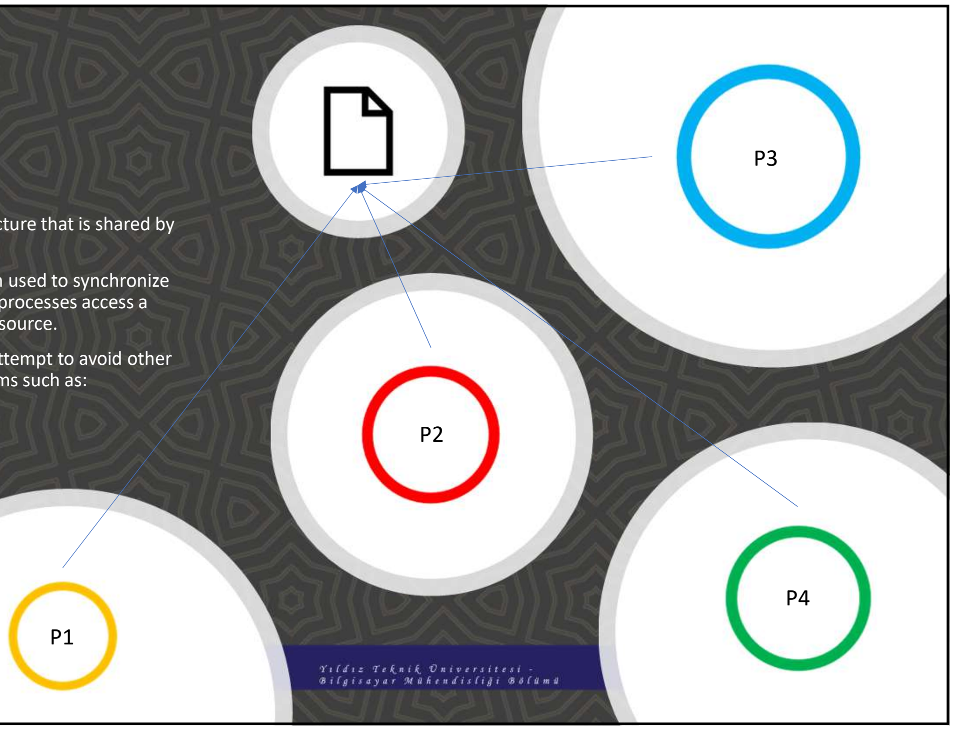
İŞLETİM SİSTEMLERİ UYGULAMA

1

Semaphores

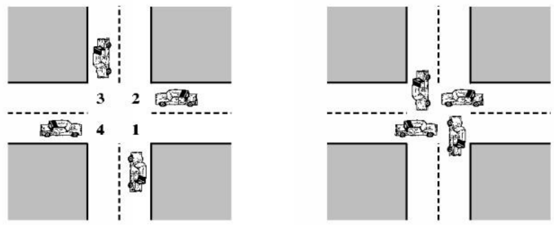
- A semaphore is a data structure that is shared by several processes.
- Semaphores are most often used to synchronize operations, when multiple processes access a common, non-shareable resource.
- By using semaphores, we attempt to avoid other multi-programming problems such as:
    - Starvation
    - Deadlock



Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

2

Starvation & Deadlock

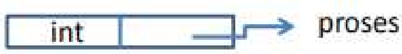


(a) Deadlock possible (b) Deadlock

3

POSIX Semaphores

- POSIX semaphores allow processes and threads to synchronize their actions.
- A semaphore is an integer whose value is **never allowed** to fall below **zero**.
- POSIX semaphores come in **two forms**:
  - named semaphores
  - unnamed semaphores.



4

Named Semaphores

- A named semaphore is identified by a name of the form /somename; that is, a null-terminated string
- Two processes can operate on the same named semaphore by passing **the same name** to sem_open().
- Named semaphore functions
  - sem_open()
  - sem_post()
  - sem_wait(), sem_timedwait(), sem_trywait()
  - sem_close()
  - sem_unlink()

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

5

Unnamed Semaphores

- An unnamed semaphore does not have a name.
  - The semaphore is placed in a region of memory that is shared between multiple threads or processes.
- A thread-shared semaphore
  - a global variable.
- A process-shared semaphore
  - must be placed in a shared memory region
    - POSIX or System V shared memory segment

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

6

Unnamed Semaphores

- Unnamed semaphore functions
  - **sem_init()**
  - sem_post()
  - sem_wait(), sem_timedwait(), sem_trywait()
  - **sem_destroy()**

