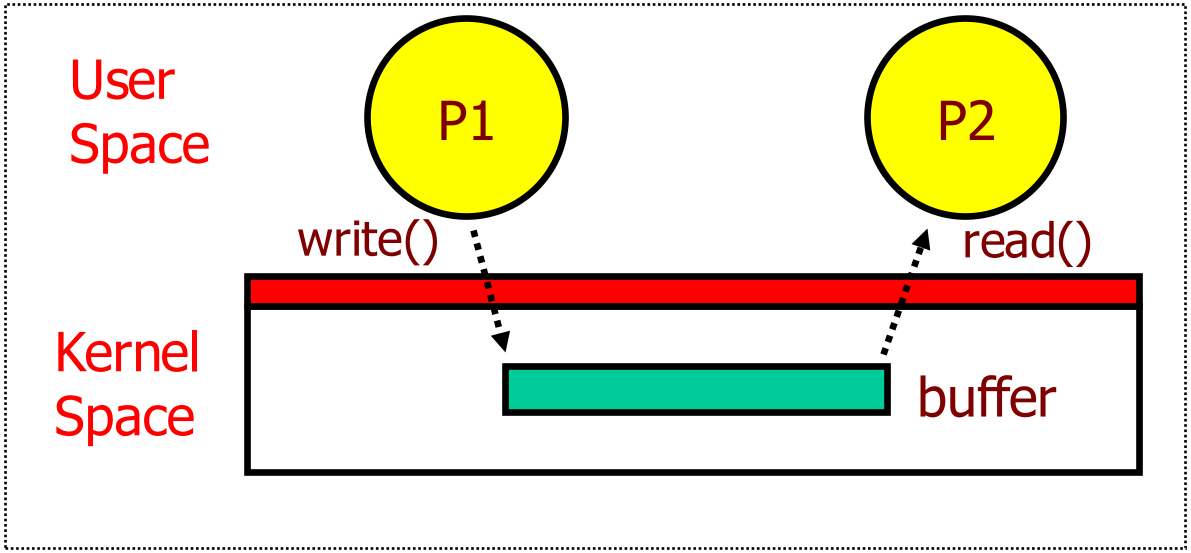
Introduction - System V IPCs vs POSIX IPCs

- System V advantages:
  - Portability
    - System V IPC is specified in SUSv3 and supported on nearly every UNIX implementation. By contrast, each of the POSIX IPC mechanisms is an optional component in SUSv3. Some UNIX implementations don't support (all of) the POSIX IPC mechanisms.
      - E.g. In Linux, a full implementation of POSIX semaphores is available only since kernel 2.6

SYSTEM V SHARED MEMORY

Why shared memory?

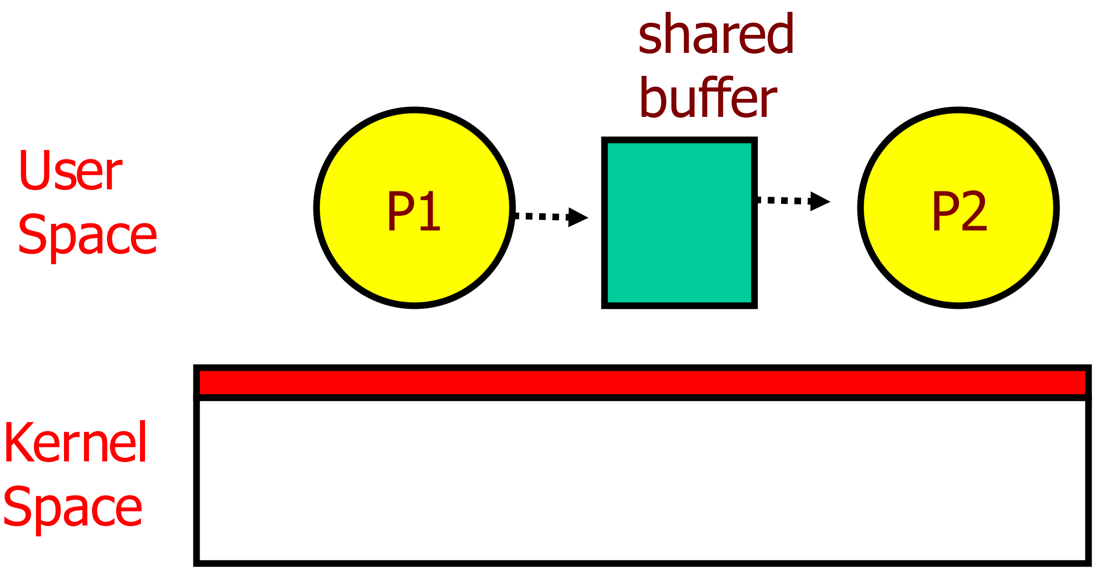
- We already know that...
  - System calls are slow!
  - Copying thought the kernel is slow!



