Embedded Software

Processes in Linux

ENGINEERING COLLEGE OF AARHUS

Agenda

- Processes in Linux
- IPC why?
- Pipes
- Message Queues
- Shared Memory
- IPC when?

Processes in Linux

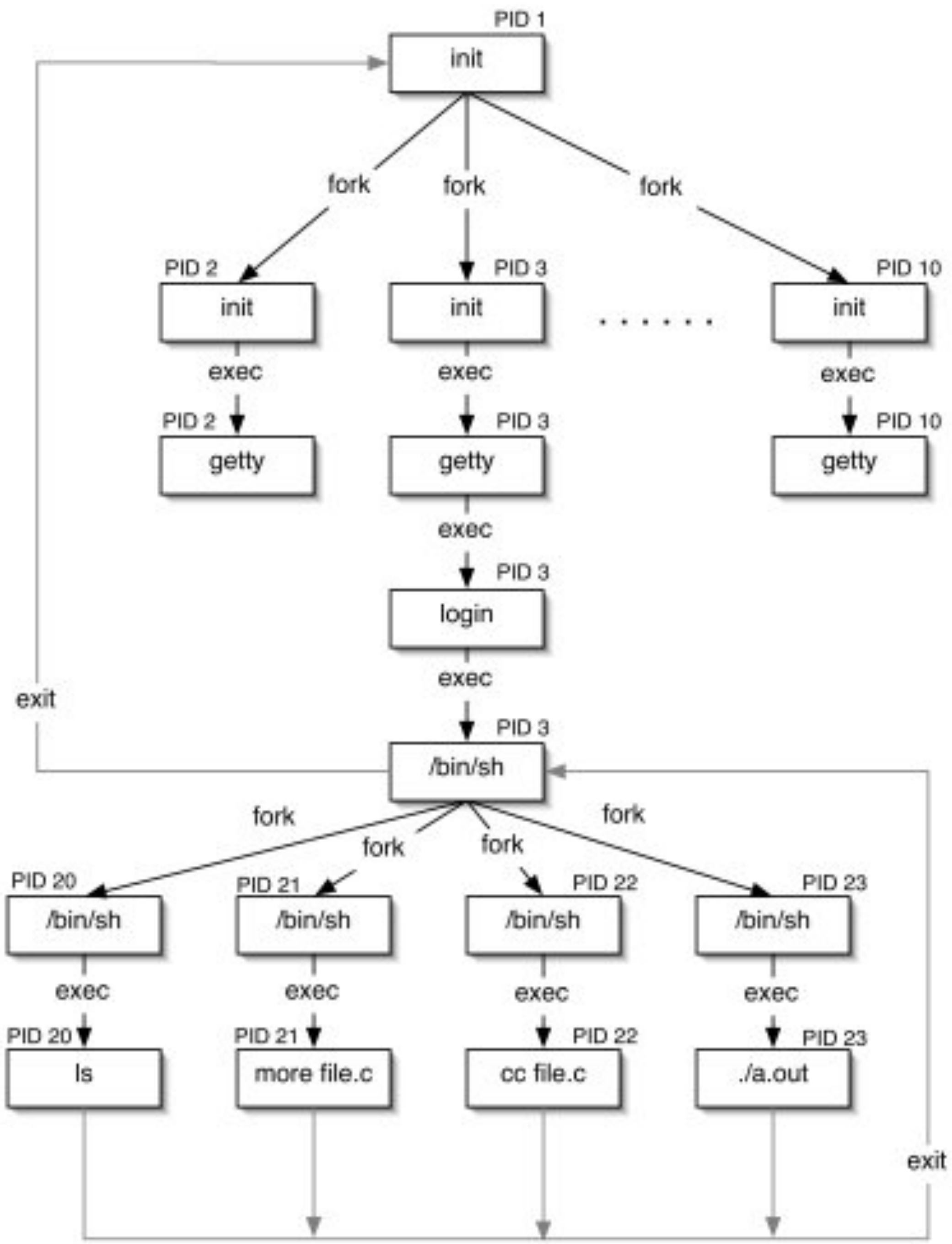
Processes in Linux

- Questions to answer:
  - How are processes created, waited for and deleted code wise
  - What happens when a process is created in Linux and of what is it composed
- Key programming concepts:
  - fork()-wait()-exit()
  - The exec()-family of functions

How are processes created, waited for and deleted code wise

Processes in Linux

- All processes are connected in a process tree



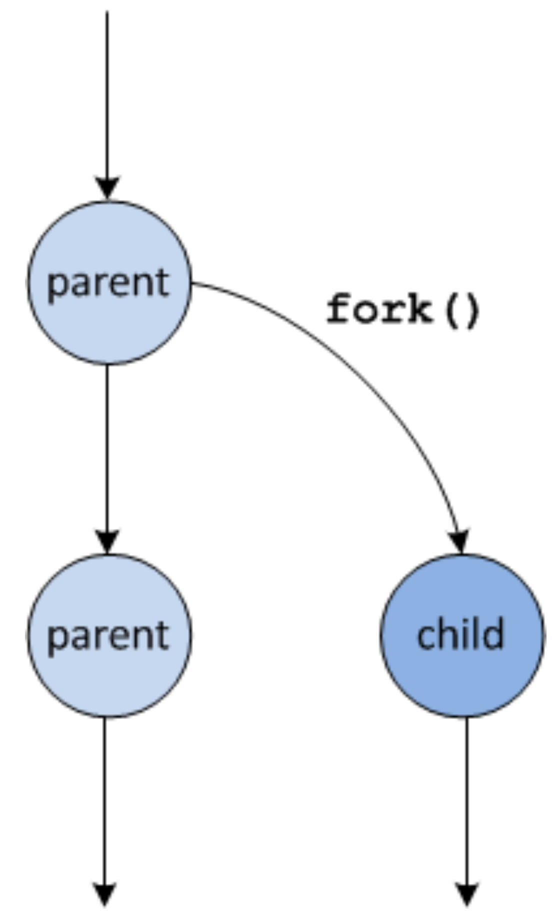
Creating new processes – fork()

- To create a new process, a program uses fork()
- fork() creates a new child process identical to the parent
- When fork() is called, a process id (pid) is returned:
  - The parent process is returned the pid of the child process
  - The child process is returned 0
- By *switching* on the returned pid, it can be determined whether the program is being executed by the parent or child process

Creating new processes – fork()

- The process creation structure:

```
void useFork()
{
   pid_t pid;
   pid = fork();
   if(pid == 0)
   {
        cout << "Hi! I'm the child process!" << endl;
        // Do child stuff
   }
   else
   {
        cout << "Hi! I'm the parent process!" << endl;
        // Do parent stuff
   }
}
```

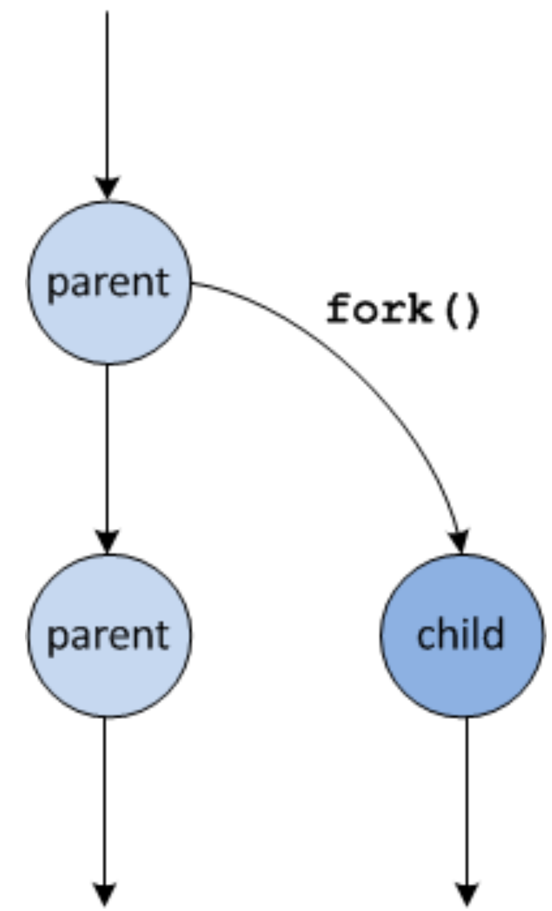


ENGINEERING COLLEGE OF AARHUS

Creating new processes – fork()

- What does this program output when executed?

```
void main()
{
   fork();
   cout << ”Hello world” << endl;
}
```



ENGINEERING COLLEGE OF AARHUS

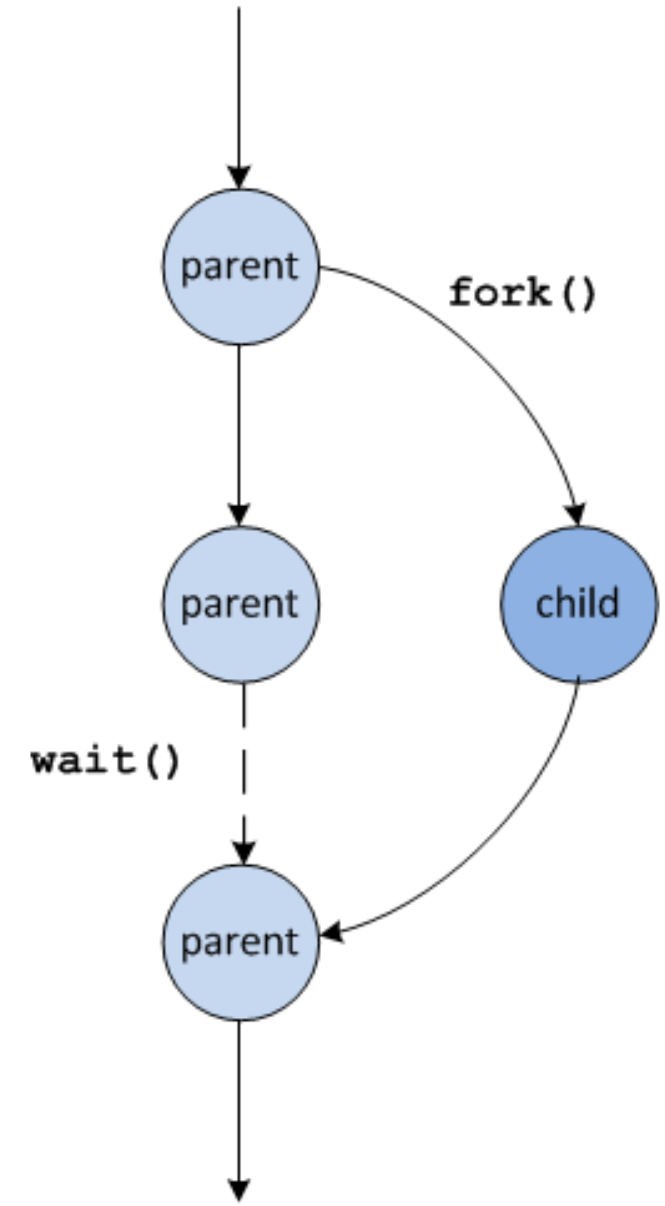
Waiting for child processes – wait() and _exit()