7

A simple semaphore example

```c
//create & initialize semaphore
mutex = sem_open(SEM_NAME,O_CREAT,0644,1);
if(mutex == SEM_FAILED)  {
    perror("unable to create semaphore");
    sem_unlink(SEM_NAME);
    exit(-1);
}

while(i<10)  {

      sem_wait(mutex);
      t = time(&t);
      printf("Process A enters the critical section at %d \n",t);
      t = time(&t);
      printf("Process A leaves the critical section at %d \n",t);
      sem_post(mutex);
      i++;
      sleep(3);
}
sem_close(mutex);
sem_unlink(SEM_NAME);
```

```c
//create & initialize existing semaphore
mutex = sem_open(SEM_NAME,0,0644,0);
if(mutex == SEM_FAILED)  {
    perror("reader:unable to execute semaphore");
    sem_close(mutex);
    exit(-1);
}

while(i<10)  {

      sem_wait(mutex);
      t = time(&t);
      printf("Process B enters the critical section at %d \n",t);
      t = time(&t);
      printf("Process B leaves the critical section at %d \n",t);
      sem_post(mutex);
      i++;
      sleep(2);
}

sem_close(mutex);
```

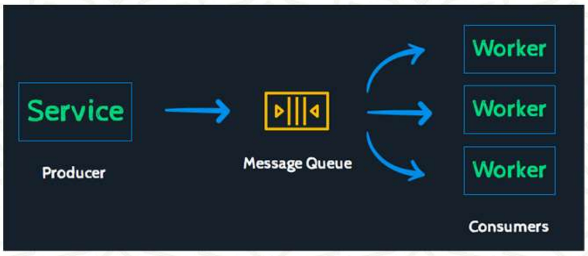
Ex_semA.c
Ex_semB.c

```
lucid@ubuntu:~$ ./PB
Process B enters the critical section at 1376420556
Process B leaves the critical section at 1376420556
Process B enters the critical section at 1376420558
Process B leaves the critical section at 1376420558
Process B enters the critical section at 1376420560
Process B leaves the critical section at 1376420560
Process B enters the critical section at 1376420562
Process B leaves the critical section at 1376420562
Process B enters the critical section at 1376420564
Process B leaves the critical section at 1376420564
Process B enters the critical section at 1376420566
Process B leaves the critical section at 1376420566
Process B enters the critical section at 1376420568
Process B leaves the critical section at 1376420568
Process B enters the critical section at 1376420570
Process B leaves the critical section at 1376420570
```

```
lucid@ubuntu: ~
File Edit View Terminal Help
lucid@ubuntu:~$ ./PA
Process A enters the critical section at 1376420554
Process A leaves the critical section at 1376420554
Process A enters the critical section at 1376420557
Process A leaves the critical section at 1376420557
Process A enters the critical section at 1376420560
Process A leaves the critical section at 1376420560
Process A enters the critical section at 1376420563
Process A leaves the critical section at 1376420563
Process A enters the critical section at 1376420566
```


8

Message Queues



- **Unlike pipes and FIFOs**, message queues support messages that have **structure**.
- Like FIFOs, message queues are persistent objects that must be initially created and eventually deleted when no longer required.
- Message queues are created with a specified maximum message size and maximum number of messages.
- Message queues are created and opened using a special version of the open system call, **mq_open**.

POSIX Message Queue Functions

- mq_open()
- mq_close()
- mq_unlink()
- mq_send()
- mq_receive()
- mq_setattr()
- mq_getattr()
- mq_notify()

mq_open(const char *name, int oflag,...)

- name
  - Must start with a slash and contain no other slashes
  - QNX puts these in the /dev/mqueue directory
- oflag
  - **O_CREAT** – to create a new message queue
  - **O_EXCL** – causes creation to fail if queue exists
  - **O_NONBLOCK** – usual interpretation
- mode – usual interpretation
- &mqattr – address of structure used during creation

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

11

mq_attr structure

- This structure, pointed to by the last argument of **mq_open**, has at least the following members:

```
struct mq_attr {
    long mq_flags;       /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue */
};
```

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

12

mq_close(mqd_t mqdes)

- This function is used to close a message queue after it has been used.
- As noted earlier, the message queue is not deleted by this call; it is persistent.
- The message queue's contents **are not altered** by **mq_close** unless a prior call(by this or another process) called **mq_unlink** (see next slide). In this respect, an open message queue is just like an open file: deletion is deferred until all open instances are closed.

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

13

mq_unlink(const char *name)

- This call is used to remove a message queue.
- Recall (from the previous slide) that the deletion is deferred until all processes that have the message queue open have closed it (or terminated).
- It is usually a good practice to call mq_unlink immediately after all processes that wish to communicate using the message queue have opened it. In this way, as soon as the last process terminates (closing the message queue), the queue itself is deleted.

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

14

Message Queue Persistence - I

- As noted, a message queue is persistent.
- Unlike a FIFO, however, the contents of a message queue are also persistent.
- It is not necessary for a reader and a writer to have the message queue open at the same time. A writer can open (or create) a queue and write messages to it, then close it and terminate.
- Later a reader can open the queue and read the messages.

mq_send(mqd_t mqdes, const char *msq_ptr, size_t msglen, unsigned msg_prio)

- **mqdes**
  - the descriptor required by mq_open
- **msg_ptr**
  - pointer to a char array containing the message
- **msglen**
  - number of bytes in the message; this must be no larger than the maximum message size for the queue
- **prio**
  - the message priority (0..MQ_PRIO_MAX); messages with larger (higher) priority leap ahead of messages with lower (smaller) priority

mq_receive(mqd_t mqdes, char *msq_ptr, size_t msglen, unsigned *msg_prio)