Why shared memory?

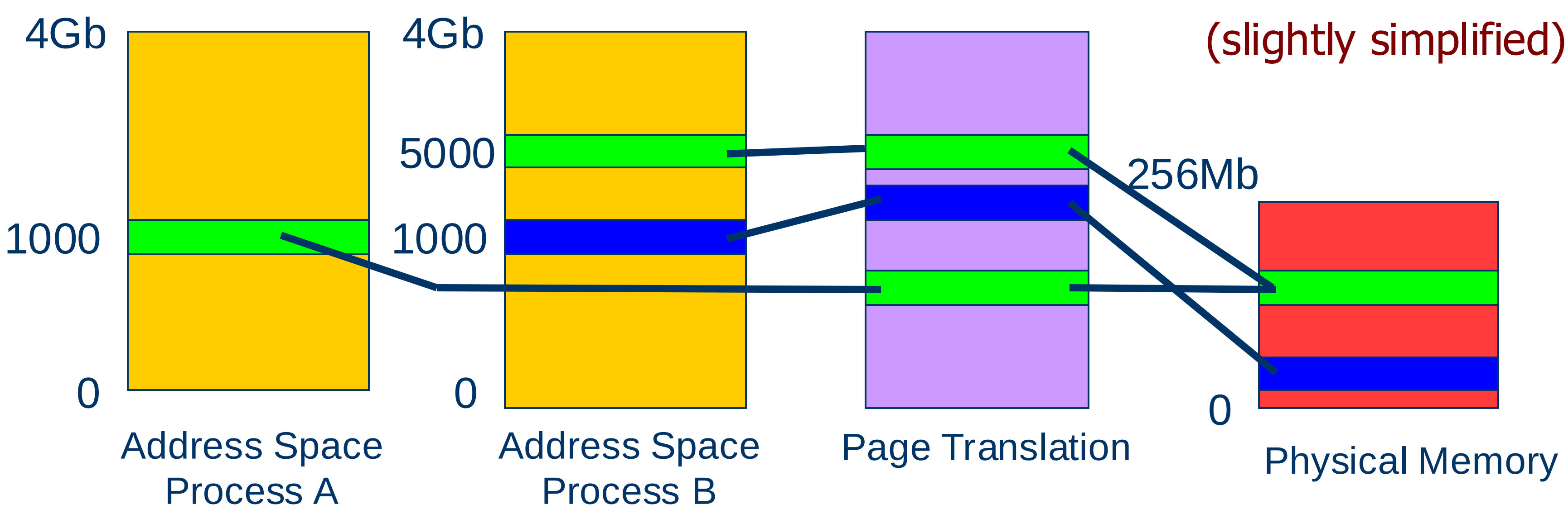
- Shared Memory
  - (Almost) No kernel involvement!
  - Fast! Very Fast!

Dangerous, very dangerous!



How does it work

- Each process has an address space
  - Each address space corresponds to a page table. There are as many page tables as there are processes
- Shared memory corresponds to putting the **same** "*real memory pages*" in the page tables of two different processes



Shared Memory – System V

- `int shmget(key_t key, int size, int flags)`
  - Obtains an identifier to an existing shared memory or creates a new one.
    - “key” can be IPC_PRIVATE (which creates a new unique identifier), or an existing identifier. ftok() can be used to generate a number based on a filename.
    - “size” its the shared memory size in bytes
      - The new shared memory segment size is equal to the value of “size” **rounded up to a multiple of PAGE_SIZE**
    - “flags”, normal mode flags. When ORed with IPC_CREAT creates a new one.
      - When using IPC_CREAT <u>always define the permissions</u> of the new shared memory
      - IPC_CREAT
        - Create a new segment. If this flag is not used, the shmget() will find the segment associated with key and check to see if the user has permission to access the segment.
      - IPC_EXCL
        - This flag is used with IPC_CREAT to ensure that this call creates the segment. If the segment already exists, the call fails.