- Parent processes may wait for child(ren) to complete
- Parents wait for a child by calling **wait()**
- Children complete by calling **_exit(int)** (***not*** **exit(int)**)

Waiting for child processes – wait() and _exit()

- Waiting and exiting structure:

```
void useFork()
{
   pid_t pid;
   pid = fork();
   if(pid == 0)
   {
       cout << ”Hi! I’m the child process!” << endl;
       // Do child stuff
       _exit(); // Do exit process
   }
   else
   {
       cout << ”Hi! I’m the parent process!” << endl;
       // Do parent stuff
       wait(NULL);     // Wait for child
   }
}
```

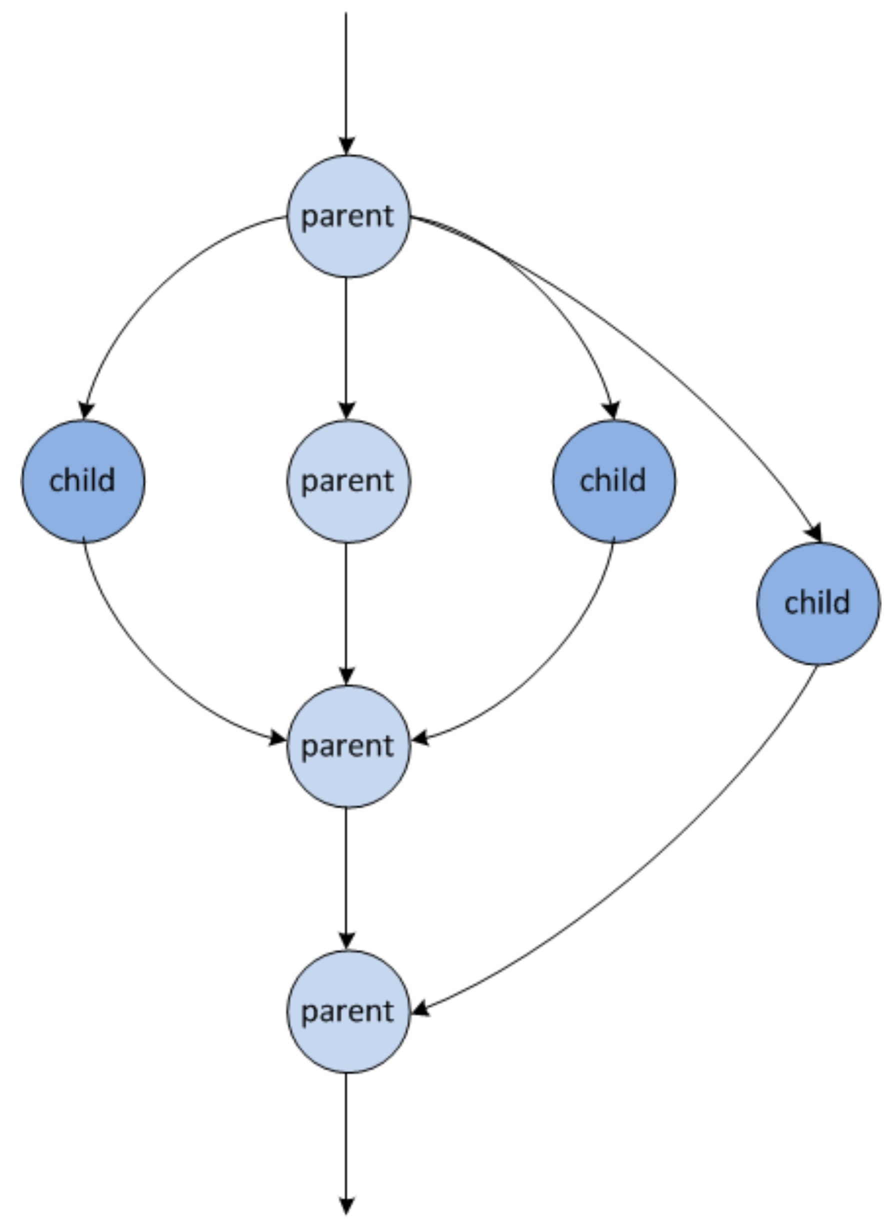


Tuesday, May 15, 12

Using fork(), wait() and _exit()

- Using **fork()**, **wait()** and **_exit()** you may create almost arbitrarily complex process relations.

- For example, how can you create this tree?



Using the exec()-family

- Fork() lets us *clone* process, but what if we want to start another process?
  - Enter the **exec()**-family of functions
  - The **exec()** system call will replace the *current process' image* with the one specified in the argument to **exec()**.
  - An example: **execp("/bin/ls", "ls")** will run the program **ls** , i.e. execute the **ls** command
  - Note: **exec()**-functions do not return unless there is an error!

Using the exec()-family

- How to use exec()-functions (example: execv()):

```
int execv(const char *path, char *const argv[]);
```

Using the exec()-family

- How to use exec()-functions (example: execv()):

```
int execv(const char *path, char *const argv[]);
```

Path to the program to execute, e.g. ”/bin/pwd”

Using the exec()-family

- How to use exec()-functions (example: execv()):

```
int execv(const char *path, char *const argv[]);
```

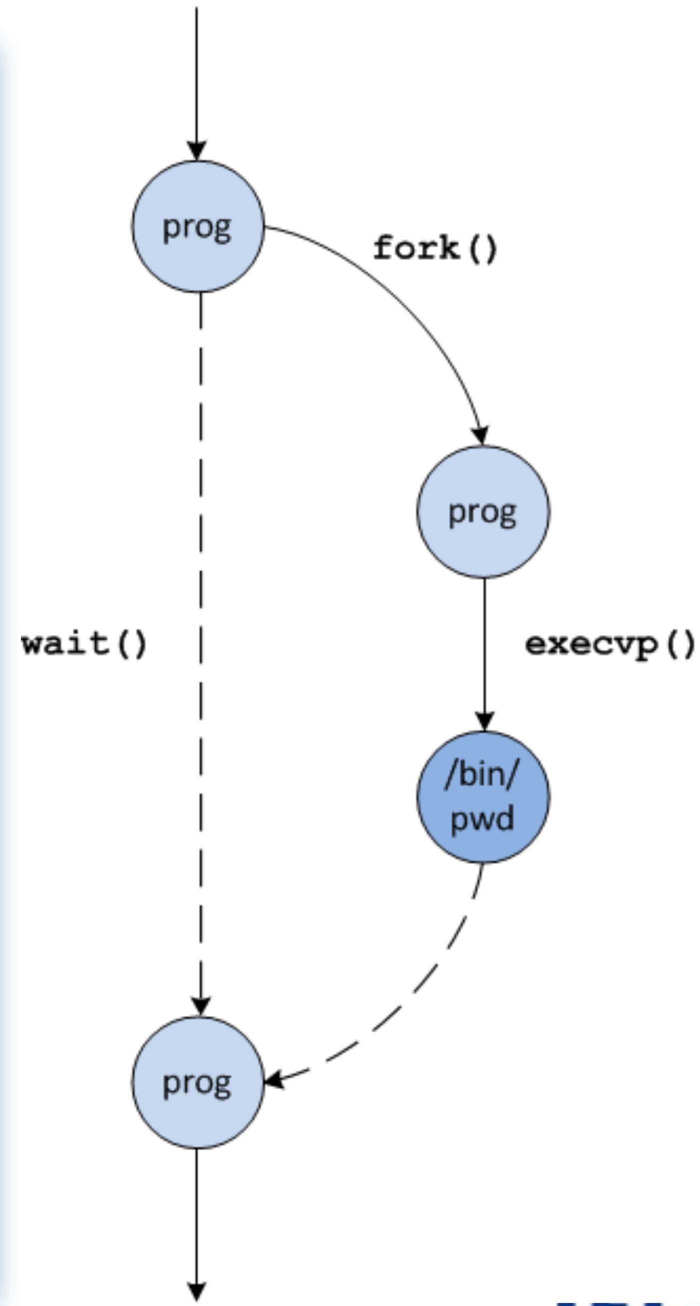
Path to the program to execute, e.g. ”/bin/pwd”

Arguments for the program to execute.
* The first must be the name of the program, e.g. ”/bin/pwd”
* The last must be NULL

Using the exec()-family

- Example: Executing the **pwd** command using **fork()** and **execv()**

```
/*
 * main.cpp
 *
 *  Created on: Jan 6, 2011
 *      Author: stud
 */
#include<sys/types.h>
#include<sys/wait.h>
#include<iostream>

using namespace std;

int main()
{
    int status;
    int pid;
    char* prog = "/bin/pwd";
    char* args[] = {"/bin/pwd", NULL};

    pid = fork();

    if(pid==0)
    {
          execv(prog, args);
          cout << "Error! " << endl;
          _exit(1);
    }
    else
    {
          wait(&status);
          return 0;
    }
}
```



Using the exec()-family

- The complete family: (#include <unistd.h>)