- **mqdes**
  - the descriptor returned by mq_open
- **msg_ptr**
  - pointer to a char array to receive the message
- **msglen**
  - number of bytes in the msg buffer; this should normally be equal to the maximum message size specified when the message queue was created
- **msg_prio**
  - pointer to a variable that will receive the message's priority
- The call returns the **size of the message, or -1**

A simple Message Queue Example
Sender

```c
/* forcing specification of "-i" argument */
if (msgprio == 0) {
    printf("Usage: %s [-q] -p msg_prio\n", argv[0]);
    exit(1);
}

/* opening the queue        --  mq_open() */
if (create_queue) {
    msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRWXU | S_IRWXG, NULL);
} else {
    msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
}
if (msgq_id == (mqd_t)-1) {
    perror("In mq_open()");
    exit(1);
}

/* producing the message */
currtime = time(NULL);
snprintf(msgcontent, MAX_MSG_LEN, "Hello from process %u (at %s).", my_pid, ctime(&currtime));

/* sending the message      --  mq_send() */
mq_send(msgq_id, msgcontent, strlen(msgcontent)+1, msgprio);

/* closing the queue        -- mq_close() */
mq_close(msgq_id);
```

- **Ex_5_mq_dropone.c**

```
lucid@ubuntu:~/Downloads$ ./Drop
Usage: ./Drop [-q] -p msg_prio
lucid@ubuntu:~/Downloads$ ./Drop -q -p 11
I (5012) will use priority 11
lucid@ubuntu:~/Downloads$ ./Drop -p 110
I (5015) will use priority 110
lucid@ubuntu:~/Downloads$ ./Drop -p 17
I (5016) will use priority 17
lucid@ubuntu:~/Downloads$
```

A simple Message Queue Example
Receiver

```
/* opening the queue        --  mq_open() */
msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
if (msgq_id == (mqd_t)-1) {
    perror("In mq_open()");
    exit(1);
}

/* getting the attributes from the queue        --  mq_getattr() */
mq_getattr(msgq_id, &msgq_attr);
printf("Queue \"%s\":\n\t- stores at most %ld messages\n\t\
       - large at most %ld bytes each\n\t- currently holds %ld messages\n",
       MSGQOBJ_NAME, msgq_attr.mq_maxmsg, msgq_attr.mq_msgsize, msgq_attr.mq_curmsgs);

/* getting a message */
msgsz = mq_receive(msgq_id, msgcontent, MAX_MSG_LEN, &sender);
if (msgsz == -1) {
    perror("In mq_receive()");
    exit(1);
}
printf("Received message (%d bytes) from %d: %s\n", msgsz, sender, msgcontent);

/* closing the queue    --  mq_close() */
mq_close(msgq_id);

mq_unlink(MSGQOBJ_NAME);
return 0;
```

```
lucid@ubuntu:~/Downloads$ ./Take
Queue "/test":
        - stores at most 10 messages
                - large at most 8192 bytes each
        - currently holds 3 messages
Received message (56 bytes) from 110: Hello from process 5015 (at Fri Aug  9 07:
34:05 2013
).
```

- **Ex_5_mq_takeone.c**

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

19

The effect of fork on a message queue

- Message queue descriptors **are not (in general) treated as file descriptors**; the unique open, close, and unlink calls should already suggest this.
- Open message queue descriptors **are not inherited by child processes** created by fork.
- Instead, a child process must explicitly open (using mq_open) the message queue itself to obtain a message queue descriptor

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

20

Detecting non-empty queues

- mq_receive on an empty queue **normally causes a process to block**, and this may not be desirable.
- Of course, **O_NONBLOCK** could be applied to the queue to prevent this behavior, but in that case the mq_receive call will return -1, and our only recourse is to try mq_receive again later.
- With the **mq_notify** call we can associate a single process with a message queue so that it (the process) will be notified when the message queue changes state from empty to non-empty

mq_notify(mqd_t mqdes, const struct sigevent *notification)

- queuefd
  - as usual, to identify the message queue
- sigev
  - a struct sigevent object that identifies the signal to be sent to the process to notify it of the queue state change.
- Once notification has been sent, **the notification mechanism is removed**. That is, to be notified of the next state change (from empty to non-empty), the notification **must be reasserted**.

Changing the process to be notified

- Only one process can be registered (at a time) to receive notification when a message is added to a previously-empty queue.
- If you wish to change the process that is to be notified, you must remove the notification from the process which is currently associated (call mq_notify with NULL for the sigev argument), and then associate the notification with a different process.