Shared Memory – System V

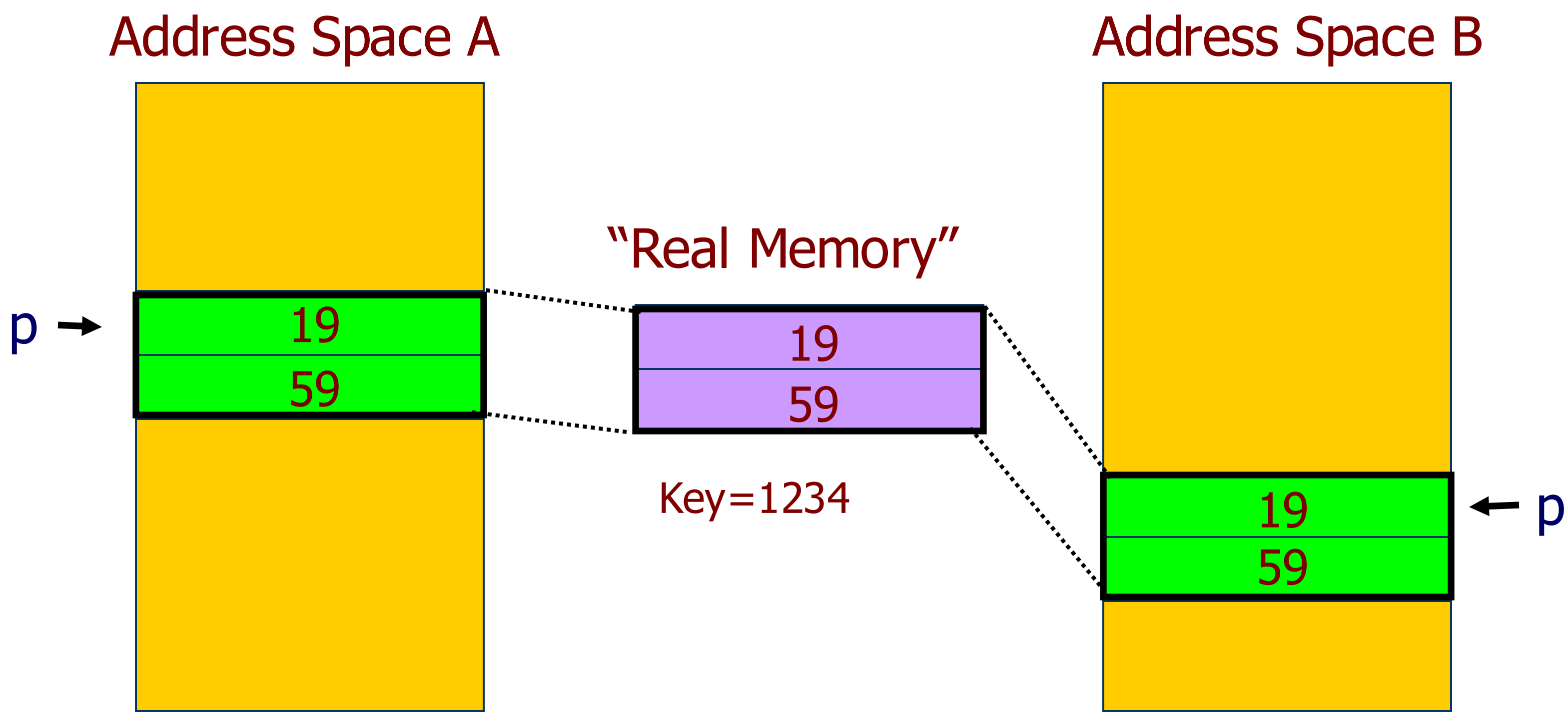
- `int` **`shmctl`**`(int shmid, int cmd, struct shmid_ds* buff)`
  - Provides a variety of control operations on the shared memory.
    - “shmid” is the value returned by shmget()
    - “cmd” is the command (most usually: IPC_RMID to remove it)
    - “buff” a structure used in some control operations

Shared Memory – System V (2)

- `void *`**`shmat`**`(int shmid, const void* where, int flags)`
  - Maps a certain shared memory region into the current process address space.
    - “shmid” represents the shared memory identifier “shmid” returned by shmget()
    - “where” represents an unused address space location where to map the shared memory (normally, use NULL)
    - “flags” represent different ways of doing the mapping (typically 0)

- `int `**`shmdt`**`(const void* where)`
  - Unmaps a certain shared memory region from the current address space.
    - “where” represents an address space location where the shared memory was mapped