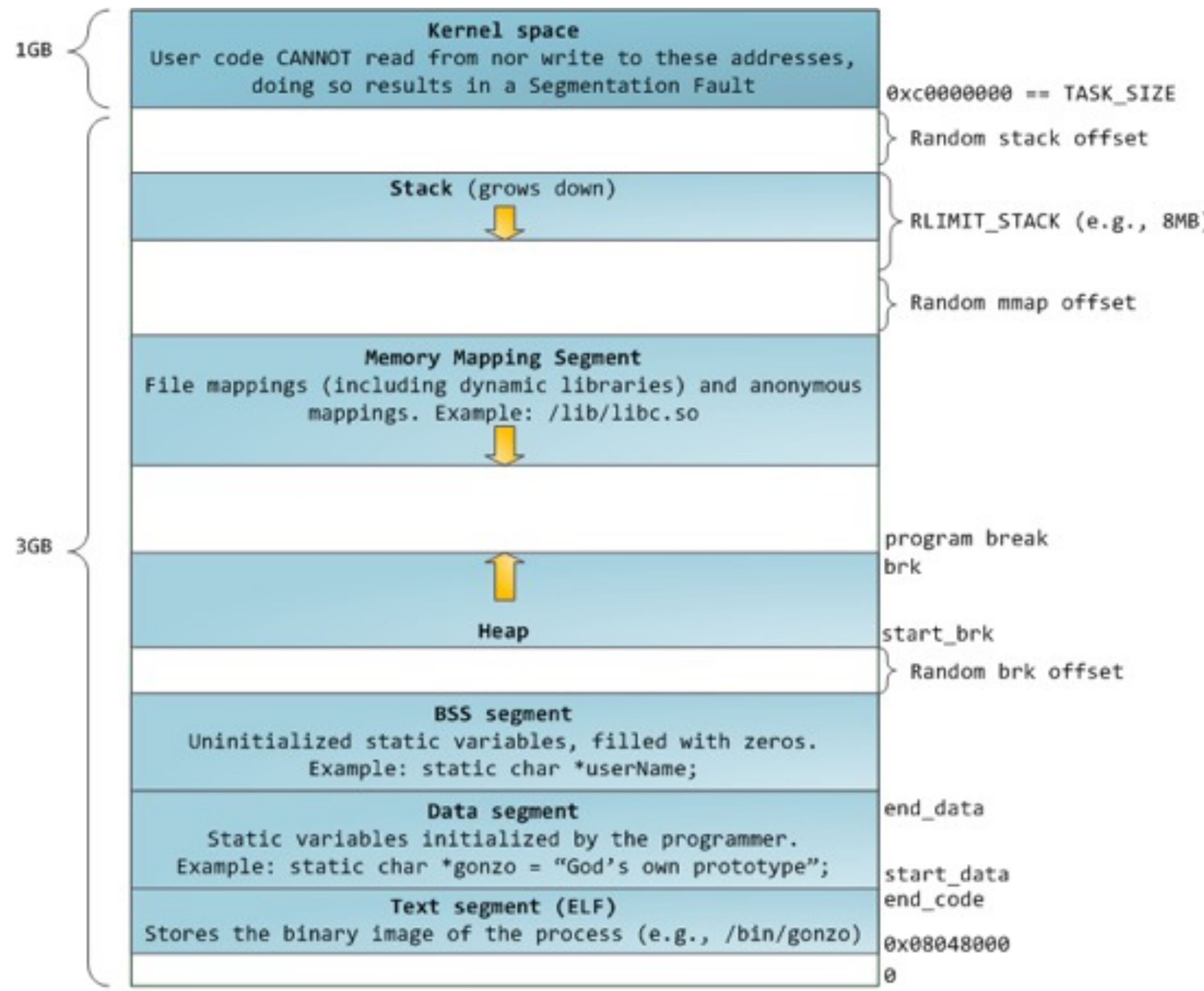
```c
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *file, char *const argv[], char * const envp[]);
```

What happens when a process is created in Linux and of what is it composed

Process - What is it?

- Process - A program being executed in Linux

Process Memory Layout



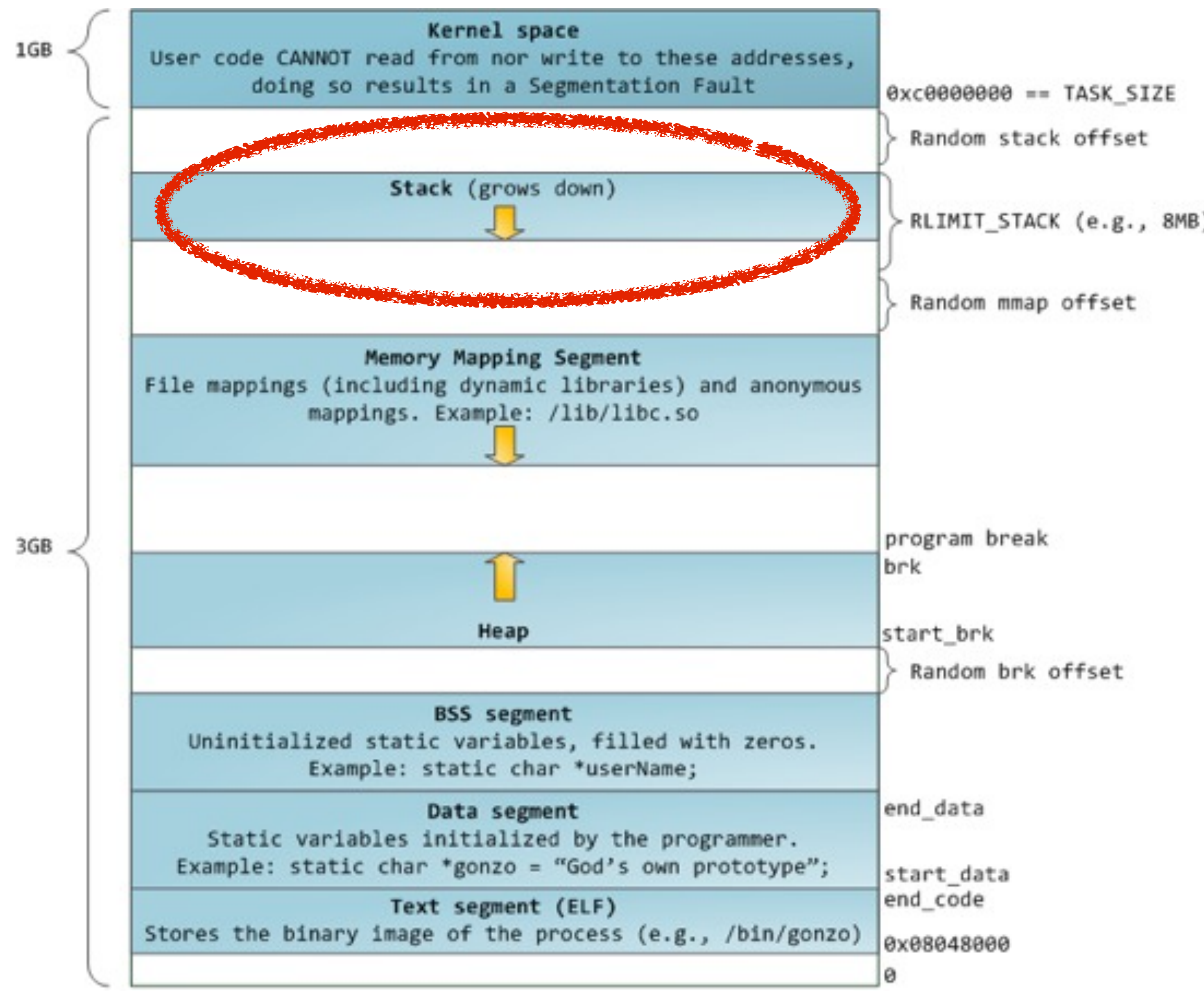
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

IHA | ENGINEERING COLLEGE OF AARHUS

Process - What is it?

- Process - A program being executed in Linux
  - Stack
    - Local variables
    - Function return values
    - LIFO

Process Memory Layout

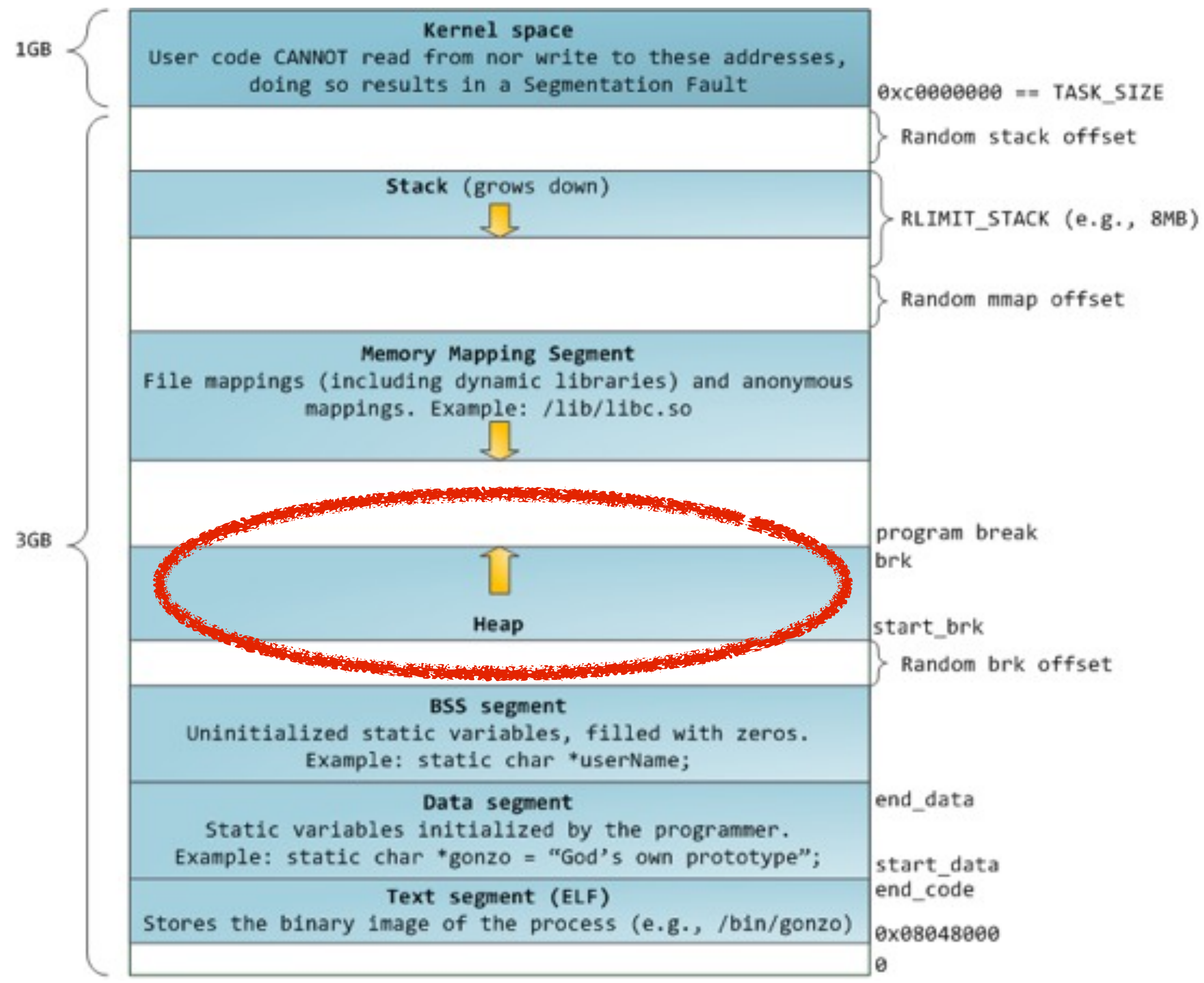


http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Process - What is it?