Attributes

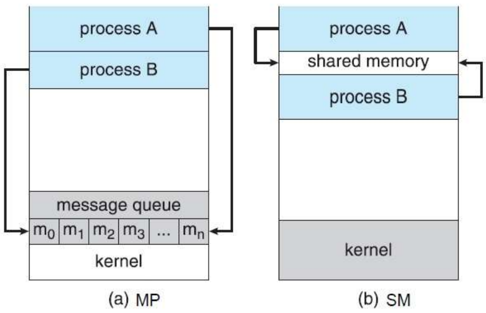
- **mq_getattr** (queuefd,&mqstat)
  - retrieves the set of attributes for a message queue to the struct mq_attr object named mqstat.
  - the mq_flags member of the attributes is not significant during mq_open, but it can be set later
- **mq_setattr** (queuefd,&mqstat,&old)
  - Set (or clear) to O_NONBLOCK flag in the mqattr structure for the identified message queue
  - Retrieve (if old is not NULL) the previously existing message queue attributes
  - Making changes to any other members of the mqattr structure is **ineffective**.

Timed send and receive

- Two additional functions, mq_timedsend and mq_timedreceive, are like mq_send and mq_receive except they have an additional argument, a pointer to a struct timespec.
- This provides the absolute time at which the send or receive will be aborted if it cannot be completed (because the queue is full or empty, respectively).

Shared Memory

- Sharing memory in POSIX (and many other systems) requires
  - creating a persistent "object" associated with the shared memory, and
  - allowing processes to connect to the object.
- creating or connecting to the persistent object is done in a manner similar to that for a file, but uses the shm_open system call.



Shared Memory Functions

- shm_open()
- mmap()
- munmap()
- ftruncate()
- shm_unlink()

shm_open (name, oflag, mode)

- name is a string identifying an existing shared memory object or a new one (to be created). It should begin with '/', and contain only one slash. In QNX 6, these objects will appear in a special directory.
- mode is the protection mode (e.g. 0644).
- shm_open returns a file descriptor, or –1 in case of error

shm_open (name, oflag, mode)

- oflag is similar to the flags for files:
  - O_RDONLY – read only
  - O_RDWR – read/write
  - O_CREAT – create a new object if necessary
  - O_EXCL – fail if O_CREAT and object exists
  - O_TRUNC – truncate to zero length if opened R/W

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

29

ftruncate(int fd, off_t len)

- This function (inappropriately named) causes the file referenced by fd to have the size specified by len.
- If the file was previously longer than len bytes, the excess is discarded.
- If the file was previously shorter than len bytes, it is extended by bytes containing zero.

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

30

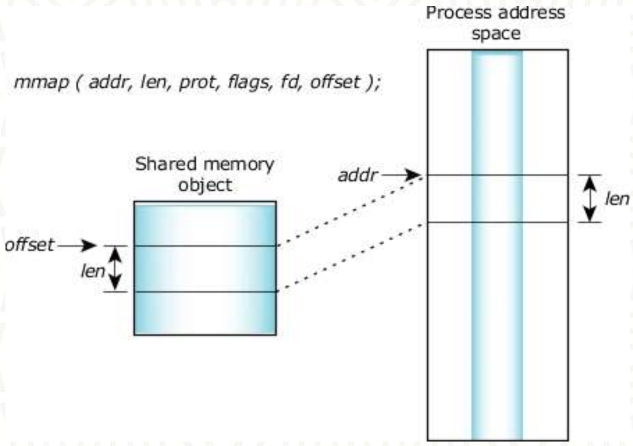
mmap (void *addr, size_t len, int prot, int flags, int fd, off_t off);

- mmap is used to map a region of the shared memory object (fd) to the process' address space.
- The mapped region has the given len starting at the specified offset off.
- Normally addr is 0, and allows the OS to decide where to map the region. This can be explicitly specified, if necessary.
- mmap returns the mapped address, or –1 on error.(more on next slide)

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

31

Mmap()



Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

32

mmap, continued

- prot – selected from the available protection settings:
  - PROT_EXEC ; This value is allowed, but is equivalent to PROT_READ.
  - PROT_NOCACHE;
  - PROT_NONE; No data access is allowed.
  - PROT_READ; Read access is allowed.
  - PROT_WRITE; Write access is allowed. Note that this value assumes PROT_READ also.
- flags – one or more of the following:
  - MAP_FIXED – interpret addr parameter exactly
  - MAX_PRIVATE – don't share changes to object
  - MAP_SHARED – share changes to object

munmap (void *addr, size_t len)

- This function removes mappings from the specified address range.
- This is not a frequently-used function, as most processes will map a fixed-sized region and use shm_unlink at the end of execution to destroy the shared memory object (which effectively removes the mappings).

shm_unlink (char *name);

- This function, much like a regular unlink system call, removes a reference to the shared memory object.
- If the are other outstanding links to the object, the object itself continues to exist.
- If the current link is the last link, then the object is deleted as a result of this call.