How attaching works




```
int id = shmget(1234, 2, IPC_CREAT|0777)
char* p = shmat(id, NULL, 0);
p[0] = 19;
```

Process A

```
int id = shmget(1234, 2, 0777)
char* p = shmat(id, NULL, 0);
p[1] = 59;
```

Process B

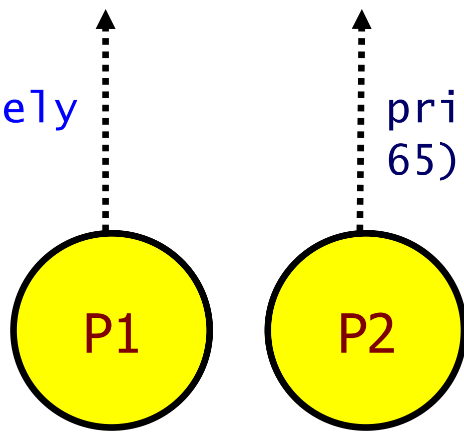
SYNCHRONIZATION OF PROCESSES
SEMAPHORES (SYSTEM V & POSIX)

What's wrong with this routine?

```
void print_work(const char* work, int user) {
    send_to_printer("--- JOB of %d ---\n", user);
    send_to_printer("%s\n",work);
    send_to_printer("--- END OF JOB ---\n");
}
```

```
print_work("This is a lovely
poem from 12 who has been
writing a lot.", 12);
```

```
print_work("I hate bad poets.",
65);
```



Two processes will execute the routine

```
--- JOB of 12 ---
This is a lovely poem from 12
--- JOB of 65 ---
I hate who has been writing a lot. bad poets.
--- END OF JOB ---
--- END OF JOB ---
```

Synchronization – Semaphores

- A semaphore is a synchronization object
  - Controlled access to a counter (a value)
  - Two operations are supported: *wait() and post()*
  - Can also be used as a resource counter – to control access to finite resources!
- **wait()**
  - If the semaphore is positive, decrement it and continue
  - If not, block the calling process (thread)
- **post()**
  - Increment the semaphore value
  - If there was any process (thread) blocked due to the semaphore, unblock one of them.

Corrected version

```
void print_work(const char* work, int user) {
    sem_wait(MUTEX);
    send_to_printer("--- JOB of %d ---\n", user);
    send_to_printer("%s\n",work);
    send_to_printer("--- END OF JOB ---\n");
    sem_post(MUTEX);
}
```

Mutual Exclusion:
Only one process can be in here!

You always have to synchronize, even if you are only reading or writing one byte!

Semaphores System V and POSIX

- UNIX System V Semaphores (aka Process Semaphores)
  - Works with semaphore arrays
  - **`semget()`, `semctl()`, `semop`**()
  - A little bit hard to use by themselves!
  - May block a process and all the threads in it!
- POSIX Semaphores
  - Quite easy to use
  - **`sem_open()`, `sem_init()`, `sem_close()`, `sem_post()`, `sem_wait()`**
  - Also work with threads!
  - Two options:
    - Named semaphores
    - Unnamed semaphores in shared memory
    - (Prior to kernel 2.6, Linux only supported unnamed, thread-shared semaphores)

System V semaphores

- SysV semaphores functions

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

  - Create a set of semaphores

```
int semget(key_t key, int nsems, int semflg);
```

  - Semaphore control operations

```
int semctl(int semid, int semnum, int cmd, ...);
  Note: May have 3 or 4 arguments
```

  - Semaphore operations

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
int semtimedop(int semid, struct sembuf *sops, unsigned
nsops, struct timespec *timeout);
```

System V semaphores

The implementation of *semlib*

- *Semlib* is a library that was created to simplify the use of System V semaphores
- It uses System V semaphores and simplifies its main operations
- Can be used in classes – direct use of System V primitives is also possible
- To use *semlib* include *semlib.c* when compiling
  - Eg. `gcc -Wall prog.c semlib.c -o prog`

- Do not use SysV semaphores to synchronize threads!
  (not all implementations are thread safe!!)

System V semaphores

The implementation of *semlib* – `semlib.h`

- semlib.h :

```
// Obtains a new array of initialized semaphores
extern int sem_get(int nsem, int init_val);

// Removes a semaphore set
extern void sem_close(int sem_id);