- Process - A program being executed in Linux
  - Stack
    - Local variables
    - Function return values
    - LIFO
  - Heap
    - "Free-store"
    - Dynamically allocated memory

Process Memory Layout

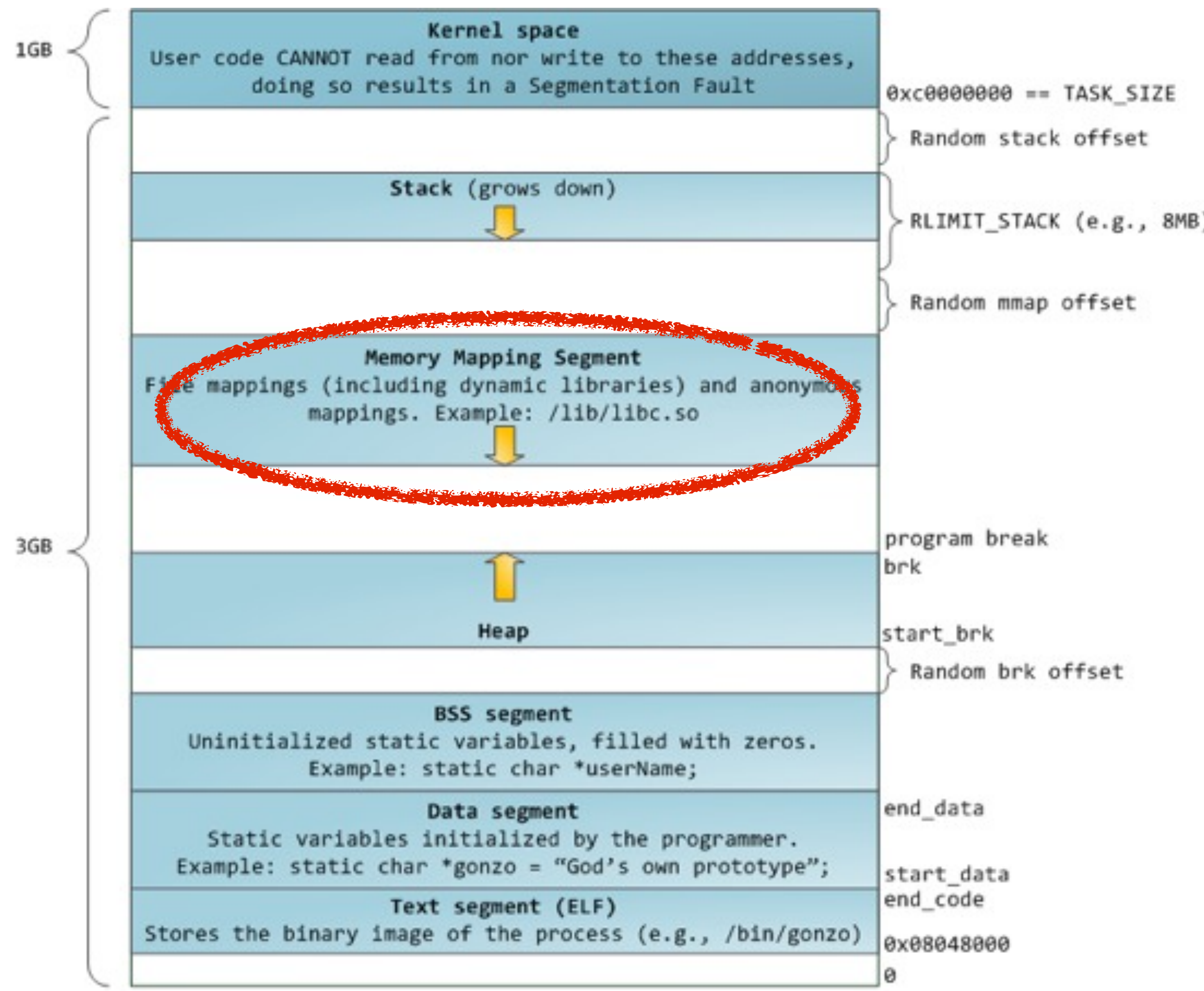


http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Process - What is it?

- Process - A program being executed in Linux
  - Stack
    - Local variables
    - Function return values
    - LIFO
  - Heap
    - “Free-store”
    - Dynamically allocated memory
  - Memory Mapping
    - File mapped in memory
    - Includes dyn libs

Process Memory Layout

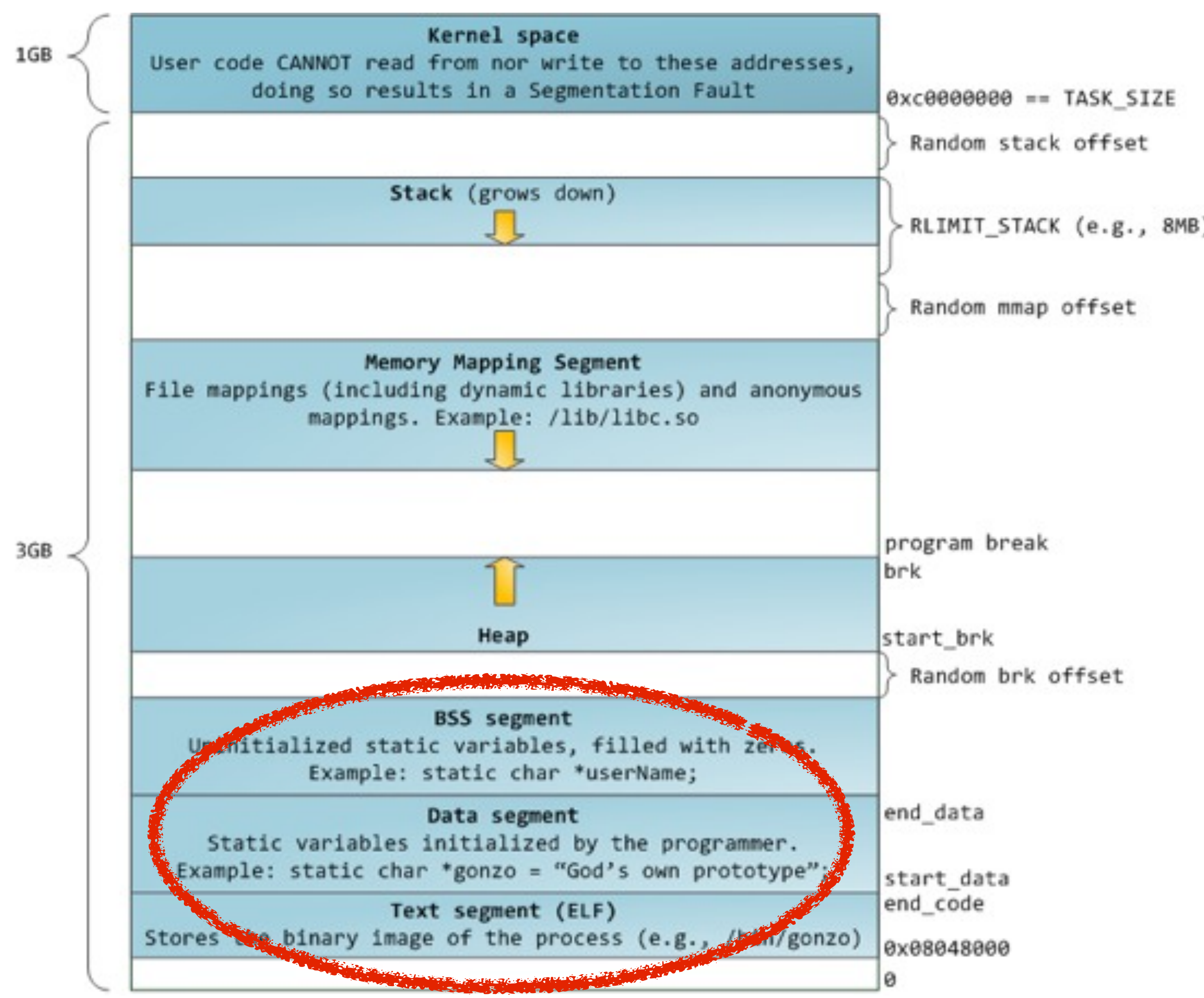


http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Process - What is it?

- Process - A program being executed in Linux
  - Stack
    - Local variables
    - Function return values
    - LIFO
  - Heap
    - “Free-store”
    - Dynamically allocated memory
  - Memory Mapping
    - File mapped in memory
    - Includes dyn libs
  - Variables and ELF

Process Memory Layout



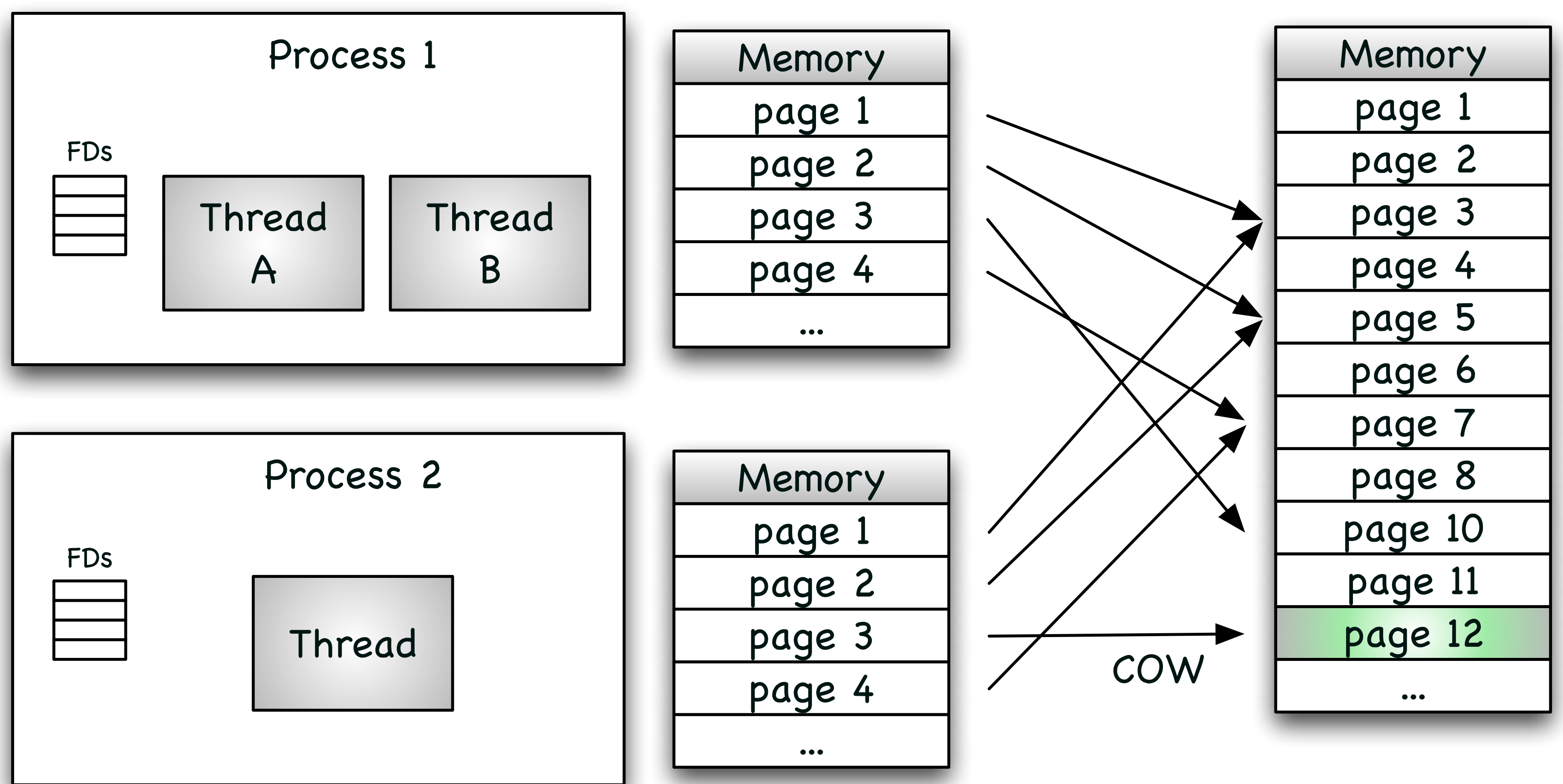
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Forking

- The child gets a new PID
- Virtual address space from parent is replicated (duplicates parents page tables)
  - Includes states of mutexes, conditionals etc.
  - Includes open file descriptors from parent (a copy)
  - Only the thread that calls fork is duplicated in the child
- Only async-safe-signals may be called (including the to be seen exec-family functions)
  - *Functions that don't rely on global variables*

Forking

- All child memory pages are marked Copy On Write (COW)
  - A marked page being written to -> Incurs a page fault, data is copied and then changed.