35

A Simple Shared Memory Example

Sender

```c
/* creating the shared memory object    --  shm_open()  */
shmfd = shm_open(SHMOBJ_PATH, O_CREAT | O_EXCL | O_RDWR, S_IRWXU | S_IRWXG);
if (shmfd < 0) {
    perror("In shm_open()");
    exit(1);
}
fprintf(stderr, "Created shared memory object %s\n", SHMOBJ_PATH);

/* adjusting mapped file size (make room for the whole segment to map)      --  ftruncate() */
ftruncate(shmfd, shared_seg_size);

/* requesting the shared segment    --  mmap() */
shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
if (shared_msg == NULL) {
    perror("In mmap()");
    exit(1);
}
fprintf(stderr, "Shared memory segment allocated correctly (%d bytes).\n", shared_seg_size);

srandom(time(NULL));
/* producing a message on the shared segment */
shared_msg->type = random() % TYPES;
snprintf(shared_msg->content, MAX_MSG_LENGTH, "My message, type %d, num %ld", shared_msg->type, random());
```

- **Ex_6_shm_server.c**

```
lucid@ubuntu:~/Downloads$ ./SHMServer
Created shared memory object /foo1423
Shared memory segment allocated correctly (56 bytes).
lucid@ubuntu:~/Downloads$ 
```


36

A Simple Shared Memory Example
Receiver

```c
/* creating the shared memory object    --  shm_open()  */
shmfd = shm_open(SHMOBJ_PATH, O_RDWR, S_IRWXU | S_IRWXG);
if (shmfd < 0) {
    perror("In shm_open()");
    exit(1);
}
printf("Created shared memory object %s\n", SHMOBJ_PATH);

/* requesting the shared segment    --  mmap() */
shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
if (shared_msg == NULL) {
    perror("In mmap()");
    exit(1);
}
printf("Shared memory segment allocated correctly (%d bytes).\n", shared_seg_size);


printf("Message type is %d, content is: %s\n", shared_msg->type, shared_msg->content);
```

- **Ex_6_shm_client.c**

```
lucid@ubuntu:~/Downloads$ ./SHMClient
Created shared memory object /foo1423
Shared memory segment allocated correctly (56 bytes).
Message type is 6, content is: My message, type 6, num 1256344664
lucid@ubuntu:~/Downloads$
```

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

37

References

- http://cs.unomaha.edu/~stanw/091/csci8530/
- http://mij.oltrelinux.com/devel/unixprg/
- Man pages
- man mq_overview
- man mq_open, mq_close etc. etc. etc.
- http://forum.soft32.com/linux2/Utilities-listing-removing-POSIX-IPC-objects-ftopict15659.html

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

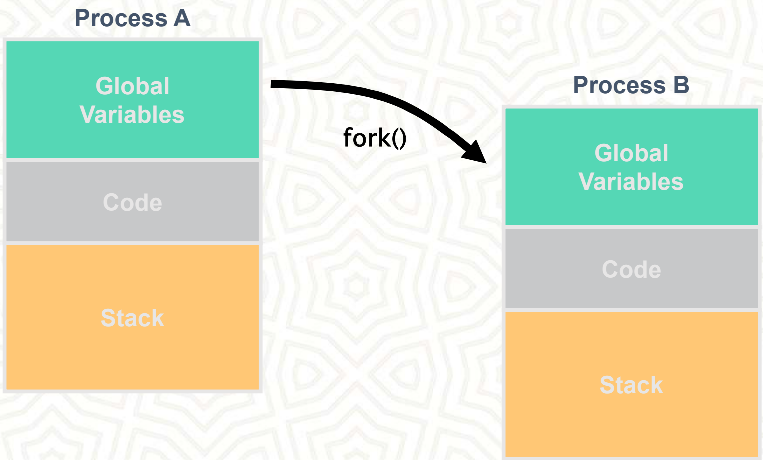
38

Threads vs. Processes

- Creation of a new process using fork is *expensive* (time & memory).
- A thread (sometimes called a *lightweight process*) does not require lots of memory or startup time.

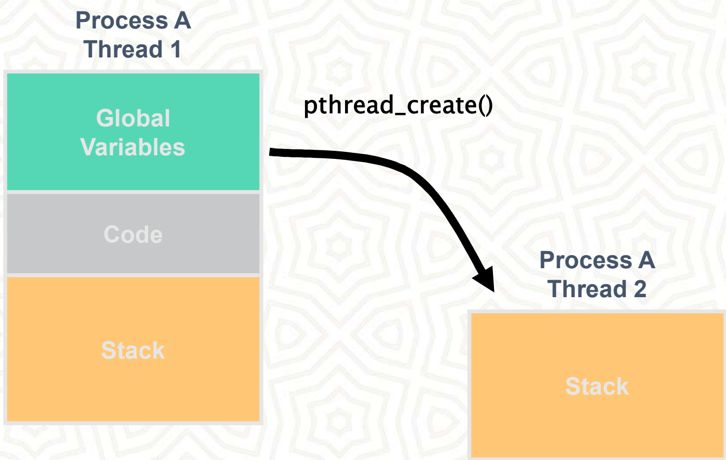
39

fork()



40

pthread_create()



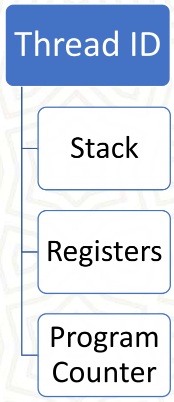
Multiple Threads

- Each process can include many threads.