// Performs a wait operation on a semaphore
extern void sem_wait(int sem_id, int sem_num);

// Performs a signal operation on a semaphore
extern void sem_post(int sem_id, int sem_num);

// Initializes the value of a semaphore
extern void sem_setvalue(int sem_id, int sem_num, int value);
```

System V semaphores

The implementation of *semlib* – `semlib.c` (1)

- semlib.c :

```
// Obtains a new array of initialized semaphores
int sem_get(int nsem, int init_val)
{
  int id;
  int i;

  if ( (id=semget(IPC_PRIVATE, nsem, IPC_CREAT|0777)) == -1 )
  {
    perror("Could not get the semafore set!");
    return -1;
  }

  for (i=0; i<nsem; i++)
    sem_setvalue(id, i, init_val);

  return id;
}

// Removes a semaphore set
void sem_close(int sem_id)
{

  semctl(sem_id, 0, IPC_RMID, 0);
}
```

System V semaphores

The implementation of *semlib* – `semlib.c` (2)

```
// Performs a wait operation on a semaphore
void sem_wait(int sem_id, int sem_num)
{
  struct sembuf op;

  op.sem_num = sem_num;
  op.sem_op  = -1;
  op.sem_flg = 0;

  if ( semop(sem_id, &op, 1) == -1 )
  {
    perror("Could not do the wait on the semaphore");
  }
}

// Performs a signal operation on a semaphore
void sem_post(int sem_id, int sem_num)
{
  struct sembuf op;

  op.sem_num = sem_num;
  op.sem_op  = +1;
  op.sem_flg = 0;

  if ( semop(sem_id, &op, 1) == -1)
  {
      perror("Could not do the signal on the semaphore");
  }
}
```

System V semaphores

The implementation of *semlib* – `semlib.c` (3)

```c
// Initializes the value of a semaphore
void sem_setvalue(int sem_id, int sem_num, int value)
{
  union semun val;
  val.val = value;

  if ( semctl(sem_id, sem_num, SETVAL, val) == -1 )
  {
      perror("Could not set the value on the semaphore");
  }
}
```

POSIX semaphores

- POSIX semaphores can be named or unnamed
  - Unnamed semaphores are allocated in process memory and initialized;
  - Named semaphores are referenced with a pathname.

- Basic functions for <u>named</u> and <u>unnamed</u> POSIX semaphores

```
#include <semaphore.h>

int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec
*abs_timeout);
int sem_getvalue(sem_t *sem, int *sval)
```

These functions return 0 on success, or -1 on error

POSIX semaphores

Unnamed semaphores

- Must use shared memory for inter-process synchronization or internal memory for inter-thread synchronization

- Creation of an unnamed semaphore
  - The semaphore is initialized at the address pointed by *sem*. The *value* argument specifies the initial value for the semaphore. *pshared* specifies if if the semaphore will be shared between threads in a process, or between processes.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- Destroy an unnamed semaphore

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

POSIX semaphores
Named semaphores

- Named Semaphores use an identifier which non-related process can access
  - In Linux named semaphores are created in a virtual file system normally mounted in `/dev/shm` with names like `sem.name`

- Creation of a named semaphore

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
unsigned int value);
```

  - On On success, sem_open() returns the address of the new semaphore; On error, sem_open() returns SEM_FAILED, with errno set to indicate the error.
    - E.g.

```
//must have #include <errno.h>
mutex_shm = sem_open("MUTEX_SHM", O_CREAT | O_EXCL , 0700, 1);
if(mutex_shm == SEM_FAILED){
    fprintf(stderr, "sem_open() failed.  errno:%d\n", errno);
    }
```

POSIX semaphores

Named semaphores

- Closing a named semaphore (removes association with a semaphore)