Exec family of functions

- Inherited or not in child process
  - Not
    - Memory Map
    - Memory Locks
    - Shared Memory
    - Etc.
  - Is
    - ***Open files remain open***

Exec family of functions

- Inherited or not in child process
  - Not
    - Memory Map
    - Memory Locks
    - Shared Memory
    - Etc.
  - Is
    - ***Open files remain open***

Why is this a potential problem?

IPC

ENGINEERING COLLEGE OF AARHUS

Why?

- Inter-Process Communication
  - When do you think or know that you would use
    - Pipes
    - Message Queues
    - Shared Memory
- ***Team up 2 and 2 for 3 mins***

IPC in Linux

IPC in Linux

- "All" IPC must go through the OS

IPC in Linux

- “All” IPC must go through the OS
- Linux and associated libraries provide a (large) set of IPC mechanisms
  - Files
  - ***Pipes (named,*** anonymous***)***
  - ***System V Message Queues***
  - ***Shared Memory***
  - Sockets
  - …

IHA | ENGINEERING COLLEGE OF AARHUS

IPC in Linux

- "All" IPC must go through the OS
- Linux and associated libraries provide a (large) set of IPC mechanisms
  - Files
  - ***Pipes (named,*** anonymous***)***
  - ***System V Message Queues***
  - ***Shared Memory***
  - Sockets
  - …
- We will investigate 3 – named pipes, message queues and shared memory

Pipes

Named pipes

Named pipes

- A named pipe is a half-duplex, point-to-point means of communicating between two processes
  - One process writes data to the pipe, the other reads from it.
  - The system will hold the data until it is read
  - Either party reader or writer are block until other party participates

Named pipes

- A named pipe is a half-duplex, point-to-point means of communicating between two processes
  - One process writes data to the pipe, the other reads from it.
  - The system will hold the data until it is read
  - Either party reader or writer are block until other party participates

- A named pipe, once created, shows up as a file in Linux
  - Both processes must agree on the pipe's (file's) name

Named pipes

- A named pipe is a half-duplex, point-to-point means of communicating between two processes
  - One process writes data to the pipe, the other reads from it.
  - The system will hold the data until it is read
  - Either party reader or writer are block until other party participates
- A named pipe, once created, shows up as a file in Linux
  - Both processes must agree on the pipe's (file's) name
- Named pipes can be used for communication between processes without a common ancestor

Named pipes example

- Server (reader) program

```
int main()
{
   int pipeFd;
   char pipeBuffer[80];

   mkfifo("/tmp/mypipe", 0666);

   pipeFd = open("/tmp/mypipe", O_RDONLY);
   read(pipeFd, pipeBuffer, 80);
   cout << "Received \"" << pipeBuffer << "\"" << endl;

   close(pipeFd);

   remove("/tmp/mypipe");
   return 0;
}
```

Named pipes example

- Server (reader) program

Create pipe node

```
int main()
{
    int pipeFd;
    char pipeBuffer[80];

    mkfifo(“/tmp/mypipe”, 0666);

    pipeFd = open(“/tmp/mypipe”, O_RDONLY);
    read(pipeFd, pipeBuffer, 80);
    cout << "Received \"" << pipeBuffer << "\"" << endl;

    close(pipeFd);

    remove(“/tmp/mypipe”);
    return 0;
}
```

Named pipes example

- Server (reader) program

Create pipe node

Open pipe
Read from pipe
Close pipe

```
int main()
{
   int pipeFd;
   char pipeBuffer[80];

   mkfifo("/tmp/mypipe", 0666);

   pipeFd = open("/tmp/mypipe", O_RDONLY);
   read(pipeFd, pipeBuffer, 80);
   cout << "Received \"" << pipeBuffer << "\"" << endl;

   close(pipeFd);

   remove("/tmp/mypipe");
   return 0;
}
```

Named pipes example

- Server (reader) program

Create pipe node

Open pipe
Read from pipe
Close pipe

Delete pipe node

```
int main()
{
   int pipeFd;
   char pipeBuffer[80];

   mkfifo("/tmp/mypipe", 0666);

   pipeFd = open("/tmp/mypipe", O_RDONLY);
   read(pipeFd, pipeBuffer, 80);
   cout << "Received \"" << pipeBuffer << "\"" << endl;

   close(pipeFd);

   remove("/tmp/mypipe");
   return 0;
}
```

Named pipes example

- Server (reader) program

Create pipe node

Open pipe
Read from pipe
Close pipe

Delete pipe node

```
int main()
{
    int pipeFd;
    char pipeBuffer[80];

    mkfifo("/tmp/mypipe", 0666);

    pipeFd = open("/tmp/mypipe", O_RDONLY);
    read(pipeFd, pipeBuffer, 80);
    cout << "Received \"" << pipeBuffer << "\"" << endl;

    close(pipeFd);

    remove("/tmp/mypipe");
    return 0;
}
```

- Client (writer) program

```
int main()
{
    int pipeFd = open("/tmp/mypipe", O_WRONLY);
    write(pipeFd, "exit", 7);
    close(pipeFd);
    return 0;
}
```

Named pipes example

- Server (reader) program

Create pipe node

Open pipe
Read from pipe
Close pipe

Delete pipe node

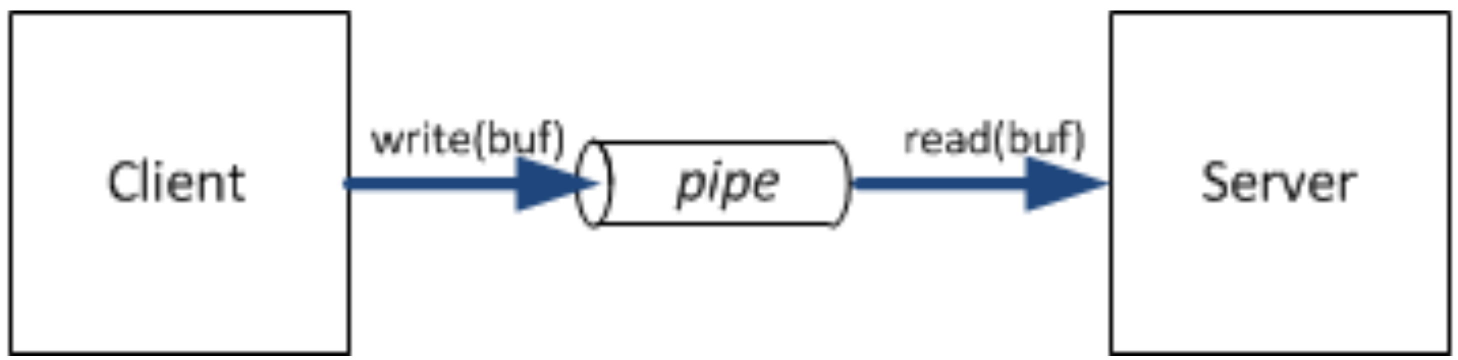
```
int main()
{
   int pipeFd;
   char pipeBuffer[80];

   mkfifo("/tmp/mypipe", 0666);

   pipeFd = open("/tmp/mypipe", O_RDONLY);
   read(pipeFd, pipeBuffer, 80);
   cout << "Received \"" << pipeBuffer << "\"" << endl;

   close(pipeFd);

   remove("/tmp/mypipe");
   return 0;
}
```

- Client (writer) program

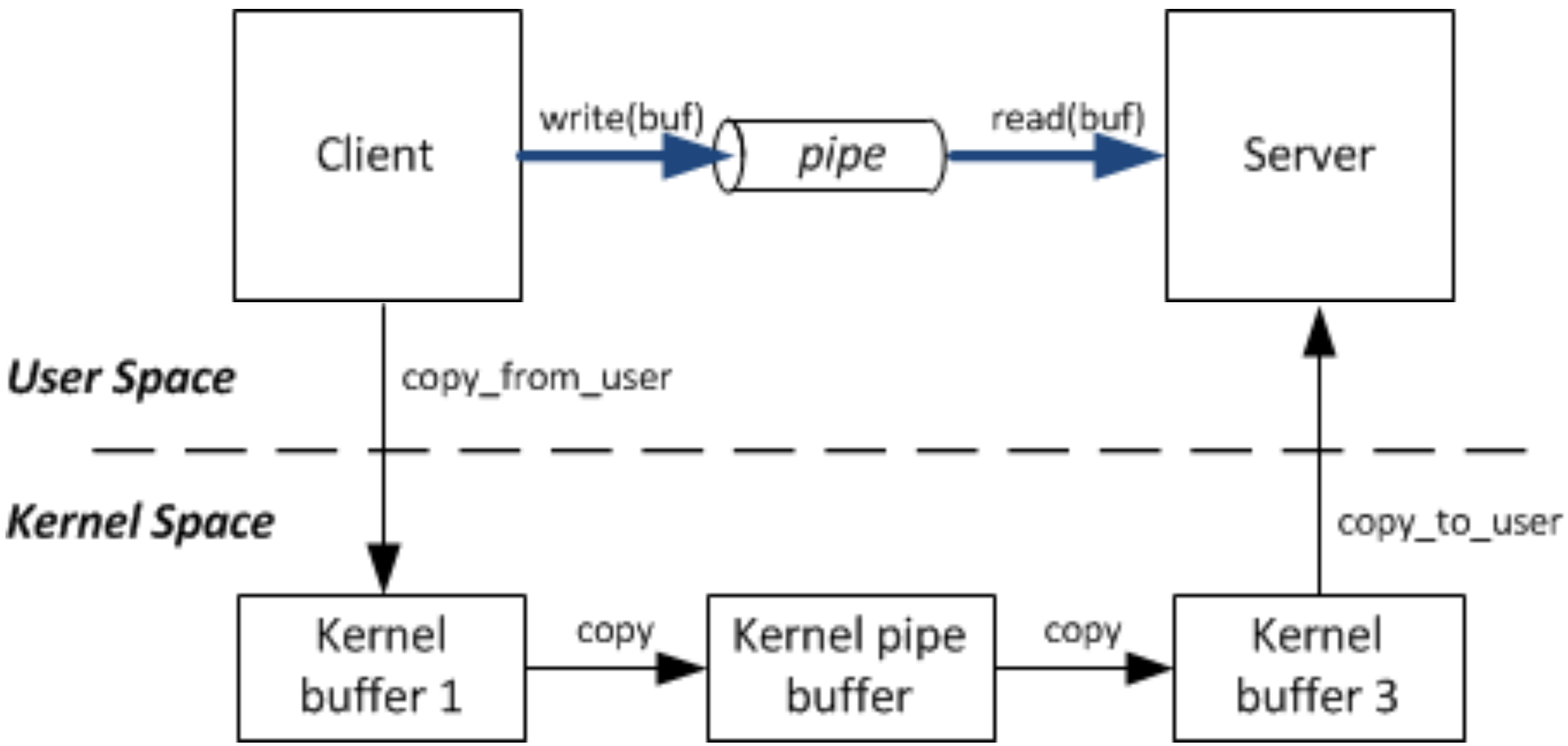
Open pipe
Write to pipe
Close pipe

```
int main()
{
   int pipeFd = open("/tmp/mypipe", O_WRONLY);
   write(pipeFd, "exit", 7);
   close(pipeFd);
   return 0;
}
```