- All threads of a process share:
  - memory (program code and global data)
  - open file/socket descriptors
  - signal handlers and signal dispositions
  - working environment (current directory, user ID, etc.)

Thread-specific Resources

- Each thread has its own
  - Thread ID
  - Stack, Registers, Program Counter
- Threads within the same process can communicate using shared memory.
  - ***Must be done carefully***



Posix Threads

- We will focus on Posix Threads - most widely supported threads programming API.
- you need to link with "**-lpthread**"

Thread Creation

- pthread_create(

  pthread_t *tid,

  const pthread_attr_t *attr,

  void *(***func**)(void *),

  void *arg);
- **func** is the function to be called.
  - when func() returns the thread is terminated.

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

45

`pthread_create()`

- The return value is 0 for OK.
  - positive error number on error.
- Does not set errno !!!
- Thread ID is returned in **tid**

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

46

pthread_create()

Creates a new thread executing a start routine (callback). function.

```
#include <pthread.h>

int pthread_create(
 pthread_t *thread,
 const pthread_attr_t *attr,
 void *(*start_routine)(void*),
 void *arg
 );
```

On success, the ID of the created thread will be stored here.

What does this mean?

Return type of the function		Name of function pointer		Type of parameter to the function	
void	* (*	start_routine	) (	void	*)

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

47

Thread IDs

- Each thread has a unique ID, a thread can find out it's ID by calling **pthread_self()**.
- Thread IDs are of type pthread_t which is usually an unsigned int. When debugging, it's often useful to do something like this:
  - **printf("Thread %u:\n",pthread_self());**

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

48

Thread Arguments

- When **func()** is called the value **arg** specified in the call to **pthread_create()** is passed as a parameter.
- **func** can have **only 1** parameter, and it can't be larger than the size of a **void ***.

49

Thread Arguments (cont.)

- Complex parameters can be passed by creating a structure and passing the address of the structure.
- The structure can't be a local variable (of the function calling **pthread_create**)!!
  - threads have different stacks!

50

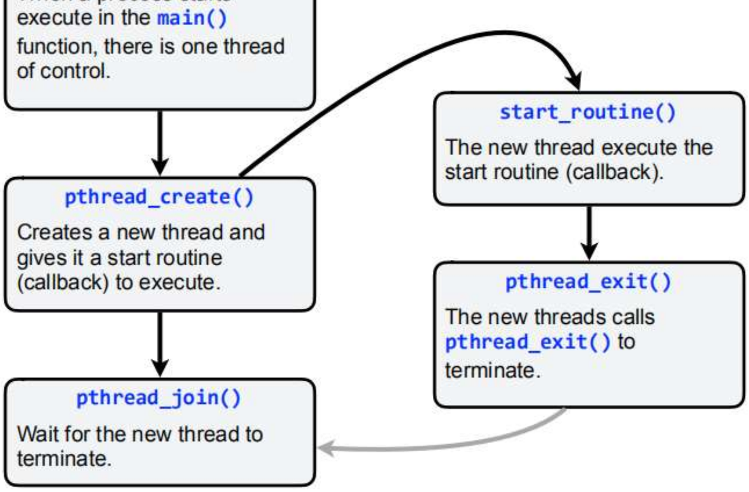
Thread args example

- struct { int x,y } 2ints;

```
void *blah( void *arg) {
        struct 2ints *foo = (struct 2ints *) arg;
        printf("%u sum of %d and %d is %d\n",
                    pthread_self(),
                   foo->x, foo->y,
                   foo->x+foo->y);
        return(NULL);
}
```


51

Pthread workflow



Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

52

Thread Lifespan

- Once a thread is created, it starts executing the function **func()** specified in the call to **pthread_create()**.
- If **func()** returns, the thread is terminated.
- A thread can also be terminated by calling **pthread_exit().**
- If **main()** returns or any thread calls exit()all threads are terminated.

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

53

pthreads_create_exit_null_join.c

This program creates four threads and wait for all of them to terminate.

```
$ ./bin/pthreads_create_exit_null_join
main() - before creaing new threads
  thread 0 - hello
  thread 1 - hello
  thread 2 - hello
  thread 3 - hello
main() - thread 0 terminated
main() - thread 1 terminated
main() - thread 2 terminated
main() - thread 3 terminated
main() - all new threads terminated
$
```

Ex_1_pthread1.c

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

54

```
void* hello(void* arg) {
  int i = *(int*) arg;
  printf("  thread %d - hello\n", i);
  pthread_exit(NULL);
}
```

This is the start routine each of the threads will execute.

Every start routine must take `void*` as argument and return `void*`.

When creating a new thread we will use a pointer to an integer as argument, pointing to an integer with the thread number.

Here we first cast from `void*` to `int*` and then dereference the pointer to get the integer value.

Terminate the thread by calling `pthread_exit(NULL)`. Here `NULL` means we don't specify a termination status.


```
/* An array of thread identifiers, needed by
   pthread_join() later. */
pthread_t tid[NUM_OF_THREADS];

/* An array to hold argument data to the hello()
   start routine for each thread. */
int arg[NUM_OF_THREADS];

/* Attributes (stack size, scheduling information
   etc) for the new threads. */
pthread_attr_t attr;

/* Get default attributes for the threads. */
pthread_attr_init(&attr);
```

Declaration of arrays used to store thread IDs and arguments for each threads start routine, the `hello()` function.