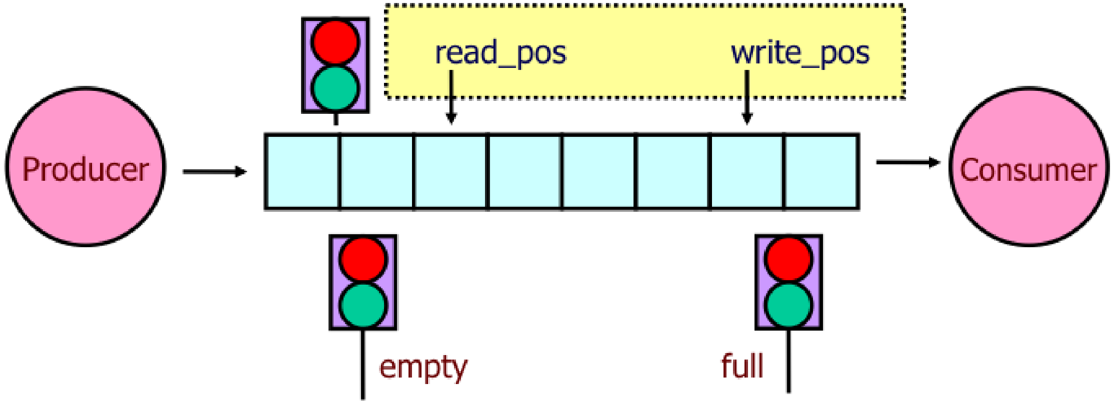
```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

- Deleting a named semaphore

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

Example of semaphores usage
Using a Producer/Consumer problem

- Example of a Producer/Consumer problem solved using System V (using the given semlib library), POSIX unnamed and POSIX named semaphores.

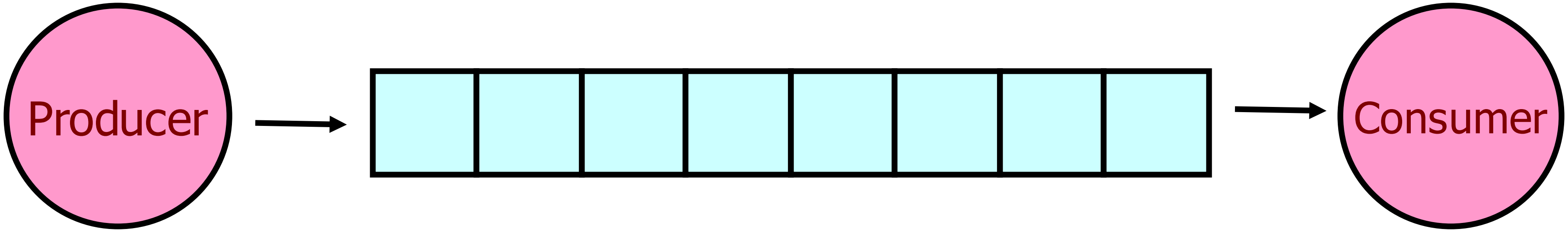


Producer/Consumer problem and solution has been detailed in Theoretical classes!

Example of semaphores usage

Producer/Consumer problem review

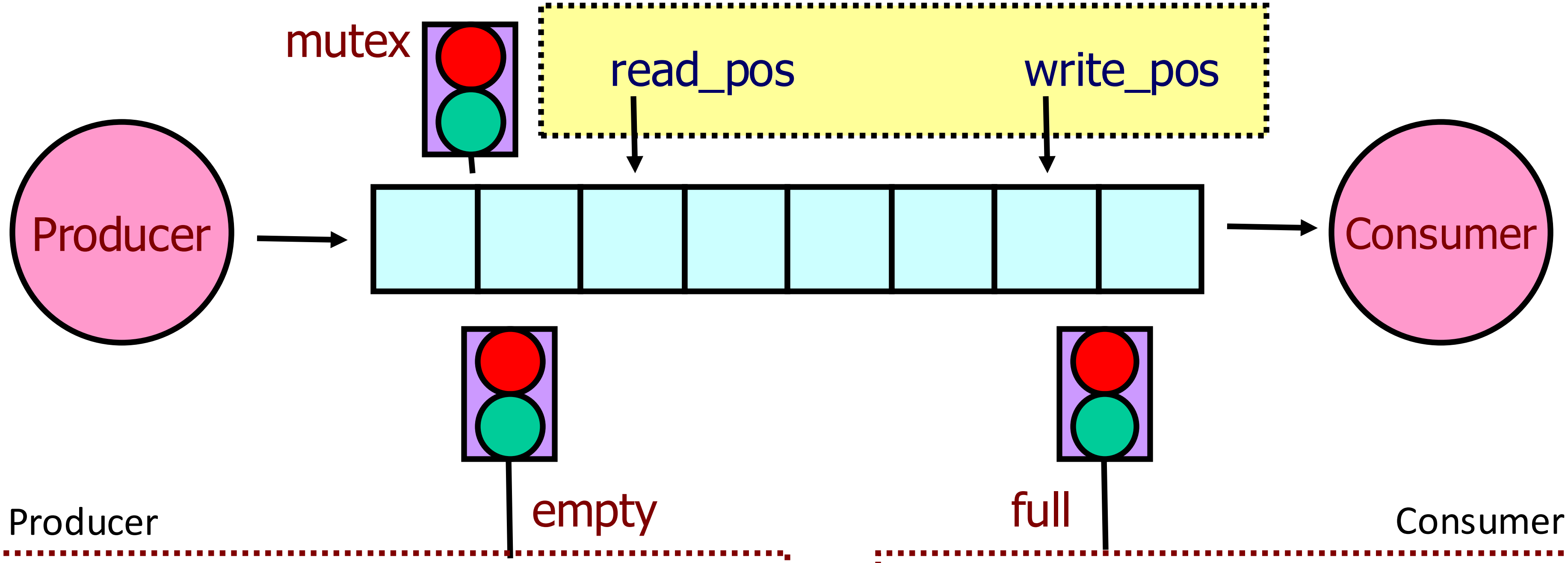
- A producer puts elements on a finite buffer. If the buffer is full, it blocks until there's space.
- The consumer retrieves elements. If the buffer is empty, it blocks until something comes along.



- We will need three semaphores
  - One to count the empty slots
  - One to count the full slots
  - One to provide for mutual exclusion to the shared buffer

Example of semaphores usage

Producer/Consumer problem review [2]



Producer