Named pipes and the OS



Named pipes and the OS



Tuesday, May 15, 12

Named Pipes

- The Good
  - File descriptor based -> means using select/poll/epoll
    - Resembles socket communication (to some extend)
    - You are told when the writer dies (close file descriptor)
  - Simple mechanism - also used when controlling slave program e.g.
    - ddd being a frontend for gdb (anonymous pipes)

Named Pipes

- The Bad/Challenge
  - Stream oriented *(depends on usage/design)*
  - What happens if one process dies while reading/writing
    - Synchronization???
  - Data must be *serialized*
  - Do not know the number of “messages” - Does not make sense to talk this way
  - 4 copies
  - Who creates and who destroys?
  - Half-duplex *(depends on usage/design)*

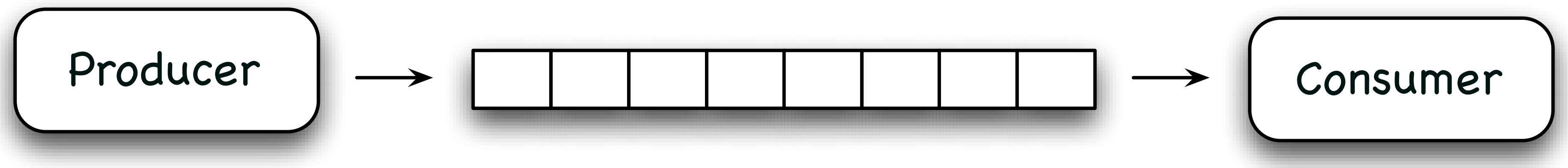
Serialization of data

- Serialization of data is the process of converting process internal data representation to a flat format.
  - Platform agnostic
    - Endianness
  - Flat - *No pointers*
  - Example formats
    - XML
    - JSON
- Local 2 Local communication can be perform using ***C structs*** without pointers
  - The C++ term is *Plain Old Data* (POD) essential good old C structs
  - ***BEWARE THIS IS NOT SOMETHING TO BE TAKEN LIGHTLY***

Message Queues

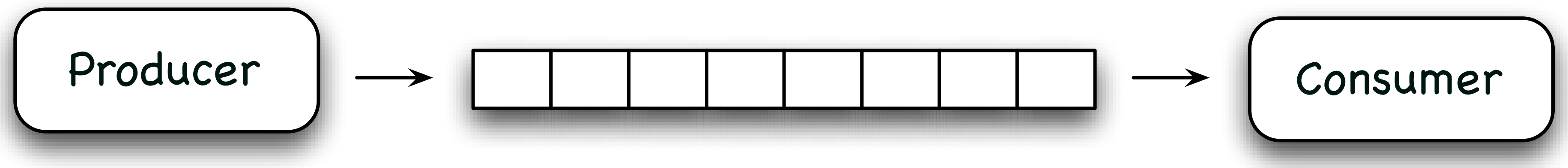
ENGINEERING COLLEGE OF AARHUS

IPC Message Queues



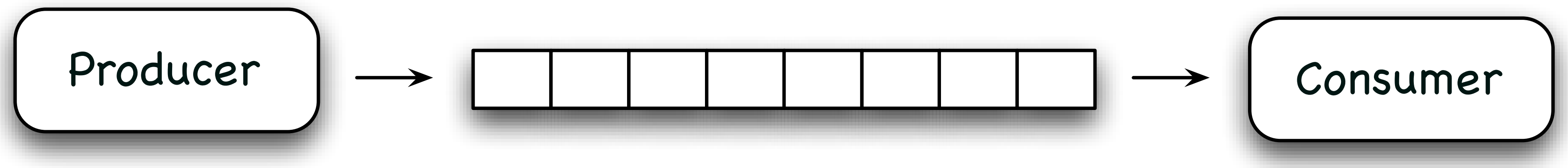
IPC Message Queues

- *Message queues* are used to send *data-bearing messages* between threads in separate processes
  - One-way communication
  - Multiple processes may receive from the queue (unusual)
  - Multiple processes may send to the queue (unusual)



IPC Message Queues

- *Message queues* are used to send *data-bearing messages* between threads in separate processes
  - One-way communication
  - Multiple processes may receive from the queue (unusual)
  - Multiple processes may send to the queue (unusual)
- Message queues are provided by the OS (akin to pipes via the kernel)

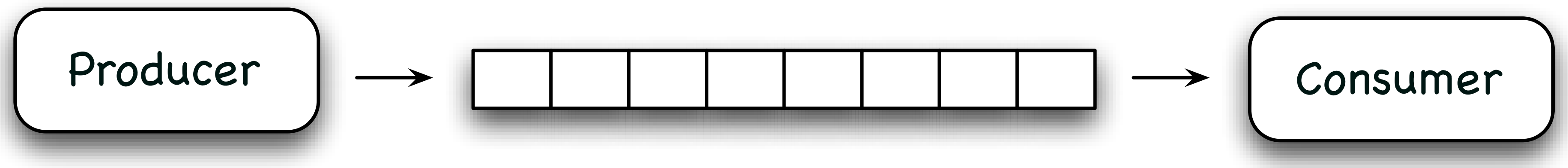


ENGINEERING COLLEGE OF AARHUS

Tuesday, May 15, 12

IPC Message Queues

- *Message queues* are used to send *data-bearing messages* between threads in separate processes
    - One-way communication
    - Multiple processes may receive from the queue (unusual)
    - Multiple processes may send to the queue (unusual)
- Message queues are provided by the OS (akin to pipes via the kernel)
- Threads block on message queues
    - Consumer(s) blocks on *empty* queue
    - Producer(s) blocks on *full* queue



IPC Message Queues in Linux

- System V IPC message queues
  - POSIX Message Queue - Also available

- The API consists of three header files and four functions:

IPC Message Queues in Linux

- System V IPC message queues
  - POSIX Message Queue - Also available

- The API consists of three header files and four functions:

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

int msgget(key_t key, int msgflag);
int msgsnd(key_t msgQId, void* msg_ptr, size_t msgSz, int msgflg);
int msgrcv(key_t msgQId, void* msg_ptr, size_t msgSz, long int msgType, int msgflg);
int msgctl(int msgid, int commadn, struct msqid_ds* buf);
```

System V Message Queues – create and delete

System V Message Queues – create and delete

```
int msgget(
          key_t key,   // the mq number. Existing, system-wide unique or IPC_PRIVATE
          int msgflag  // Permission and creation flags - use 0666 | IPC_CREAT
          );
```

```
int msgctl(
          int key,   // the mq ID as returned from msgget()
          int cmd, // Command - set to IPC_RMID (ReMove ID) to delete
          struct msqid_ds* buf  // Further commands - set to NULL
          );
```

```
int main()
{
   key_t myKey1 = ftok("myTestProgram", 'a');
   key_t myKey2 = ftok("myTestProgram", 'b');
    int mqId1 = msgget(myKey1, 0666 | IPC_CREAT);
    int mqId2 = msgget(myKey2, 0666 | IPC_CREAT);
   ...
   msgctl(mqId1, IPC_RMID, NULL);
   msgctl(mqId2, IPC_RMID, NULL);

   return 0;
}
```

System V Message Queues – create and delete

```
int msgget(
          key_t key,   // the mq number. Existing, system-wide unique or IPC_PRIVATE
          int msgflag  // Permission and creation flags - use 0666 | IPC_CREAT
          );
```

```
int msgctl(
          int key,   // the mq ID as returned from msgget()
          int cmd, // Command - set to IPC_RMID (ReMove ID) to delete
          struct msqid_ds* buf  // Further commands - set to NULL
          );
```

```
int main()
{
   key_t myKey1 = ftok("myTestProgram", 'a');
   key_t myKey2 = ftok("myTestProgram", 'b');
    int mqId1 = msgget(myKey1, 0666 | IPC_CREAT);
    int mqId2 = msgget(myKey2, 0666 | IPC_CREAT);
   ...
   msgctl(mqId1, IPC_RMID, NULL);
   msgctl(mqId2, IPC_RMID, NULL);

   return 0;
}
```

Key generation
Done using inodes from existing file

System V Message Queues – create and delete

```
int msgget(
          key_t key,   // the mq number. Existing, system-wide unique or IPC_PRIVATE
          int msgflag  // Permission and creation flags - use 0666 | IPC_CREAT
          );
```

```
int msgctl(
          int key,   // the mq ID as returned from msgget()
          int cmd, // Command - set to IPC_RMID (ReMove ID) to delete
          struct msqid_ds* buf  // Further commands - set to NULL
          );
```

```
int main()
{
   key_t myKey1 = ftok("myTestProgram", 'a');
   key_t myKey2 = ftok("myTestProgram", 'b');
    int mqId1 = msgget(myKey1, 0666 | IPC_CREAT);
    int mqId2 = msgget(myKey2, 0666 | IPC_CREAT);
   ...
   msgctl(mqId1, IPC_RMID, NULL);
   msgctl(mqId2, IPC_RMID, NULL);

   return 0;
}
```

Key generation
Done using inodes from existing file

Clean-up is important! Message queues will live until deleted or system shut-down, even though your process terminates!

System V Message Queues in Linux – send and receive

- Messages are expected to be a structure:

```
struct Message{
   long int type;
   char data[MSG_LEN];
};
```

- You can also send a struct (or object) – just wrap it in a Message struct:

```
struct MyStruct
{
   ...
};

struct Message{
   long int type;
   MyStruct data;
};
```


IHA | ENGINEERING COLLEGE OF AARHUS

System V Message Queues in Linux – send and receive

- Messages are expected to be a structure:

```
struct Message{
   long int type;
   char data[MSG_LEN];
};
```

The type of message (must be >0)

- You can also send a struct (or object) – just wrap it in a Message struct:

```
struct MyStruct
{
   ...
};

struct Message{
   long int type;
   MyStruct data;
};
```

System V Message Queues in Linux – send and receive

- Messages are expected to be a structure:

```
struct Message{
   long int type;
   char data[MSG_LEN];
};
```

The type of message (must be >0)

The actual message data

- You can also send a struct (or object) – just wrap it in a Message struct:

```
struct MyStruct
{
   ...
};

struct Message{
   long int type;
   MyStruct data;
};
```

System V Message Queues in Linux – send and receive

- Messages are expected to be a structure:

```
struct Message{
   long int type;
   char data[MSG_LEN];
};
```

The type of message (must be >0)

The actual message data

- You can also send a struct (or object) – just wrap it in a Message struct:

```
struct MyStruct
{
   ...
};

struct Message{
   long int type;
   MyStruct data;
};
```

The object you actually wish to send

System V Message Queues in Linux – send and receive

- Messages are expected to be a structure:

```
struct Message{
   long int type;
   char data[MSG_LEN];
};
```

The type of message (must be >0)

The actual message data

- You can also send a struct (or object) – just wrap it in a Message struct:

Object to be serialized
Any criteria for object?

```
struct MyStruct
{
   ...
};

struct Message{
   long int type;
   MyStruct data;
};
```

The object you actually wish to send

System V Message Queues in Linux – send and receive

```
int msgsnd(
   key_t msgQId,
   void* msg_ptr,
   size_t msgSz,
   int msgflg
);
```

```
int msgrcv(
   key_t msgQId,
   void* msg_ptr,
   size_t msgSz,
   long int msgType,
   int msgflg
);
```

```
struct Message{
   long int type;
   char data[MSG_LEN];
};
```

System V Message Queues in Linux – send and receive

```
int msgsnd(
   key_t msgQId,
   void* msg_ptr,
   size_t msgSz,
   int msgflg
);
```

The message queue to send/rec through

```
int msgrcv(
   key_t msgQId,
   void* msg_ptr,
   size_t msgSz,
   long int msgType,
   int msgflg
);
```

```
struct Message{
   long int type;
   char data[MSG_LEN];
};
```

System V Message Queues in Linux – send and receive

```c
int msgsnd(
    key_t msgQId,
    void* msg_ptr,
    size_t msgSz,
    int msgflg
);
```

The message queue to send/rec through

The message to send/receive

```c
int msgrcv(
    key_t msgQId,
    void* msg_ptr,
    size_t msgSz,
    long int msgType,
    int msgflg
);
```

```c
struct Message{
    long int type;
    char data[MSG_LEN];
};
```

System V Message Queues in Linux – send and receive

```c
int msgsnd(
    key_t msgQId,
    void* msg_ptr,
    size_t msgSz,
    int msgflg
);
```

```c
int msgrcv(
    key_t msgQId,
    void* msg_ptr,
    size_t msgSz,
    long int msgType,
    int msgflg
);
```

```c
struct Message{
    long int type;
    char data[MSG_LEN];
};
```

The message queue to send/rec through

The message to send/receive

The size of data in the message itself
(excl. the ***type*** field)

System V Message Queues in Linux – send and receive

```c
int msgsnd(
    key_t msgQId,
    void* msg_ptr,
    size_t msgSz,
    int msgflg
);
```

```c
int msgrcv(
    key_t msgQId,
    void* msg_ptr,
    size_t msgSz,
    long int msgType,
    int msgflg
);
```

```c
struct Message{
    long int type;
    char data[MSG_LEN];
};
```

The message queue to send/rec through

The message to send/receive

The size of data in the message itself (excl. the ***type*** field)

Message flags (set to 0)

System V Message Queues in Linux – send and receive

```
int msgsnd(
    key_t msgQId,
    void* msg_ptr,
    size_t msgSz,
    int msgflg
);
```

```
int msgrcv(
    key_t msgQId,
    void* msg_ptr,
    size_t msgSz,
    long int msgType,
    int msgflg
);
```

- The message queue to send/rec through
- The message to send/receive
- The size of data in the message itself (excl. the ***type*** field)
- Message flags (set to 0)
- The type of message to receive (set to 0)

```
struct Message{
    long int type;
    char data[MSG_LEN];
};
```


ENGINEERING COLLEGE OF AARHUS

System V Message Queues in Linux – send and receive

- Example:

```
int main()
{
   MessageType1 msg1;
   MessageType2 msg2;

   key_t myKey1 = ftok("myTestProgram", 'a');
   key_t myKey2 = ftok("myTestProgram", 'b');

   int mqId1 = msgget(myKey1, 0666 | IPC_CREAT);
   int mqId2 = msgget(myKey2, 0666 | IPC_CREAT);

   msgsnd(mqId1, (void*) &msg1, sizeof(MessageType1.data), 0);
   ...
   msgrcv(mqId2, (void*) &msg2, sizeof(MessageType2.data), 0, 0);

   ...

   msgctl(mqId1, IPC_RMID, NULL);
   msgctl(mqId2, IPC_RMID, NULL);

   return 0;
}
```

System V Message Queues

- The Good
  - Structured/packet data - Complete chunk received
    - Either you have it or you don't
  - Priority based
  - Event driven (*Design related*)

ENGINEERING COLLEGE OF AARHUS

System V Message Queues

- The Bad
  - You have to create some means of synchronization aka. sockets
    - Other party has died -> handle internal state appropriate
    - Two queues for duplex communication

Named pipes vs. Message queues

Named pipes vs. Message queues

- Named pipe or Message Queue – what's the difference?

Named pipes vs. Message queues

- Named pipe or Message Queue – what's the difference?
- Named pipe
  - Only two processes (can be related or unrelated) can communicate.
  - Data read from FIFO is first in first out manner.

Named pipes vs. Message queues

- Named pipe or Message Queue – what's the difference?
- Named pipe
  - Only two processes (can be related or unrelated) can communicate.
  - Data read from FIFO is first in first out manner.
- Message queues:
  - Any number of processes can read/write from/to the queue.
  - Data can be read selectively. (need not be in FIFO manner)

Named pipes vs. Message queues

- Named pipe or Message Queue – what's the difference?
- Named pipe
  - Only two processes (can be related or unrelated) can communicate.
  - Data read from FIFO is first in first out manner.
- Message queues:
  - Any number of processes can read/write from/to the queue.
  - Data can be read selectively. (need not be in FIFO manner)
- Named pipes are built on message queues (!)

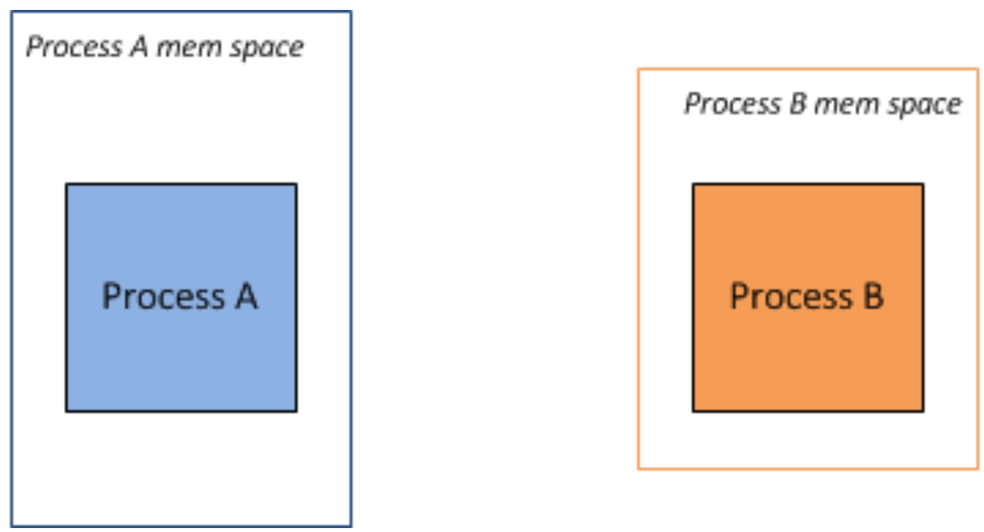
Named pipes vs. Message queues

- Named pipe or Message Queue – what’s the difference?
- Named pipe
  - Only two processes (can be related or unrelated) can communicate.
  - Data read from FIFO is first in first out manner.
- Message queues:
  - Any number of processes can read/write from/to the queue.
  - Data can be read selectively. (need not be in FIFO manner)
- Named pipes are built on message queues (!)
- Message queues are faster than pipes

Shared Memory

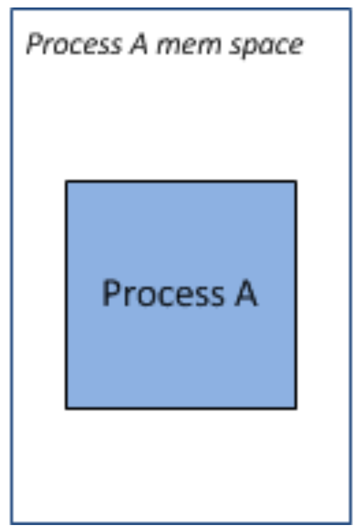
Shared Memory

- Generally, each process has it's own memory space

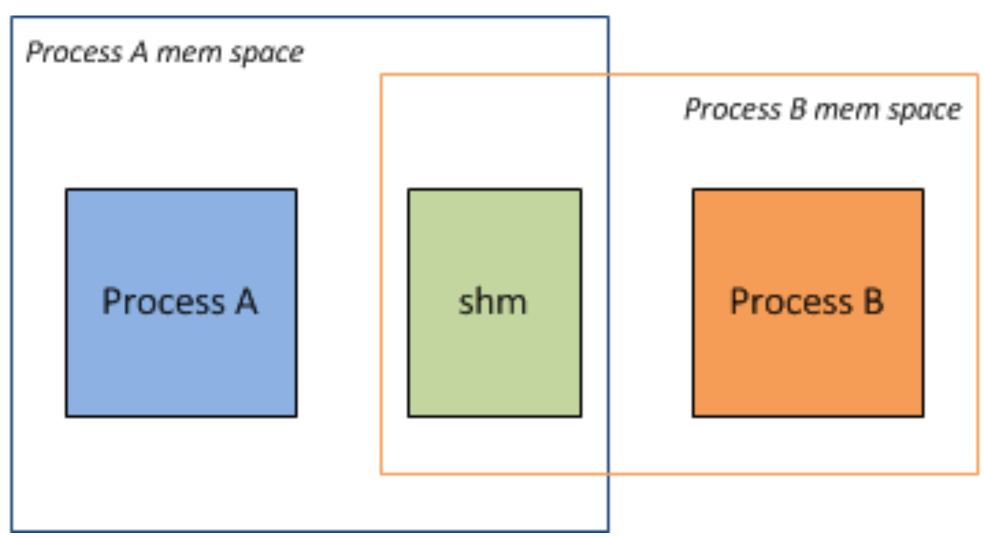


Shared Memory

- Generally, each process has it's own memory space



- However, processes can agree to create and both include a section of memory – this section is called shared memory (shm)



ENGINEERING COLLEGE OF AARHUS

System V Shared Memory

- System V Shared Memory API

System V Shared Memory

- System V Shared Memory API

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

int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Shared Memory example (System V shm)

- Server (reader) program