Use default attributes when creating new threads.

```c
/* Create new threads, each executing the
   hello() function. */
for (int i = 0; i < NUM_OF_THREADS; i++) {
  arg[i] = i;
  pthread_create(&tid[i], &attr, hello, &arg[i]);
}
```

1 2 3 4

1) Pass in a pointer to **tid_t**. On success **tid[i]** will hold the thread ID of thread number **i**.
2) Pass a pointer to the default attributes.
3) The start routine (a function pointer).
4) A pointer to the argument for the start routine for thread number **i**.

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

57

```c
/* Wait for all threads to terminate. */
for (int i = 0; i < NUM_OF_THREADS; i++){
  if (pthread_join(tid[i], NULL) != 0) {
    perror("pthread_join");
    exit(EXIT_FAILURE);
  }
  printf("main() - thread %d terminated\n", i);
}

printf("main() - all new threads terminated\n");
```

1 2

Ex_2_pthread2.c

1) Wait for thread with thread ID **tid[i]** to terminate.
2) Pass **NULL** here means we don't care about the exit status of the terminated thread.

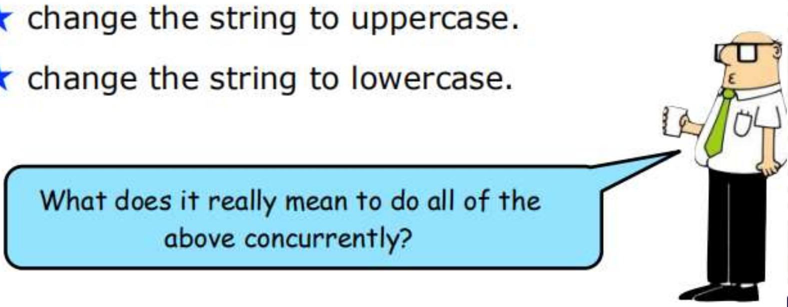
Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

58

pthreads_unsynchronized_concurrency.c

Given a string, write a program using Pthreads to concurrently:

- ★ calculate the length of the string.
- ★ calculate the number of spaces in the string.
- ★ change the string to uppercase.
- ★ change the string to lowercase.



59

Header files and global data

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h> // sleep()

#define NUM_OF_THREADS 4

/* A global string for the threads to work on. */
char STRING[] = "The string shared among the threads.";

/* Global storage for results. */
int LENGTH;
int NUM_OF_SPACES;
```

Start routines (1)

```
void* length(void *arg) {
  char *ptr = (char*) arg;
  int i = 0;
  while (ptr[i]) i++;
  LENGTH = i;
}

void* num_of_spaces(void *arg) {
  char *ptr = (char*) arg;
  int i = 0;
  int n = 0;

  while (ptr[i]) {
    if (ptr[i] == ' ') n++;
    i++;
  }
  NUM_OF_SPACES = n;
}
```

The implementation details of these functions are not important for the purpose of this exercise.

But, note that to for Pthreads to be able to use these functions as start routines for the threads, they must all be declared `void*` and take a single argument of type `void*`.

60

main() - step 1

```c
int main(int argc, char *argv[]) {
  /* An array of thread identifiers, needed by pthread_join() later... */
  pthread_t tid[NUM_OF_THREADS];

  /* Attributes (stack size, sche
  pthread_attr_t attr;

  /* Get default attributes for the thre   . */
  pthread_attr_init(&attr);

  pthread_create(&tid[i], &attr, length, STRING);
```

We could simply call **pthread_create()** four times using the four different string functions:

- length()
- num_of_spaces()
- to_upppercase()
- to_lowercase()

, for example like this.

But, it is more practical (and fun) to collect pointers to all the functions in an array.

main() - step 2

```c
int main(int argc, char *argv[]) {
  /* An array of thread identifiers, needed by pthread_join() later... */
  pthread_t tid[NUM_OF_THREADS];

  /* An array of pointers to the callback functions. */
  void* (*callback[NUM_OF_THREADS]) (void* arg) =
    {length,
     to_uppercase,
     to_lowercase,
     num_of_spaces};

  /* Attributes (stack size, scheduling information) for the threads. */
  pthread_attr_t attr;

  /* Get default attributes for the threads. */
  pthread_attr_init(&attr);

  /* Create one thread running each of the callbacks. */
  for (int i = 0; i < NUM_OF_THREADS; i++) {
    pthread_create(&tid[i], &attr, *callback[i], STRING);
  }

  /* Wait for all threads to terminate. */
  for (int i = 0; i < NUM_OF_THREADS; i++){
    pthread_join(tid[i], NULL);
  }

  /* Print results. */
  printf("       lenght(\"%s\") = %d\n", STRING, LENGTH);
  printf("num_of_spaces(\"%s\") = %d\n", STRING, NUM_OF_SPACES);
}
```

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

61

Test runs

Terminal — a.out — 74×17