```
put_element(e) {
  sem_wait(empty);
  sem_wait(mutex);
  buf[write_pos] = e;
  write_pos = (write_pos+1) % N;
  sem_post(mutex);
  sem_post(full);
}
```

Consumer

```
get_element() {
  sem_wait(full);
  sem_wait(mutex);
  e = buf[read_pos];
  read_pos = (read_pos+1) % N;
  sem_post(mutex);
  sem_post(empty);
  return e;
}
```

Example of Producer/Consumer

Common code to SysV and POSIX semaphores

```
void producer() {
  for (int i=TOTAL_VALUES; i>0; i--) {
    printf("[PRODUCER] Writing %d\n", i);
    put_element(i);
}}

void consumer() {
  for (int i=0; i<TOTAL_VALUES; i++) {
    int e = get_element();
    printf("[CONSUMER] Retrieved %d\n", e);
    sleep(1);
  }
  terminate();
}

int main(int argc, char* argv[]) {
  init();

  if (fork() == 0) {
    producer();
    exit(0);
  }
  else {
    consumer();
    exit(0);
  }
}
```

Example of Producer/Consumer (2)
Using SysV semaphores

- `put_element()` and `get_element()` with SysV

```
void put_element(int e) {
  sem_wait(sem, EMPTY);
  sem_wait(sem, MUTEX);

  buf[*write_pos] = e;
  *write_pos = (*write_pos+1) % N;

  sem_post(sem, MUTEX);
  sem_post(sem, FULL);
}

int get_element() {
  sem_wait(sem, FULL);
  sem_wait(sem, MUTEX);

  int e = buf[*read_pos];
  *read_pos = (*read_pos+1) % N;

  sem_post(sem, MUTEX);
  sem_post(sem, EMPTY);

  return e;
}
```

Example of Producer/Consumer (3)
Using POSIX semaphores

- `put_element()` and `get_element()` with POSIX

```
void put_element(int e) {
  sem_wait(empty);
  sem_wait(mutex);

  buf[*write_pos] = e;
  *write_pos = (*write_pos+1) % N;

  sem_post(mutex);
  sem_post(full);
}

int get_element() {
  sem_wait(full);
  sem_wait(mutex);

  int e = buf[*read_pos];
  *read_pos = (*read_pos+1) % N;

  sem_post(mutex);
  sem_post(empty);

  return e;
}
```

Example of Producer/Consumer (4)
Using SysV semaphores

- `init()` and `terminate()` with SysV semaphores

```
typedef struct {
     int buf[N], write_pos, read_pos;
} mem_struct;

int shmid, sem, *write_pos, *read_pos, *buf;
mem_struct *mem;

void init() {
  shmid = shmget(IPC_PRIVATE, sizeof(mem_struct), IPC_CREAT|0700);
  mem = (mem_struct*) shmat(shmid, NULL, 0);

  sem=sem_get(3, 0);
  sem_setvalue(sem, EMPTY, N);
  sem_setvalue(sem, FULL, 0);
  sem_setvalue(sem, MUTEX, 1);

  mem->write_pos = mem->read_pos = 0;
  write_pos = &mem->write_pos;
  read_pos = &mem->read_pos;
  buf = (int*)&mem->buf;
}
```