```
const unsigned int SHM_KEY(5678);

int main()
{
    int shmid = shmget(SHM_KEY, SHMSZ, IPC_CREAT | 0666);
    char* shm = (char*)shmat(shmid, (void*)0, 0);

    cout << "Reader read: " << shm << "\n";
    *shm = '*';

    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

Shared Memory example (System V shm)

- Server (reader) program

Create and attach to SHM

R/W SHM

Clean up

```
const unsigned int SHM_KEY(5678);

int main()
{
    int shmid = shmget(SHM_KEY, SHMSZ, IPC_CREAT | 0666);
    char* shm = (char*)shmat(shmid, (void*)0, 0);

    cout << "Reader read: " << shm << "\n";
    *shm = '*';

    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

Shared Memory example (System V shm)

- Server (reader) program

Create and attach to SHM

R/W SHM

Clean up

```
const unsigned int SHM_KEY(5678);

int main()
{
    int shmid = shmget(SHM_KEY, SHMSZ, IPC_CREAT | 0666);
    char* shm = (char*)shmat(shmid, (void*)0, 0);

    cout << "Reader read: " << shm << "\n";
    *shm = '*';

    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

- Client (writer) program

```
const unsigned int SHM_KEY(5678);

int main()
{
    char c;
    int shmid = shmget(SHM_KEY, SHMSZ, IPC_CREAT | 0666);
    char* shm = (char*)shmat(shmid, (void*)0, 0);

    char *temp = shm;
    for (c = 'a'; c <= 'z'; c++) *temp++ = c;
    *temp = NULL;

    while (*shm != '*') sleep(1);

    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

Shared Memory example (System V shm)

- Server (reader) program

Create and attach to SHM

R/W SHM

Clean up

```
const unsigned int SHM_KEY(5678);

int main()
{
    int shmid = shmget(SHM_KEY, SHMSZ, IPC_CREAT | 0666);
    char* shm = (char*)shmat(shmid, (void*)0, 0);

    cout << "Reader read: " << shm << "\n";
    *shm = '*';

    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

- Client (writer) program

Create and attach to SHM

R/W SHM

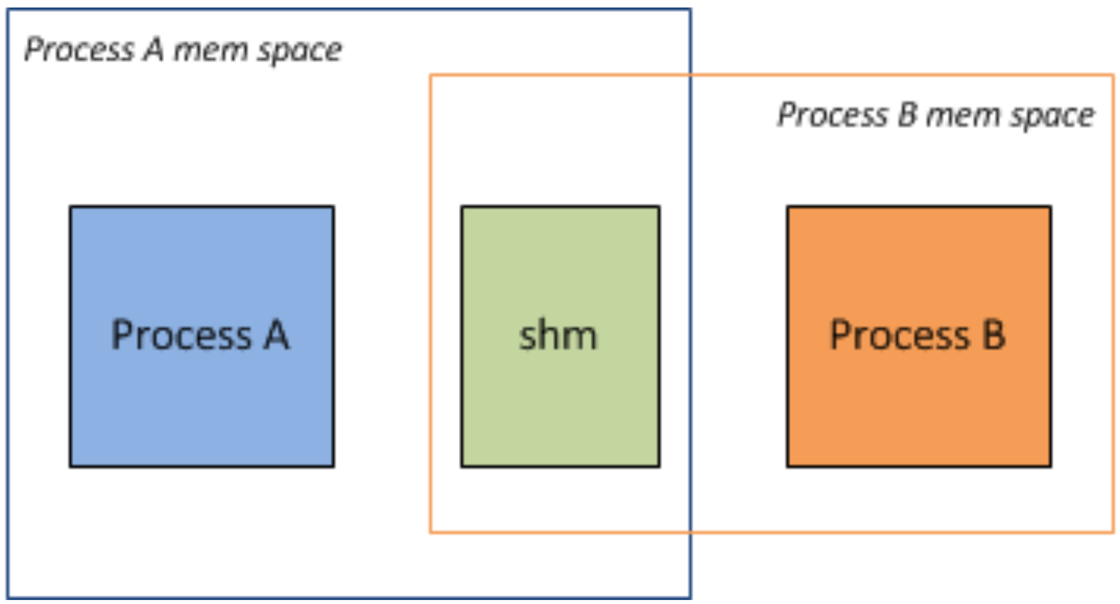
Clean up

```
const unsigned int SHM_KEY(5678);

int main()
{
    char c;
    int shmid = shmget(SHM_KEY, SHMSZ, IPC_CREAT | 0666);
    char* shm = (char*)shmat(shmid, (void*)0, 0);

    char *temp = shm;
    for (c = 'a'; c <= 'z'; c++) *temp++ = c;
    *temp = NULL;

    while (*shm != '*') sleep(1);

    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

Shared Memory

- The SHM segment is *created* and *included* using system calls
  - **shmget()** – create SHM segment
  - **shmat()** – attach SHM segment to process' mem space

- Once created and included, R/W access is entirely in user space through obtained pointers – very fast!



- Mutual exclusion (if desired) must be enforced by the processes themselves

Mutexes in Shared Memory

- shared_mutex must be pre-allocated in shared memory

Mutexes in Shared Memory

- shared_mutex must be pre-allocated in shared memory

Init attributes

```
#include <pthread.h>

// Function signatures
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);

// Core code extraction
pthread_mutex_t* shared_mutex; // Placed in shared memory

pthread_mutexattr_t mutex_attr;
pthread_mutexattr_init(&mutex_attr);
pthread_mutexattr_setpshared(&mutex_attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(shared_mutex, &mutex_attr);
```

Mutexes in Shared Memory

- shared_mutex must be pre-allocated in shared memory

Init attributes

Set mutex as shared

```c
#include <pthread.h>

// Function signatures
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);

// Core code extraction
pthread_mutex_t* shared_mutex; // Placed in shared memory

pthread_mutexattr_t mutex_attr;
pthread_mutexattr_init(&mutex_attr);
pthread_mutexattr_setpshared(&mutex_attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(shared_mutex, &mutex_attr);
```

Mutexes in Shared Memory

- shared_mutex must be pre-allocated in shared memory

```c
#include <pthread.h>

// Function signatures
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);

// Core code extraction
pthread_mutex_t* shared_mutex; // Placed in shared memory

pthread_mutexattr_t mutex_attr;
pthread_mutexattr_init(&mutex_attr);
pthread_mutexattr_setpshared(&mutex_attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(shared_mutex, &mutex_attr);
```

Init attributes

Set mutex as shared

Init mutex with attrs

Conditionals in Shared Memory

- shared_conditional must be pre-allocated in shared memory

Conditionals in Shared Memory

- shared_conditional must be pre-allocated in shared memory

```
#include <pthread.h>

// Function signatures
int pthread_condattr_setpshared(const pthread_condattr_t *restrict attr,
                                int pshared);

// Core code extraction
pthread_cond_t* shared_condvar; // Placed in shared memory

pthread_condattr_t cond_attr;
pthread_condattr_init(&cond_attr);
pthread_condattr_setpshared(&cond_attr, PTHREAD_PROCESS_SHARED);
pthread_cond_init(shared_condvar, &cond_attr);
```

Conditionals in Shared Memory

- shared_conditional must be pre-allocated in shared memory

Init attributes

```
#include <pthread.h>

// Function signatures
int pthread_condattr_setpshared(const pthread_condattr_t *restrict attr,
                                int pshared);

// Core code extraction
pthread_cond_t* shared_condvar; // Placed in shared memory

pthread_condattr_t cond_attr;
pthread_condattr_init(&cond_attr);
pthread_condattr_setpshared(&cond_attr, PTHREAD_PROCESS_SHARED);
pthread_cond_init(shared_condvar, &cond_attr);
```

Conditionals in Shared Memory

- shared_conditional must be pre-allocated in shared memory

Init attributes

Set conditional as shared

```
#include <pthread.h>

// Function signatures
int pthread_condattr_setpshared(const pthread_condattr_t *restrict attr,
                                int pshared);

// Core code extraction
pthread_cond_t* shared_condvar; // Placed in shared memory

pthread_condattr_t cond_attr;
pthread_condattr_init(&cond_attr);
pthread_condattr_setpshared(&cond_attr, PTHREAD_PROCESS_SHARED);
pthread_cond_init(shared_condvar, &cond_attr);
```

Conditionals in Shared Memory

- shared_conditional must be pre-allocated in shared memory

```c
#include <pthread.h>

// Function signatures
int pthread_condattr_setpshared(const pthread_condattr_t *restrict attr,
                                int pshared);

// Core code extraction
pthread_cond_t* shared_condvar; // Placed in shared memory

pthread_condattr_t cond_attr;
pthread_condattr_init(&cond_attr);
pthread_condattr_setpshared(&cond_attr, PTHREAD_PROCESS_SHARED);
pthread_cond_init(shared_condvar, &cond_attr);
```

Init attributes

Set conditional as shared

Init conditional with attrs

Shared Memory

- The Good
  - Extremely fast since communicating processes write directly to the same memory space
    - Reduces the need for memory copies drastically
  - Multiple processes may share the same memory
    - Aka. threads in a process

Shared Memory

- The Bad/Challenge
  - Synchronization as between threads
  - What happens if one process dies while reading/writing
    - Semaphore/Mutex/Conditional locked and the program crashes... what next?
  - Who creates and who destroys?
    - When are shared structures ready for use?
  - You have no way of knowing if recipient is dead
    - Requires extra control
      - pipe or socket
  - Data must be *flat* or internal pointers must point to shared memory areas

IPC when?

IPC When?

- Pipes
  - Point 2 Point communication
  - Streamed data
  - Very simple and very portable
- System V Message Queues
  - Structured/Packet based with priority
  - Event driven
- Shared Memory
  - Extremely fast memory transfer between processes
  - Willing to pay for the extra coupling and control
  - N-way communication