```
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: gcc -std=c99 pthreads.c
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
         lenght("tHE STRING SHared among the threads.") = 36
num_of_spaces("tHE STRING SHared among the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
         lenght("THE STRING SHARED AMONG THE THREADS.") = 36
num_of_spaces("THE STRING SHARED AMONG THE THREADS.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
         lenght("THE STRING SHARED among the threads.") = 36
num_of_spaces("THE STRING SHARED among the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
         lenght("THE STRING SHARED AMONG THE THREADS.") = 36
num_of_spaces("THE STRING SHARED AMONG THE THREADS.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
         lenght("tHe string shared amOng the threads.") = 36
num_of_spaces("tHe string shared amOng the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: 
```

Because the threads execute and operate on the same data concurrently, the result of **to_uppercase()** and **to_lowercase()** will be unpredictable due to data races.

Ex_3_pthread3.c

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

62

Detached vs. Joinable

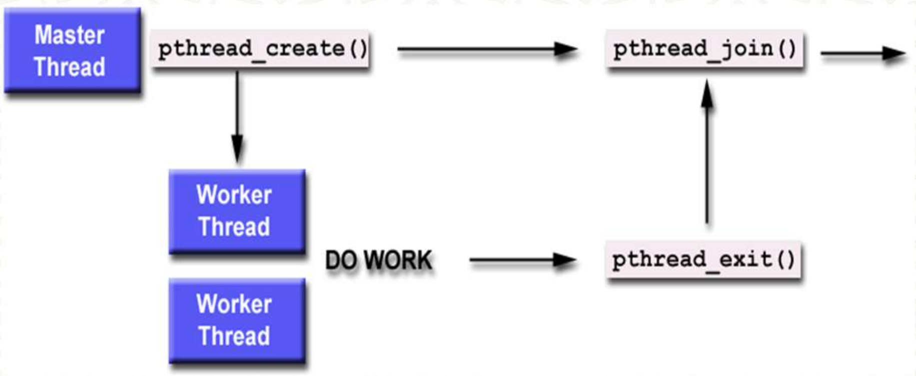
Ex_4_pthread4.c

- Each thread can be either **joinable** or **detached**.
- **Joinable:** on thread termination the thread ID and exit status are saved by the OS.
- **Detached:** on termination all thread resources are released by the OS. A detached thread cannot be joined.

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

63

Detached vs. Joinable (Contd.)



Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

64

Howto detach

Ex_5_pthread5.c

```
#include <pthread.h>

pthread_t      tid;  // thread ID
pthread_attr_t attr; // thread attribute

// set thread detachstate attribute to DETACHED
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

// create the thread
pthread_create(&tid, &attr, start_routine, arg);
...
```

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

65

Shared Global Variables

- Possible problems
  - Global variables
- Avoiding problems
- Synchronization Methods
  - Mutexes
  - Condition variables

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

66

Possible problems

- Sharing global variables is dangerous - two threads may attempt to modify the same variable at the same time.
- Just because you don't see a problem when running your code doesn't mean it can't and won't happen!!!!

67

Avoiding problems

pthreads includes support for **Mutual Exclusion** primitives that can be used to protect against this problem.

The general idea is to **lock** something before accessing global variables and to unlock as soon as you are done.

Shared socket descriptors should be treated as **global variables**!!!

68

Mutexes

Ex_6_pthread6.c

A global variable of type pthread_mutex_t is required:

pthread_mutex_t counter_mtx = **PTHREAD_MUTEX_INITIALIZER;**

Initialization to PTHREAD_MUTEX_INITIALIZER is required for a static variable!

69

Lock & Unlock

To lock use:

- **pthread_mutex_lock(pthread_mutex_t &);**

To unlock use:

- **pthread_mutex_unlock(pthread_mutex_t &);**

Both functions are blocking!

70

Condition Variables

pthreads support **condition variables**, which allow one thread to wait (sleep) for an event generated by any other thread.

This allows us to avoid the **busy waiting** problem.

pthread_cond_t foo =
PTHREAD_COND_INITIALIZER;

Yıldız Teknik Üniversitesi - Bilgisayar

71

Condition Variables (cont.)

- A condition variable is always used with mutex.
- **pthread_cond_wait(pthread_cond_t *cptr,**
  pthread_mutex_t *mptr);
- **pthread_cond_signal(pthread_cond_t *cptr);**

don't let the word signal confuse you - this has nothing to do with Unix signals

Ex_7_pthread7.c

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

72

Summary

- Threads are awesome, but dangerous. You have to pay attention to details or it's easy to end up with code that is incorrect (doesn't always work, or hangs in deadlock).
- Posix threads provides support for mutual exclusion, condition variables and thread-specific data.
- IHOP serves breakfast 24 hours a day!

References

- https://github.com/uu-os-2019/
- Getting Started With POSIX Threads by Tom Wagner & Don Towsley Department of Computer Science University of Massachusetts at Amherst July 19, 1995