```
void terminate() {
  sem_close(sem);
  shmctl(shmid, IPC_RMID, NULL);
}
```

Example of Producer/Consumer (5)

Using POSIX unnamed semaphores

- `init()` and `terminate()` with unnamed POSIX semaphores

```c
typedef struct {
     sem_t sem_empty, sem_full, sem_mutex;
     int buf[N], write_pos, read_pos;
} mem_struct;

int shmid, *write_pos, *read_pos, *buf;
mem_struct *mem;
sem_t *empty, *full, *mutex;

void init() {
  shmid = shmget(IPC_PRIVATE, sizeof(mem_struct), IPC_CREAT|0700);
  mem = (mem_struct*) shmat(shmid, NULL, 0);

  sem_init(&mem->sem_empty, 1, N);
  empty = &mem->sem_empty;
  sem_init(&mem->sem_full, 1, 0);
  full = &mem->sem_full;
  sem_init(&mem->sem_mutex, 1, 1);
  mutex = &mem->sem_mutex;

  mem->write_pos = mem->read_pos = 0;
  write_pos = &mem->write_pos;
  read_pos = &mem->read_pos;
  buf = (int*)&mem->buf;
}
```

```c
void terminate() {
  sem_destroy(empty);
  sem_destroy (full);
  sem_destroy (mutex);
  shmctl(shmid, IPC_RMID, NULL);
}
```

Example of Producer/Consumer (6)

Using POSIX named semaphores

- `init()` and `terminate()` with named POSIX semaphores

```
typedef struct {
     int buf[N], write_pos, read_pos;
} mem_struct;

int shmid, *write_pos, *read_pos, *buf;
mem_struct *mem;
sem_t *empty, *full, *mutex;

void init() {
  shmid = shmget(IPC_PRIVATE, sizeof(mem_struct), IPC_CREAT|0700);
  mem= (mem_struct*) shmat(shmid, NULL, 0);

  sem_unlink("EMPTY");
  empty=sem_open("EMPTY",O_CREAT|O_EXCL,0700,N);
  sem_unlink("FULL");
  full=sem_open("FULL",O_CREAT|O_EXCL,0700,0);
  sem_unlink("MUTEX");
  mutex=sem_open("MUTEX",O_CREAT|O_EXCL,0700,1);

  mem->write_pos = mem->read_pos = 0;
  write_pos = &mem->write_pos;
  read_pos = &mem->read_pos;
  buf = (int*)&mem->buf;
}
```

```
void terminate() {
  sem_close(empty);
  sem_close(full);
  sem_close(mutex);
  sem_unlink("EMPTY");
  sem_unlink("FULL");
  sem_unlink("MUTEX");
  shmctl(shmid, IPC_RMID, NULL);
}
```

Class demos included

- Remove all IPCs create by the user

  `Kill_ipcs.sh`
- *semlib* library files (this library uses SysV semaphores!)

  `semlib.h`

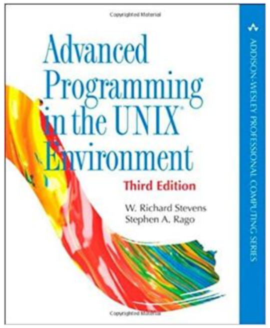
  `semlib.c`
- Example of SysV semaphores with a producer/consumer (using *semlib* library)

  `sem_test-sysv.c`
- Example of POSIX named semaphores with a producer/consumer

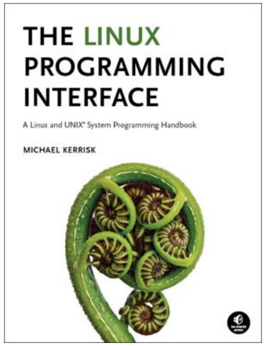
  `sem_test-posix_named.c`
- Example of POSIX unnamed semaphores with a producer/consumer

  `sem_test-posix_unnamed.c`

References



- Advanced Programming in the UNIX Environment
  2nd, 3rd Edition (2013)
  W. Richard Stevens, Stephen A. Rago
  Addison-Wesley



- The Linux Programming Interface
  2010
  Michael Kerrisk
  No Starch Press

INTRODUCTION TO ASSIGNMENT 04 – “SHARED MEMORY AND SEMAPHORES”

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who. —Rudyard Kipling